Preface

This volume is the supplemental proceedings of the *13th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2000)* held in Portland, Oregon, USA during August 14–18, 2000. In keeping with tradition, TPHOLs 2000 offers this venue for the presentation of work in progress, where researchers invite discussion by means of a brief preliminary talk and then discuss their work at a poster session. The supplementary proceedings is published by the Oregon Graduate Institute (OGI) as technical report CSE-00-009.

The TPHOLs conference traditionally changes continent each year, to maximize the chances of researchers all over the world to attend. The proceedings of *TPHOLs* or its predecessor have been published in the following volumes of the Springer-Verlag *Lecture Notes in Computer Science* series since 1993:

1993 (Canada)	780	1997 (USA)	1275
1994 (Malta)	859	1998 (Australia)	1479
1995 (USA)	971	1999 (France)	1690
1996 (Finland)	1125	2000 (USA)	1869

The 2000 conference was organized by a team from Intel Corporation and the Oregon Graduate Institute. Financial support came from Compaq, IBM, Intel, Levetate, Synopsys and OGI. A generous grant from the National Science Foundation allowed the organizers to offer bursaries covering part of the cost of attending TPHOLs for students. The support of all these organizations is gratefully acknowledged.

July, 2000

Mark Aagaard, John Harrison, Tom Schubert

Conference Organization

Mark Aagaard (General Chair)
Kelly Atkinson
Robert Beers
Nancy Day
John Harrison (Program Chair)
Naren Narasimhan
Tom Schubert

Program Committee

Mark Aagaard (Intel)
Flemming Andersen (IBM)
David Basin (Freiburg)
Richard Boulton (Glasgow)
Gilles Dowek (INRIA)
Harald Ganzinger (Saarbrucken)
Ganesh Gopalakrishnan (Utah)
Mike Gordon (Cambridge)
Jim Grundy (ANU)
Elsa Gunter (Bell Labs)
John Harrison (Intel)
Doug Howe (Ottawa)
Warren Hunt (IBM)
Bart Jacobs (Nijmegen)
Paul Jackson (Edinburgh)
Steve Johnson (Indiana)
Sara Kalvala (Warwick)
Tom Melham (Glasgow)
Paul Miner (NASA)
Tobias Nipkow (München)
Sam Owre (SRI)
Christine Paulin-Mohring (INRIA)
Lawrence Paulson (Cambridge)
Klaus Schneider (Karlsruhe)
Sofiene Tahar (Concordia)
Ranga Vemuri (Cincinnati)

Contents

Author Index

Deriving a special-purpose prover for compositional model checking in Coq

Pablo Argon[1,2] and Kenneth McMillan[2]

[1] INRIA
[2] Cadence Berkeley Labs
2001 Addison St.
Berkeley, CA 94702
USA
{pargon|mcmillan}@cadence.com

Abstract. We describe our current ongoing work using the Coq Proof Assistant to formally prove the soundness of the proof decomposition rules implemented in the SMV system. Our final goal is not only to verify this rules, but eventually to extract the code implementing these rules, using the extraction mechanism of Coq.

1 Introduction

Model checking provides an effective method for the formal verification of finite state systems. In practice, however, it is limited for complexity reasons to fairly small systems. To handle larger systems, and systems whose state space is not finitely bounded, we can apply compositonal methods. For example, the SMV proof system [10] provides a collection of proof decomposition and reduction rules that can reduce a proof about a complex, infinite-state system to a finite number of sub-goals that can be discharged by model checking.

The philosophy behind the SMV proof system is that a special-purpose prover tailored to particular style of proof will be more efficient and easier to use that a general-purpose system. This has been borne out to some extent in the application of the system to hardware designs. For example, the system yields a relatively simple proof of Tomasulo's algorithm for out-of-order instruction execution, when compared to general purpose provers [10]. A proof strategy based on these techniques has also been used, for example, to verify the implementation of a multiprocessor cache coherence protocol in low-level hardware [5].

However, a disadvantage of this approach is that the proof rules used by the system are considerably more complex in their implementation than those of a general purpose prover. Thus, a programming error in the implementation can easily lead to an unsoundness in the system. At present, the system is implemented in an ad-hoc manner in the C programming language, and in fact, numerous cases of unsoundness have been found in this implementation during its development.[1] For this reason, we have begun an effort to embed the system in the Coq prover [2], and to extract an implementation of it from its proof of correctness. Besides improving the security of the system, this embedding may also provide a means of extensibility, allowing derived proof rules to be added by the user.

Related work There is now a fairly wide literature on combining model checking and theorem proving. Most of it concerns the integration of a model checker as a decision procedure inside a theorem prover [16, 12, 8]. A concrete problem version, (very large or infinite), is defined using the theorem prover specification language, and then (a proved correct) finite abstract version of the problem is submitted to the model checker [4, 3].

[1] For an interesting case of an unsoundness remaining in the implementation of another theorem prover for a considerable period of time, see the draft by N. Shankar at `http://www.csl.sri.com/shankar/shostak2000.ps.gz`.

In this work, by contrast, we propose to use a general-purpose theorem prover to derive special-purpose prover supporting a particular proof methodology. The prover will be derived from a proof of correctness of a collection of proof decomposition rules. For this purpose, we are using the Coq theorem prover because of its capability to specify, prove and then extract programs from proved terms [13, 15]. This technique of program extraction has been succesfully used for the proof of small and well-known algorithms [14, 7], and also for some non-trivial examples [6] and medium size systems [1, 17].

2 The SMV proof system

SMV provides a proof system based on a first-order temporal logic. The proof system is designed to make it as straightforward as possible to reduce a complex verification problem to a collection of "finite state" proof subgoals that can be discharged by a model checker. A collection of proof decomposition and reduction techniques are provided by the system for this purpose:

Circular compositional rule Proofs are decomposed structurally using a "circular compositional proof" technique. In effect, this allows us to prove a collection of temporal propositions by mutual induction over time [9]. The mutual induction is required because components of a system usually must assume their input is correct up to time $t-1$ in order to guarantee that their output is correct at time t. For example, suppose we have two temporal properties $\phi_1(t)$ and $\phi_2(t)$. The circular compositional rule tells us that the following inference is sound: $\forall t: (\phi$

$$\frac{(\forall t' < t: \ \phi_1(t')) \Rightarrow \phi_2(t) \qquad (\forall t' < t: \ \phi_2(t')) \Rightarrow \phi_1(t)}{{}_1(t) \wedge \phi_2(t))}$$

Since the antecedents can be expressed in temporal logic, it is possible to discharge them automatically by model checking. This rule can be generalized to any number of temporal properties and also to "zero-delay" dependencies, provided the zero-delay dependence relation is well-founded. Typically the properties proved in this way are "refinement relations", specifying some desired relationship between an abstract reference model and an implementation. Specifying refinement maps for internal variables of the implementation allows the localization of proof sub-goals to small components of the implementation. This localization is the primary method of decomposing a complex problem into model checking subgoals of tractable size.

Conservative extension The system provides a mechanism of conservative extension that supports definitions that are mutually inductive over time. This makes it possible, for example, to define reference models, and to add "auxiliary" state variables into the system. These variables are generally used to store some history information, and assist in writing refinement relations specifying internal system variables. The system verifies that the definitions are well founded in the sense that every dependency cycle involves at least one time unit of delay.

Parameterization Another important method of decomposing the proof of a property is to introduce a parameter. Verifying the property for any particular value of the parameter can be done using a more abstract model of the system than is required to prove the general property. For example, suppose we wish to prove the correctness of a memory subsystem. Typically we parameterize the problem on the address being read at the current time. For any particular address, we abstract away the state of all but a single memory location, thus greatly simplify the verification problem. Parameterization of a property is accomplished using a "temporal case splitting" rule, which allows us to make the following inference:

$$\frac{\forall i \in T: \ \forall t: \ ((v(t) = i) \Rightarrow \phi(t))}{\forall t: \ \phi(t)}$$

where v is a temporal variable of type T. That is, for each value of the parameter i, we must show that ϕ holds at those times t when v has the value i.

Data type reduction The system has a collection of conservative abstractions that it can use to reduce large or infinite state spaces to small finite state spaces [10]. These abstractions are usually relative

to a choice of parameter values. For example, in the verification of our memory subsystem, the parameter is the address currently being read. For a particular value a of this parameter, the type "address" might be reduced to just two values: a and an abstract value representing all the address not equal to a. A suitable abstract interpretation of the logic guarantees conservatism, that is, that any property proved in the abstract model is true in the original model.

Exploitation of symmetry Finally, the number of parameter valuations that must be checked can be reduced to a tractably small number by exploiting symmetry. This is done by introducing symmetric data types. Type checking rules guarantee that the semantics of formulas is invariant under permutations of these types. This makes it possible to choose a finite set of representative parameter values for any given symmetric type, such that every parameter valuation can be reduced to a representative one by permuting the type. Thus, the combination of parameterization, data type reduction and symmetry makes it possible to prove properties of infinite state spaces by checking a finite number of formulas on finite models. A similar abstraction and case reduction technique makes it possible to prove properties by induction over the natural numbers [11].

3 Embedding the SMV proof system in Coq

We will now consider the problem of embedding the SMV proof system in the Coq prover, and extracting an implementation of it from its proof of correctness. We should first note that the entire SMV proof system is probably too complex to allow it to be formally derived in this way. Our intention is, rather, to identify a kernel of the system that can be developed formally, while the remainder of the system (the greater part) is developed informally. The overall system is structured roughly as shown in figure 1. A compiler first translates the SMV external syntax into formulas of a much simpler base logic. This removes many language constructs that can be viewed as "syntactic sugar". This process produces a specification and implementation in the base logic, as well as various control declarations. The latter, combined with a collection of heuristics, determine the proof decomposition, which in turn yields a set of finite-state subgoals to be discharged by a model checker. We intend to develop an implementation

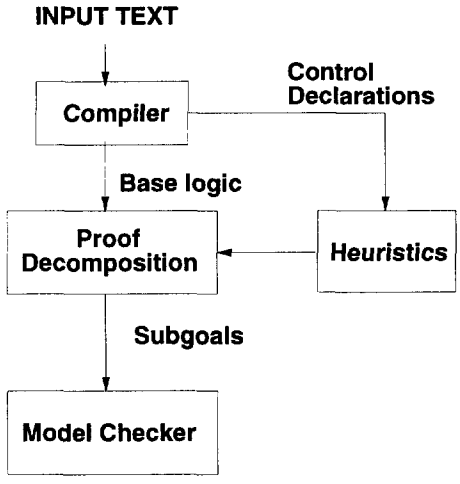

Fig. 1. SMV system structure

of the "proof decomposition" component of the system formally in order to provide a strong guarantee of security of this part of the system. One fact that should prove helpful in this regard is that we do

not require a highly efficient implementation of this part of the system, since its run time tends to be dominated by other components. This, of course, still leaves significant security risk in other areas. Note, however, that while a large part of the complexity of the system is in the heuristics, the soundness of the system does not depend on their correctness. Thus, while an error in this code could prove very frustrating for a user, it cannot result in "proving" a non-theorem. Therefore we consider security of this code to be of a lower priority. The model checker, on the other hand is critical to the soundness of the system. Unfortunately, in this case execution efficiency is critical. Thus, it seems more practical to use a model checker implemented at a very low level in the C language, and to rely for security on the fact that the model checker is a well established technology and has been in use for a long period of time. Finally, the "compiler" also presents a significant security risk. Since its implementation is considerably less subtle than that of the proof decomposition component, we consider it a lower priority for formalization.

In short, we would like to obtain a compromise that provides substantial improvement in the robustness of the system in exchange for a reasonable effort in formalization.

4 Progress to date, and future work

We have begun at semantic level (i.e., with a shallow embedding of the logic in the Coq prover), and have thus far proved theorems underlying the "circular compositonal" proof technique and the soundess of conservative extension. Some results have also been obtained regarding the symmetry reduction technique. In order to continue, however, it will be necessary to reason about syntax, and thus to provide a deep embedding of the logic. It remains, in addition, to prove correctness of the parameterization and data type reduction steps, as well as a variety of transformation steps that are used in the process of generating subgoals. Our objective is to extract an implementation of the proof decomposition component of the system that inputs a specification and implementation in the base logic, and outputs formulas to be verified on finite models. At this point it is far from clear that this can be achieved. However, we expect that the formalization effort will at the very least help to clarify the theory underlying the system, and may lead to a cleaner and more robust implementation even if we do not obtain a complete proof of correctness.

References

1. Pablo Argon, John Mullins, and Olivier Roux. Correct compiler construction using coq. In *CADE-13 Workshop on Proof Search in Type Theoretic Languages*, New Brunswick (USA), July 1996. Rutgers University.
2. The Coq Proof Assistant. `http://coq.inria.fr`.
3. Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331, Vancouver, Canada, June 1998. Springer-Verlag.
4. Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A tool for the verification of invariants. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 505–510, Vancouver, Canada, June 1998. Springer-Verlag.
5. À. P. Eiríksson. The formal design of 1M-gate ASICs. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in System Design (FMCAD '98)*, volume 1522 of *LNCS*, pages 49–63, 1998.
6. J.-C. Filliâtre. Proof of Imperative Programs in Type Theory. Research Report 97–24, LIP - ENS Lyon, July 1997.
7. J.-C. Filliâtre and N. Magaud. Certification of sorting algorithms in the system Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.
8. Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.
9. K. L. McMillan. Circular compositional reasoning about liveness. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 342–345, 1999.

10. K. L. McMillan. Verification of an infinite state systems by compositional model checking. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 219–233, 1999.
11. K. L. McMillan, S. Qadeer, and J. Saxe. Induction in compositional model checking, 2000. to appear.
12. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
13. C. Parent. Developing certified programs in the system Coq - the program tactic. In H. Barendregt and T. Nipkow, editors, *Proceedings of Types'93*, 1994. galement rapport LIP-ENS Lyon RR 93-29 and by anonymous ftp from lip.ens-lyon.fr file pub/Rapports/RR/RR93/RR93-29.ps.Z.
14. C. Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives.* thèse d'université, École Normale Supérieure de Lyon, January 1995.
15. C. Parent. Synthesizing proofs from programs in the calculus of inductive constructions. In *Third International Conference on the Mathematics of Program Construction*, number 947 in Lecture Notes in Computer Science. Springer-Verlag, Juillet 1995.
16. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.
17. Delphine Terrasse-Kaplan. Vers la certification du compilateur v5 d'ESTEREL. Rapport de post-doctorat., September 1996.

Towards a Library of Formal Mathematics

Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, Irene Schena
Department of Computer Science
Via di Mura Anteo Zamboni 7, 40127 Bologna, ITALY.
contact: asperti@cs.unibo.it

Abstract

The eXtensible Markup Language (XML) opens the possibility to start anew, on a solid technological ground, the ambitious goal of developing a suitable technology for the creation and maintenance of a virtual, distributed, hypertextual library of formal mathematical knowledge. In particular, XML provides a central technology for storing, retrieving and processing mathematical documents, comprising sophisticated web-publishing mechanisms (stylesheets) covering notational and stylistic issues. In this paper, we discuss the overall architectural design of the new systems, and our progress in this direction (http://www.cs.unibo.it/~asperti/HELM/home.html).

1 Introduction

Existing logical systems are not suitable for the creation of large repositories of structured mathematical knowledge accessible via Web. In fact, libraries in logical frameworks are usually saved in two formats: a textual one, in the specific tactical language of the proof assistant, and a compiled (proof checked) one in some internal, concrete representation language. Both representations are clearly unsatisfactory, since they are too oriented to the specific application: the information is not directly available, if not by means of the functionalities offered by the system itself. This is in clear contrast with the main guidelines of the modern Information Society, and with its new emphasis on *content*. Moreover, the information provided by such libraries usually lacks of a satisfactory form of presentation. This is a separate, still parallel aspect which is undoubtedly fundamental to achieve a significant dissemination of mathematical knowledge.

The eXtensible Markup Language (XML, see [2]), whose aim is to encode information according to its structure and content, is rapidly imposing as the main tool for representation, manipulation, linking and exchange of structured information in the networked age. In this paper we advocate the pivotal role of XML in the development of a suitable technology for the creation and maintenance of large repositories of structured mathematical knowledge, and describe the overall architectural design of the new systems.

The feasibility of describing mathematical structures using a markup language is already testified by the MathML project[1] [3]. The Mathematical Markup Language is an instance of XML for describing mathematical expressions capturing both their presentation and content. Although the emphasis of MathML is just on mathematical expressions, (while we are also concerned with different mathematical entities such as proofs, definitions, theorems, sections, theories, metadata and so on) MathML is an essential component of our architecture, as discussed in section 6 (see also [8]).

Let us finally remark that the broad goal of the project goes far beyond the trivial suggestion to adopt XML as a neutral specification language for the "compiled" versions

[1] We joined the MathML Working Group of the World Wide Web Consortium in October 1999.

of the libraries, or even the observation that in this way we could take advantage of a lot of functionalities on XML-documents already offered by standard commercial tools. First of all, having a common, application independent, meta-language for mathematical proofs, similar software tools could be applied to different logical dialects, regardless of their concrete nature. This would be especially relevant for all those operations like searching, retrieving, displaying or authoring (just to mention a few of them) that are largely independent from the specific logical system. Moreover, if having a common representation layer is not the ultimate solution to all interoperability problems between different applications, it is however a first and essential step in this direction. Finally, this "standardization" process naturally leads to a substantial simplification and re-organization of the current, "monolithic" architecture of logical frameworks. All the many different and often loosely connected functionalities of these complex programs (proof checking, proof editing, proof displaying, search and consulting, program extraction, and so on) could be clearly split in more or less autonomous tasks, possibly (and hopefully!) developed by different teams, in totally different languages.

Readers already acquainted with XML can skip section 2, where we give a brief introduction to the language, and start directly with the general overview (section 3).

2 The eXtensible Markup Language

Perhaps, the best way to introduce XML in few lines is to take a look at a simple example, given below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<recipebook>
  <recipe>
    <recipetype name="sweets"/>
    <title>Apple Cake</title>
    <ingredient>apples</ingredient>
    <ingredient>meal</ingredient>
    <ingredient>sugar</ingredient>
    <step number="1">
       ...
    </step>
    <step number="2">
       ...
    </step>
  </recipe>
</recipebook>
```

XML gives a method for putting structured data in a text file. Roughly speaking, the XML specification says that a XML document is made of *tags* (words bracketed by '<' and '>'), *attributes* (of the form `name="value"`) and text. Tags are used to delimit *elements*. Elements may appear in one of the following two forms: either they are non-empty elements, as `recipe` or `ingredient` (they can contain other elements or text), or they are empty elements, as `recipetype`.

The XML specification defines a XML document to be well-formed if it meets some syntactical constraints over the use of tags and attributes. For example, non-empty elements must be perfectly balanced. For this reason, someone can think of tags of non-empty elements as labelled brackets for structuring the document.

It is remarkable that XML does not specify any predefined tag set at all. Rather, it lets the user specify his own grammar by means of a Document Type Definition (DTD), a document which defines the allowed tags, the related attributes and which is the legal content for each element. The XML specification just defines the validity of a XML document with

respect to a given DTD. This is why XML is a *meta-language* that can be instantiated to a potentially infinite set of languages, each with its own DTD.

For example, the document above is valid with respect to the following DTD:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT recipebook (recipe)+ >
<!ELEMENT recipe (recipetype, title, ingredient+, step+) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT ingredient (#PCDATA) >
<!ELEMENT step (#PCDATA) >
<!ATTLIST step number CDATA #REQUIRED >
<!ELEMENT recipetype EMPTY >
<!ATTLIST recipetype name CDATA #REQUIRED >
```

We can note that a `recipebook` tag can contain one or more `recipe` elements, and that each `recipe` can contain exactly one `recipetype` followed by exactly one `title`, a positive number of `ingredients` and at least one `step` element. `title` is an example of element containing text only (`PCDATA`). `name` and `number` are attributes of `step` and `recipetype` tags respectively. The keyword `REQUIRED` states that an attribute is mandatory, i.e. it cannot be omitted when using its related tag.

References to Documents Documents and resources in general must have a name in order to be accessible over the Web. This is accomplished via the use of URIs (Universal Resource Identifiers) as defined in [10]. A generic URI is made of a formatted (structured) string of characters, without any intended meaning. URLs (Uniform Resource Locators) are a particular kind of URI specifically designed to name resources accessed by means of a given protocol (for example, HTML documents are accessed via the HTTP protocol).

As an example, let us suppose that the DTD above is stored in a document whose URI is `bookstore:/books/recipebook.dtd`; we can associate this DTD to a XML recipe book adding a special prologue:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE recipebook SYSTEM "bookstore:/books/recipebook.dtd"!>
<recipebook>
...
</recipebook>
```

Since URIs are designed to be arbitrarily extensible, standard browsers and processing tools can only be required to handle URLs, while URIs are meant to be processed by specific applications aware of them.

3 The HELM project

The overall architecture of our project, the Hypertextual Electronic Library of Mathematics, is fully described in Figure 1.

Once XML has been chosen as the standard encoding format[2] for the library, we must face the problem of recovering the already codified mathematical knowledge. Hence, we need new modules implementing exporting functionalities toward the XML representation for all the available tools for proof reasoning. Currently, we have just written such a module only for the Coq proof assistant [9]. In the near future, we expect that similar exporting functionalities

[2] A standard *format*, not a standard *language*!. In other words, the standardisation we are pursuing is not at the *logical* level, but at the *technological* one.

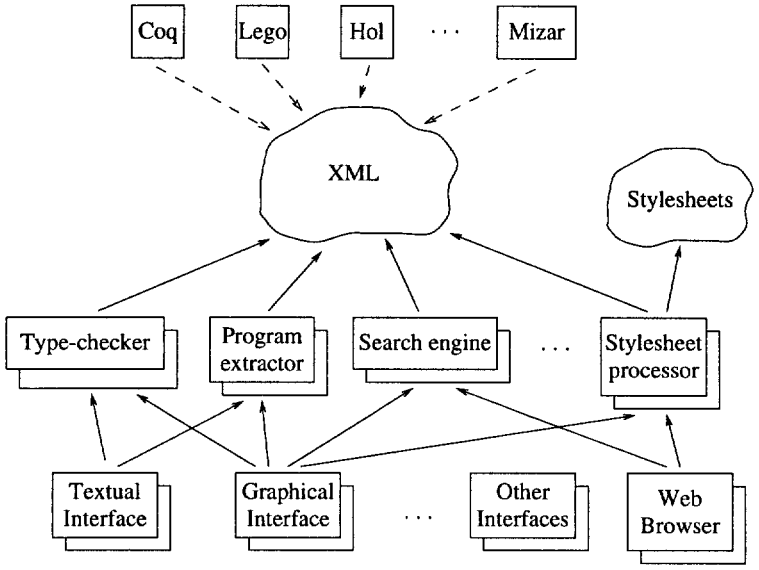

Figure 1: Architecture of the HELM project.

will be provided by the developers of the other logical systems. We will describe the exporting issues in section 4.

To exploit and augment the library, we need several tools to provide all the functionalities given by the current "monolithic" proof assistants, such as type-checking, proof searching, program verification, code extraction and so on. Moreover, we can use the available well-developed and extensible tools for processing, retrieval and rendering of XML-encoded information. In particular, to render the library information, we advocate the use of stylesheets, which are a standard way of associating a presentation to the content of XML files. This allows the user to introduce new mathematical notations by simply writing a new stylesheet. In section 5 we shall briefly discuss our implementation of a type-checking tool, while in section 6 stylesheets are addressed in details.

The user will interact with the library through several interfaces that integrate the different tools to provide an homogeneous view of the functionalities. We are developing two interfaces, described in section 7.

Because of the nature of the library, we have also provided a model of distribution, which is discussed in section 8.

4 Exporting from Coq

Coq [9] is one of the most developed proof-assistant, based on a very rich logical framework called the Calculus of (Co)Inductive Constructions (CIC). The great number of functionalities (proof editing, proof checking, proof searching, proof extraction, program verification) have made the system very big and complex. In order to work on the information encoded in such a system, the only practical way is that of writing a new module that extends it, gaining access to its internal representation.

Finding the right information inside the system itself is not a trivial task: first of all, information is encoded inside the data structures of Coq and data structures change from one version of the system to another; secondly, the information searched is often not directly available. For example, when a file is loaded, its path is completely forgotten, even if this

information could be necessary thereafter (e.g. we need it for exporting). Due to these difficulties, Coq has proven itself a challenging test bench for exporting information to XML.

To do the exporting, we have written a module in which we have implemented a set of top-level commands that, given the name of one or more CIC objects (variables, constants, types or axiom definitions), generate the corresponding XML file. The choice of writing a module without modifying the system itself seemed to be the best one, but we had to cope with the problem of information not directly available at this level. For example, the fact that the file pathnames are not present has forced us to structure the exported files into directories during a second phase. In this further phase, when moving files, we have also to modify all the URIs in all the files to reflect the changes of their position. This solution and those to similar problems are not acceptable because they add too much complexity without being necessary: we hope that the exporting functionality will be integrated within the system itself.

To design the module, the first difficulty has been the identification of which information should be exported and what should be its structure. We have chosen not to export:

Parsing and pretty-printing rules Parsing rules should depend only on the proof engine. To be able to use other proof engines different from Coq we need not to rely on Coq's own rules. Similarly, pretty-printing rules should depend only on the users choice and the type of available browser.

Tactics-related information These, too, are proof engine dependent. Moreover, we do not think that the tactics used to do a proof are really meaningful to understand the proof itself (surely, they are not the real informative content). In fact, the level of tactics and the level at which a proof should be understood are not the same: what some simple tactics do (as "Auto" that automatically search a proof) is not at all obvious. Moreover, the sequence of tactics used is clearly reflected in the lambda-term of the proof; hence it is possible to add as an attribute to a subterm the name and parameters of the tactic used to generate it.

Redundant information added by Coq to the terms of CIC Coq adds in several places a lot of information to CIC terms in order to speed up the type-checking. For example, during the type-checking of an inductive definition, Coq records which are the recursive parameters of its inductive constructors; this information is then used during the type-checking of fix functions to ensure their termination. This is an example of a clearly redundant information, useless for browsing purposes and that could be useless also for other type-checkers. We have then decided to discard it accordingly to a **principle of minimalism**: *no redundant information should be exported.* If the principle were not followed, every time we use an XML file we would have to add checks to verify the consistency of the redundant information.

Sometimes Coq also adds some non-redundant rendering information, for example when the user asks the system to infer a type and does not want to view the inferred type thereafter. This information will eventually be exported, even if this is not implemented yet.

The remaining, interesting information could be structured into three different levels that we have clearly separated in the XML representation. The first one is the *level of terms.* Terms (or expressions) could never appear alone, but only as part of an object definition. In Coq the terms are CIC lambda-expressions, i.e. variables (encoded as DeBrujin indexes), lambda-abstractions and applications, product types and sorts, augmented with a system of inductive types in the spirit of the ones of Martin-Löf, comprising (co)inductive types and constructors, case analysis operators and inductive and coinductive function definitions. The whole level is extremely dependent from the logical framework.

The second level, that uses the previous one to encode both bodies and types, is the one of *objects*. Every object is put into a different file. The files are structured into directories that corresponds to sections in Coq, i.e. delimiters of the scope of a variable. Sections are also used in Coq to structure a large theory into subtheories. In HELM, the former usage is retained, while theories are described in another way (see the third level).

Constants (definitions/theorems/axioms) Constant objects of Coq are used to represent definitions, theorems and also axioms. Definitions and theorems are syntactically identical and have a body and a type. The only difference is semantical: theorems are usually opaque (only their type is used in CIC terms) because of proof-irrelevance, while definitions are transparent (their body is substituted in their occurrences in other terms during type-checking). Axioms, instead, are constants with a type but without a body. We choose, during extraction, to discriminate only axioms from definitions and theorems, that become indistinguishable once extracted. We leave to the next level (that of theories) the responsibility of "marking" non-axiom constants as theorems, definitions, lemmas, facts, ...

Variables Variables have only a type and not a body. A variable behaves like an axiom inside the section where it is defined and as a parameter when referring to another object of that section. Hence, sections are used to delimit the scope of variables.

(Co)Inductive Definitions In Coq, blocks of mutual (co)inductive definitions can also be defined. Each definition inside such a block has a name, a type (called arity in Coq) and a possibly empty list of constructors. Each constructor has a name and a type. A simple example is the inductive type of natural numbers whose name is nat, whose type is Set and whose constructors are O of type nat and S of type (nat $\rightarrow$ nat).

Blocks, as constants, could depend on variables; the list of variables (parameters) on which all the definitions in the block depend is also exported from Coq.

Proof in progress We choose also to export unterminated proofs. As terminated theorems, an unterminated proof has a name, a type and a body; moreover, it has also a list of conjectures on which the body depends. Each conjecture has a type but not a body: to end the proof you must provide a body for each conjecture.

An example of an object file describing a constant (a theorem) can be found in appendix A. Another one is the following where you can see the definition of the inductive type of natural numbers:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE InductiveDefinition SYSTEM "http://localhost:8081/getdtd?url=cic.dtd">
<InductiveDefinition noParams="0" params="">
 <InductiveType name="nat" inductive="true">
  <arity><SORT value="Set"/></arity>
   <Constructor name="O">
    <REL value="1" binder="nat"/>
   </Constructor>
  <Constructor name="S">
   <PROD>
    <source><REL value="1" binder="nat"/>
    </source>
    <target><REL value="2" binder="nat"/>
    </target>
   </PROD>
  </Constructor>
 </InductiveType>
</InductiveDefinition>
```

Tags of the term level (`PROD`, `REL`, `SORT`) have all the letters capitalized. Tags of the object level (`InductiveDefinition`, `InductiveType`, `Constructor`) have only the initials capitalized. The remaining tags (`arity`, `source`, `target`) are only "syntactic sugar". The meaning of each tag is clearly understandable to people acquainted with CIC.

The last level is the *level of theories* which is completely independent from the particular logical framework. In our idea, a theory is a (structured) mathematical document containing objects taken almost freely from different sections. Writing a new theory should consist in developing new objects and assembling these new objects and older ones into the mathematical document. It is during the creation of a theory that objects must also be assigned the particular, semantical, meaning used to classify them, for example into lemmas, conjectures, corollaries, etc. Obviously, each theory, that is exported to a different XML file, does not include the objects directly, but refers to them via their URIs.

Theory files have also sections delimiting the scope of variable declarations: to include (a reference to) a definition D depending on a variable V when both reside in a section (directory) R, you must open, inside the theory file, a section referring to R and put inside it the references to V and R. A very small example of a theory file will clarify the above statement:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Theory SYSTEM "http://localhost:8081/getdtd?url=maththeory.dtd">
<Theory uri="cic:/coq/INIT/Logic">
 <!-- Require Export Datatypes -->
 <DEFINITION uri="True.ind"/>
 <DEFINITION uri="False.ind"/>
 <DEFINITION uri="not.con"/>
 <SECTION uri="Conjunction">
  <DEFINITION uri="and.ind"/>
  <VARIABLE uri="A.var"/>
  <VARIABLE uri="B.var"/>
  <THEOREM id="id1" uri="proj1.con"/>
  <THEOREM id="id2" uri="proj2.con"/>
 </SECTION>
 <SECTION uri="Disjunction">
  <DEFINITION uri="or.ind"/>
 </SECTION>
</Theory>
```

All the URIs, but that of `Theory`, are relative URIs; so, the absolute URI of `id1` is "`cic:/coq/INIT/Logic/Conjunction/id1`". In the example you can also see the usage of sections to bound the scope of variables: the scope of `A` and `B` is the section `Conjunction`.

It is important to note that at the theory level, sections are not used to structure the document into, for instance, chapters, paragraphs and so on; many kind of (XML) markup languages have just been developed to do so. Accordingly to the spirit of XML, our theory markup will be freely and modularly intermixed with other kinds of markup, such as XHTML[3]; so, our language will play for mathematical theories the same role of MathML for mathematical expressions and SVG[4] for vectorial graphics. The added value of using the theory level (instead of directly embedding the object markup) is that, while enriching the semantics of the objects of the previous level, it could also be used to enforce some constraints as, for example, on the scope of variables or on the links between theorems and lemmas.

[3] http://www.w3.org/TR/xhtml1

[4] http://www.w3.org/TR/SVG

The module is full working and has been used to export the whole library provided with the Coq System, yielding about 64 Mb of XML (2 Mb after compression). All the obtained XML documents are valid with respect to the DTDs developed for the three levels; we cannot show the DTDs here because of lack of space.

5 Type-Checker

In order to verify that all the needed information was exported from Coq, we have developed a stand-alone type-checker for CIC objects, similar to the Coq one, but fairly simpler and smaller thanks to its independence from the proof engine. The type-checker is now almost finished, lacking only the management of universes. It is the first example of a tool working directly on the XML encoding.

With respect to other type-checkers (as the one of Coq), it is fairly standard but for the peculiar management of the environment: usually, the type-checkers are used to check whole theories, i.e. sequence of definitions or proofs. Each time a definition is checked, it is added to the environment and then it is used in subsequent type-checkings. So, every theorem is always checked with the same, statically defined environment[5]. Our type-checker, instead, is also used to check single objects in an environment that could not yet have the definitions required (e.g. the empty environment). In this case, the environment (a cache, actually) is built "on-demand" during the type-checking of the object: every time a reference to another object not present in the environment is found, the type-checking is interrupted, processing the new object first. Checks are introduced in order to avoid cycles in the definitions, corresponding to an inconsistent environment. In order to make the user understand the strange behaviors described in note 5, that could in theory appear more often in our model, we advocate a way for the user to ask the system what is the inferred universe level of each type.

6 XSL Transformations and MathML

XSLT [7] is a language for transforming XML documents into other XML documents. In particular, a stylesheet is a set of rules expressed in XSLT to transform the tree representing a XML document (the Document Object Model, DOM [1]) into a result tree. When a pattern is matched against elements in the source tree, the corresponding template is instantiated to create part of the result tree. In this way the source tree can be filtered and reordered, and arbitrary structure can be added. A pattern is an expression of XPath [6] language, that allows to match elements according to their values, structure and position in the source tree.

XSLT is primarily aimed to associate a *style* to a XML document, generating formatted documents suitable for rendering purposes. Once XML is chosen as the data description language to encode the mathematical information, it is quite natural to use stylesheets as the standard mechanism to automatically generate the associated notation from a XML mathematical document.

In the same way MathML can naturally be chosen as the target formatting language for mathematics. MathML [3] is an instance of XML for describing the notation of a mathematical expression, capturing both its structure and content. MathML has, roughly, two categories of markup elements: the presentation markup, which can be used to describe the layout structure of mathematical notation, and the content markup, whose aim is to provide an explicit encoding of the *underlying* mathematical structure of an expression.

[5]This is almost true: a user is free to load two theories in any order, changing in this way the environment used during the second type-checking. Sometimes this can lead to strange behaviors, i.e. two theories that are correct if loaded alone could not be if loaded together. This phenomenon is due to the type-inference of the universe level and in particular to the creation of cycles between the universe level constraints.

Although the target of MathML is the encoding of expressions (so it cannot describe mathematical objects and documents), the use of MathML presentation markup as a privileged rendering format is clearly justified by the fact of providing a standard way to enable mathematical expressions to be served, received and processed on the world wide web. Moreover, its presentation part has been already implemented by several applications.

Also, the choice of using MathML content markup as an intermediate representation between the logic-dependent representation of the mathematical information and its presentation is justified by several reasons:

- Even if the content markup is restricted to the encoding of a particular set of formulas (the ones used until the first two years of college in the United States), it is essentially extensible and flexible[6].
- Passing through this semi-formal representation will improve the modularity of the overall architecture: many specific logical dialects can be mapped into the same intermediate language (or into suitable extensions of it). Moreover several stylesheets can be written for this single intermediate language to generate specific rendering formats.
- The characteristic of portability of MathML content markup can be exploited for instance when cutting and pasting terms from an application to another.
- This content level simplifies the structure of a CIC term: there is no more syntactic sugar, but only "pure expressions".
- We can capture the semantics of well-known terms, as for example the disjunction, marking them with the corresponding content elements (e.g. `or`).

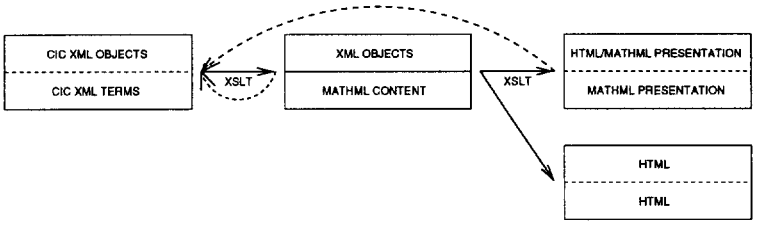

Figure 2: Transformations on the first two levels of CIC XML files: the backward arrows represent links from the content and presentation files to the corresponding CIC XML files.

As you can see in Figure 2, there are two phases of stylesheets application: the first generates the content representation from the CIC XML one; the second generates from this intermediate representation two (and eventually several other) possible kinds of output format, either the MathML presentation markup or the HTML markup. We can immediately note that MathML content can only describe CIC terms (expressions) and so we have had to add a second language to describe CIC objects in the intermediate step. Every obtained document is valid with respect to the corresponding DTD developed for it; in particular, we use the DTD recommended by the last MathML specification.

This is an example of content markup[7]:

[6]The most important element for extensibility purposes is `csymbol`, defined for constructing a symbol whose semantics is not part of the core content elements provided by MathML, but defined externally.

[7]This fragment belongs the example of content file in Appendix A.

```
<m:apply>
 <m:csymbol>app</m:csymbol>
 <m:ci definitionURL="cic:/coq/INIT/Logic/Conjunction/and_ind.con">and_ind</m:ci>
 <m:ci>A</m:ci>
 <m:ci>B</m:ci>
 <m:ci>A</m:ci>
 <m:lambda>
  <m:bvar><m:ci>H0</m:ci><m:type><m:ci>A</m:ci></m:type></m:bvar>
  <m:lambda>
   <m:bvar><m:ci>H1</m:ci><m:type><m:ci>B</m:ci></m:type></m:bvar>
   <m:ci>H0</m:ci>
  </m:lambda>
 </m:lambda>
 <m:ci>H</m:ci>
</m:apply>
```

During the realization of the content stylesheets, we have had to face and solve several problems connected to the MathML specification:

- Objects in general, cannot be considered as terms. Hence, we have added to the MathML content markup a XML level to describe the level of objects (the mixing will be possible using the W3C namespaces [4]). Anyway, this language is still dependent on the particular logical framework.
- We need to semantically describe in content markup not only entities defined in CIC, as the conjunction, but also the primitive CIC operators, as the application or the lambda abstraction. When encoding the entities, MathML markup could be really exploited. To preserve their formal semantics, we keep pointers to the XML files of their CIC definitions. With regard to most CIC operators, there are no specific MathML content markup elements. To solve this problem we use the `csymbol` element. In particular, we have chosen to introduce also a `csymbol` for the CIC application (see the example above) instead of using the content element `apply`. The MathML application is the general way of building up a mathematical expression, and so it is different from the CIC application. We want to make a clean semantical distinction between the logical application between two terms, and the "application" of some operators (like the sin function) to its arguments.

As you can see in Figure 2, we produce MathML content and presentation in two distinct steps. The only way to combine and link together content and presentation in compliance to the MathML specification consists of using the `semantics` element. This content element is quite ambiguous, a kind of "bridge" between content and presentation; moreover, it is currently ignored by all the browsers supporting MathML. For us, a natural improvement should consist of having content and the associated presentations in different files, one for the content expression and one for each presentation. Then we need to relate a presentation expression and subexpressions to the respective content expression and subexpressions. This can be achieved in a standard way using the machinery of XLink [5].

The above solution can also be exploited for the implementation of the links of Figure 2 for linking the content and presentation markup to the corresponding source CIC XML terms. In this way the user can browse the MathML presentation and also modify it: the changes will have effect on the corresponding CIC XML file.

An example of MathML presentation markup generated after the second phase is[8]:

[8]This fragment belongs the the example of content file in Appendix A.

```
<m:mrow>
 <m:mo stretchy="false">(</m:mo>
 <m:mi>and_ind</m:mi>
 <m:mphantom><m:mtext>_</m:mtext></m:mphantom>
 <m:mi>A</m:mi>
 <m:mphantom><m:mtext>_</m:mtext></m:mphantom>
 <m:mi>B</m:mi>
 <m:mphantom><m:mtext>_</m:mtext></m:mphantom>
 <m:mi>A</m:mi><m:mphantom><m:mtext>_</m:mtext></m:mphantom>
 <m:mrow>
  <m:mo color="Red">&lambda;</m:mo>
  <m:mi>H0</m:mi>
  <m:mo>:</m:mo>
  <m:mi>A</m:mi>
  <m:mo>.</m:mo>
  <m:mrow>
   <m:mo color="Red">&lambda;</m:mo>
   <m:mi>H1</m:mi>
   <m:mo>:</m:mo>
   <m:mi>B</m:mi>
   <m:mo>.</m:mo>
   <m:mi>H0</m:mi>
  </m:mrow>
 </m:mrow>
 <m:mphantom><m:mtext>_</m:mtext></m:mphantom>
 <m:mi>H</m:mi>
 <m:mo stretchy="false">)</m:mo>
</m:mrow>
```

To generate the presentation markup from the corresponding content markup, we use, among others, a stylesheet, compliant with the last specification of MathML, written by Igor Rodionov[9]. This stylesheet, written in collaboration with the MathML Working Group, transforms MathML presentation markup in MathML content one. Here, we want to stress that the possibility to re-use work done by other people is an essential aspect of the our general methodology of work.

We have had to solve several problems regarding the presentation output:

- We have had to associate an output to every object and to every `csymbol` defined in the content phase.
- We have modified the MathML stylesheet to implement the policy of line-breaking and alignment for long terms: our choice consists of using tables made of multiple rows. The `mtable` element is specifically designed to arrange expressions in a bi-dimensional layout and in particular it provides a set of related elements and attributes to achieve proper alignment and line-breaking. Our policy consists of breaking expressions only in specific points and only when the row length exceeds a threshold (that could make the reading difficult), generating tables for alignment purposes.

MathML is not the only format exploited: another presentation format is HTML, due to the wide-spreading of browsers for it and its native hypertextual nature. Thanks to the modular architecture (see Figure 2), many others could be added too.

[9]Computer Science Department of the University of Western Ontario, London, Canada.

We will exploit the same modular architecture of the object level at the level of theories. At this level we can use the same presentation formats of the previous levels; on the contrary, there is no standard language for describing theories at the content level. So we will develop a new (XML) language that will be largely independent from the specific foundational dialect and could aspire to become a standard in the same way MathML is for mathematical expressions.

7 Interfaces to HELM

Two of the main goals of the project are the easiness in augmenting and browsing the library:

1. Every user with a small amount of http or FTP space should be able to publish a document.
2. Every user with a common browser should be able to browse the library.

To fulfill these aims, we must face the actual state of technology:

1. Currently, almost all of the Internet users have a web space, but usually without being allowed to run any kind of program on the server, even simple CGIs. So no intelligence can be put on the publisher side.
2. The browser technology is rapidly evolving in such a way that we can expect in a few time to have browsers able to understand MathML and, probably, even to apply XSLT stylesheets. At the same time, though, if we require the browser to be standard, then we have to put the intelligence on the other side, i.e. on the server.

Therefore, where can we put the intelligence? A first answer is the creation of *presentation sites* able to get documents from distribution sites, process them (e.g. applying stylesheets) and return them to the users in the user requested format. We have been able to create presentation sites based on Cocoon[10], a XML server-based web-publishing framework. In a future work ([8]) we will describe in details how we have done this.

Though this solution is perfect for browsing and doing simple elaborations, it gives the user too strict interaction possibilities, which are required for more complex tasks (as the creation of new theories, for example). Hence, more advanced interfaces with such capabilities are required. These interfaces must be run on the user's machine and should, at least, be able to provide all the processing functionalities of the presentation servers, including XSLT stylesheets application. At the same time, they should also overcome the limitations of standard browsers through the addition of new interaction possibilities.

Since our preferential browsing language will be MathML, our interface should at least be able to render its presentation markup. Unfortunately, there are no satisfactory implementations available yet. Moreover, we need also to interact with the MathML rendered files, for example for editing. Not only forms of interaction with this kind of markup have never been proposed before, but we also need to reflect the changes on the source files of the XSLT rendering transformations. This has lead us to the development of a brand new engine with rendering and editing capabilities for documents embedding MathML presentation markup. This engine is designed to be stand-alone and will be made freely available as a Gtk[11] widget.

We have just integrated our widget, the type-checker and an external XSLT processor into a minimal interface that we are going to extend with editing functionalities.

[10] http://xml.apache.org/cocoon
[11] http://www.gtk.org

8 The model of distribution

Mathematical documents have some peculiar properties. First of all a mathematical document should be immutable: the correctness of a document A that refers to a document B can be guaranteed only if B does not change. Notwithstanding this, new versions of a mathematical document could be released (for example if a conjecture is actually proved). Secondly, a user cannot be forced to retain a copy of his document forever, even if other documents refer to it. So, it should be possible for everyone to make a copy of a document and also distribute it. When more than a copy is available, the user could be able to download it from a particular server (for example, from the nearest one). This implies that documents could not be referred to via URLs, but only with logical names in the form of URIs. A particular naming policy should then be adopted to prevent users to publish different documents under the same URI.

To fulfill these requirements, we have adopted almost the same model of distribution of the Debian packaging system APT[12] that has similar requirements (a package could not be modified, but only updated, it is available on different servers, could be downloaded from the user preferred server).

Every document is identified by an URI. "`cic:/coq/INIT/Datatypes/nat.ind`" is an example of such an URI that references an inductive definition (".ind") in the subsection `Datatypes` of the subsection `INIT` of the section `coq`.
Similarly, the URI "`theories:/coq/INIT/Datatypes.theory`" refers to the mathematical theory named `Datatypes` located in the subdirectory `INIT` of the directory `coq`.

In order to translate the URI to a valid URL, a particular tool, named *getter*, is needed. It takes in input an ordered list of servers and an URI and returns the URL of the document on the first server in the list providing it. In order to know which documents a server provides, each server publishes a list of the URIs of its documents with the respective URLs.

During the processing of an XSLT stylesheet, the processor must be able to open some documents, i.e. it should be able to ask the getter to resolve the URIs it needs. There is no standard way to tell the XSLT processor how to resolve URIs or concatenate external programs to do this. So, we have implemented the getter as an HTTP proxy-daemon reachable through a known URL that takes the URI as a CGI parameter, downloads the document using the resolved URL and returns it. An example of the syntax we are currently using to contact the getter is "`http://phd.cs.unibo.it:8081/get?uri="cic:/coq/INIT/Datatypes/nat.ind"`"

If the getter resides on the user machine, then the downloaded document could be cached for improving performances. Once cached, it could also be added to the list of documents the server has. In such a way, often referred to or simply interesting documents fast widespread over the net, downloading times are reduced and the author can freely get rid of his copy of the document if he needs it no more.

This architecture imposes no constraints on the naming policy: up to now we have not chosen or implemented one yet. To face the issue, one possibility can be the choice of having a centralized naming authority, even if other more distributed scenarios will surely be considered.

9 Further Developments

We are soon going to develop:

[12] http://www.debian.org

Tools for indexing and retrieval of mathematical documents, based on meta-data specified in the Resource Description Framework (RDF)[13]. RDF uses XML to define a foundation for processing meta-data, complements XML and provides interoperability between applications that exchange machine-understandable information on the web.

Tools for the (re)annotation of mathematical objects and terms: the intuitive meaning of these entities is usually lost in their description in a logical framework. Even their automatically extracted presentations in a natural language are often unsatisfactory, being quite different from the typical presentation in a book. We believe that a feasible solution is giving the user the possibility of enriching terms with annotations given in an informal, still structured language.

10 Conclusions

In this paper we have presented the current status of the HELM project, whose aim is to exploit the potentiality of XML technology for the creation and maintenance of an electronic, distributed, hypertextual library of formal mathematical knowledge.

Our ultimate goal is the extension of the library to other logical frameworks and systems. This will also be an important test bench for the whole architecture.

Another fundamental improvement would be the development of new modular proof engines, supporting step-by-step informal annotations on proofs in natural language (see [8] for a deeper discussion of this topic).

References

[1] Document Object Model (DOM) Level 2 Specification. Version 1.0, W3C Candidate Recommendation, 10 May 2000.
`http://www.w3.org/TR/2000/CR-DOM-Level-2-20000510/`

[2] Extensible Markup Language (XML) Specification. Version 1.0. W3C Recommendation, 10 February 1998.
`http://www.w3.org/TR/REC-xml`

[3] Mathematical Markup Language (MathML) 2.0 W3C Working Draft, 28 March 2000.
`http://www.w3.org/TR/2000/WD-MathML2-20000328/`.

[4] Namespaces in XML, W3C Recommendation, 14 January 1999.
`http://www.w3.org/TR/REC-xml-names/`

[5] XML Linking Language (XLink), W3C Working Draft (last call), 21 February 2000.
`http://www.w3.org/TR/xlink/`

[6] XML Path Language (XPath) Version 1.0, W3C Recommendation, 16 November 1999.
`http://www.w3.org/TR/xpath`

[7] XSL Transformations (XSLT). Version 1.0, W3C Recommendation, 16 November 1999.
`http://www.w3.org/TR/xslt`.

[8] Asperti, A., Padovani, L., Sacerdoti Coen, C., Schena, I., "XML, Stylesheets and the re-mathematization of Formal Content", Department of Computer Science, Bologna, Italy, May 2000.

[13] http://www.w3.org/TR/REC-rdf-syntax.

[9] B. Barras et al., "The Coq Proof Assistant Reference Manual, version 6.3.1", http://pauillac.inria.fr/coq/

[10] Berners-Lee, T., "Universal Resource Identifiers in WWW", RFC 1630, CERN, June 1994.

11 APPENDIX A

Cic Xml file:

```
<!DOCTYPE HTML SYSTEM "http://localhost:8081/getdtd?url=cic.dtd">
<Definition name="proj1" params="0: A B">
  <body>
   <LAMBDA>
     <source>
       <APPLY>
         <MUTIND notype="0" uri="cic:/coq/INIT/Logic/Conjunction/and.ind"/>
         <VAR reluri="0,A"/>
         <VAR reluri="0,B"/>
       </APPLY>
     </source>
     <target binder="H">
       <APPLY>
         <CONST uri="cic:/coq/INIT/Logic/Conjunction/and_ind.con"/>
         <VAR reluri="0,A"/>
         <VAR reluri="0,B"/>
         <VAR reluri="0,A"/>
         <LAMBDA>
           <source><VAR reluri="0,A"/></source>
           <target binder="HO">
             <LAMBDA>
               <source><VAR reluri="0,B"/></source>
               <target binder="H1"><REL binder="HO" value="2"/></target>
             </LAMBDA>
           </target>
         </LAMBDA>
         <REL binder="H" value="1"/>
       </APPLY>
     </target>
   </LAMBDA>
  </body>
  <type>
   <PROD>
     <source>
       <APPLY>
         <MUTIND notype="0" uri="cic:/coq/INIT/Logic/Conjunction/and.ind"/>
         <VAR reluri="0,A"/>
         <VAR reluri="0,B"/>
       </APPLY>
     </source>
     <target><VAR reluri="0,A"/></target>
   </PROD>
  </type>
</Definition>
```

Corresponding content file:

```
<?xml version="1.0" encoding="UTF-8"?>
<Definition name="proj1" xmlns:m="http://www.w3.org/1998/Math/MathML">
  <Params>0: A B</Params>
  <body>
   <m:math>
    <m:lambda>
     <m:bvar>
      <m:ci>H</m:ci>
      <m:type>
       <m:apply>
        <m:and definitionURL="cic:/coq/INIT/Logic/Conjunction/and.ind"/>
        <m:ci>A</m:ci>
        <m:ci>B</m:ci>
       </m:apply>
      </m:type>
     </m:bvar>
     <m:apply>
      <m:csymbol>app</m:csymbol>
      <m:ci definitionURL="cic:/coq/INIT/Logic/Conjunction/and_ind.con">and_ind</m:ci>
      <m:ci>A</m:ci>
      <m:ci>B</m:ci>
      <m:ci>A</m:ci>
      <m:lambda>
       <m:bvar><m:ci>H0</m:ci><m:type><m:ci>A</m:ci></m:type></m:bvar>
       <m:lambda>
        <m:bvar><m:ci>H1</m:ci><m:type><m:ci>B</m:ci></m:type></m:bvar>
        <m:ci>H0</m:ci>
       </m:lambda>
      </m:lambda>
      <m:ci>H</m:ci>
     </m:apply>
    </m:lambda>
   </m:math>
  </body>
  <type>
   <m:math>
    <m:apply>
     <m:csymbol>arrow</m:csymbol>
     <m:apply>
      <m:and definitionURL="cic:/coq/INIT/Logic/Conjunction/and.ind"/>
      <m:ci>A</m:ci>
      <m:ci>B</m:ci>
     </m:apply>
     <m:ci>A</m:ci>
    </m:apply>
   </m:math>
  </type>
</Definition>
```

Corresponding presentation file:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN" "http://www.w3.org/TR/REC-html40/strict.dtd">
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
  <m:mtable align="baseline 1" columnalign="left" equalrows="false">
   <m:mtr>
    <m:mtd><m:mrow><m:mtext>DEFINITION proj1() OF TYPE</m:mtext></m:mrow></m:mtd>
   </m:mtr>
   <m:mtr>
    <m:mtd>
     <m:mrow>
      <m:mphantom><m:mtext>__</m:mtext></m:mphantom>
      <m:semantics>
       <m:mrow>
        <m:mo stretchy="false">(</m:mo>
        <m:mrow>
         <m:mi>A</m:mi>
         <m:mo><mchar name="wedge"></mchar></m:mo>
         <m:mi>B</m:mi>
        </m:mrow>
        <m:mo color="Blue">&rarr;</m:mo>
        <m:mi>A</m:mi>
        <m:mo stretchy="false">)</m:mo>
       </m:mrow>
       <m:annotation-xml encoding="MathML">
        <m:apply>
         <m:csymbol>arrow</m:csymbol>
         <m:apply>
          <m:and definitionurl="cic:/coq/INIT/Logic/Conjunction/and.ind"/>
          <m:ci>A</m:ci>
          <m:ci>B</m:ci>
         </m:apply>
         <m:ci>A</m:ci>
        </m:apply>
       </m:annotation-xml>
      </m:semantics>
     </m:mrow>
    </m:mtd>
   </m:mtr>
   <m:mtr><m:mtd><m:mrow><m:mtext>AS</m:mtext></m:mrow></m:mtd></m:mtr>
   <m:mtr>
    <m:mtd>
     <m:mrow>
      <m:mphantom><m:mtext>__</m:mtext></m:mphantom>
      <m:semantics>
       <m:mrow>
        <m:mo color="Red">&lambda;</m:mo>
        <m:mi>H</m:mi>
        <m:mo>:</m:mo>
        <m:mrow>
         <m:mi>A</m:mi>
         <m:mo><mchar name="wedge"></mchar></m:mo>
         <m:mi>B</m:mi>
        </m:mrow>
        <m:mo>.</m:mo>
        <m:mrow>
         <m:mo stretchy="false">(</m:mo>
         <m:mi>and_ind</m:mi>
         <m:mphantom><m:mtext>_</m:mtext></m:mphantom>
         <m:mi>A</m:mi>
         <m:mphantom><m:mtext>_</m:mtext></m:mphantom>
         <m:mi>B</m:mi>
         <m:mphantom><m:mtext>_</m:mtext></m:mphantom>
         <m:mi>A</m:mi><m:mphantom><m:mtext>_</m:mtext></m:mphantom>
         <m:mrow>
          <m:mo color="Red">&lambda;</m:mo>
          <m:mi>H0</m:mi>
          <m:mo>:</m:mo>
          <m:mi>A</m:mi>
          <m:mo>.</m:mo>
           <m:mrow>
            <m:mo color="Red">&lambda;</m:mo>
            <m:mi>H1</m:mi>
            <m:mo>:</m:mo>
            <m:mi>B</m:mi>
            <m:mo>.</m:mo>
            <m:mi>H0</m:mi>
```

```
            </m:mrow>
           </m:mrow>
           <m:mphantom><m:mtext>_</m:mtext></m:mphantom>
           <m:mi>H</m:mi>
           <m:mo stretchy="false">)</m:mo>
          </m:mrow>
         </m:mrow>
         <m:annotation-xml encoding="MathML">
          <m:lambda>
           <m:bvar>
            <m:ci>H</m:ci>
            <m:type>
             <m:apply>
              <m:and definitionurl="cic:/coq/INIT/Logic/Conjunction/and.ind"/>
              <m:ci>A</m:ci><m:ci>B</m:ci>
             </m:apply>
            </m:type>
           </m:bvar>
           <m:apply>
            <m:csymbol>app</m:csymbol>
            <m:ci definitionurl="cic:/coq/INIT/Logic/Conjunction/and_ind.con">and_ind</m:ci>
            <m:ci>A</m:ci>
            <m:ci>B</m:ci>
            <m:ci>A</m:ci>
            <m:lambda>
             <m:bvar><m:ci>H0</m:ci><m:type><m:ci>A</m:ci></m:type></m:bvar>
             <m:lambda>
              <m:bvar><m:ci>H1</m:ci><m:type><m:ci>B</m:ci></m:type></m:bvar>
              <m:ci>H0</m:ci>
             </m:lambda>
            </m:lambda>
            <m:ci>H</m:ci>
           </m:apply>
          </m:lambda>
         </m:annotation-xml>
        </m:semantics>
       </m:mrow>
      </m:mtd>
     </m:mtr>
    </m:mtable>
</m:math>
```

Protocols for Interactive e-Proof

David Aspinall

May 2000

LFCS, University of Edinburgh, U.K.
http://www.dcs.ed.ac.uk/home/da

Abstract. I motivate and describe work-in-progress on *Proof General Kit* , an evolution of the Proof General project. Proof General Kit introduces a new architecture for Proof General. Instead of the present monolithic implementation in Emacs Lisp, where each proof assistant is connected through a separate customization, Proof General will become a collection of pluggable communicating components using a standard protocol. The Kit is centred around this *protocol for interactive electronic proof*, dubbed PGIP. The design of PGIP has been influenced by the generic mechanisms that Proof General presently uses to communicate with various interactive proof assistants. PGIP is a rationalization of these mechanisms, and will be proposed as an open standard for future proof assistants to support. Messages in PGIP are small XML documents. PGIP comes with an associated markup language, PGML, for representing structural aspects of concrete syntax.

See the home page at `http://www.lfcs.ac.uk/proofgen` for more about Proof General.

1 Introduction

Proof General is a generic interface for interactive proof assistants, based on Emacs [1, 2].

Proof General was first built to address the needs of a particular class of users. Many proof assistants still have a primitive command line interface. Even when sophisticated GUI alternatives are available, it is observed that expert users often *prefer* the command line interface and work more effectively with it. This may be for several reasons: perhaps because the GUIs are poorly engineered, perhaps because they are overly restrictive or do not scale to large developments, or simply because current experts do not want to change their working practices and spend effort on learning an interface. (For more opinions, see [11]).

Proof General targetted expert users, by fitting quite closely with their existing models of interaction, but making those interactions more effective and convenient by adding short-cuts and centralising development around the end product — the *proof script*. Recent improvements such as buttons, menus, and symbol fonts have made Proof General more accessible for novice users too. It now provides a middle ground in interface technology, largely text-based rather than graphical, but with sophisticated features like *script management* and *proof by pointing* (both inspired by the CtCoq GUI [5]), implemented in a lightweight and efficient way. Proof General's implementation of script management is particularly impressive, allowing an integration with the file-handling system of the proof assistant.

The strategy of targetting experts as well as novices has been a success. Proof General is now widely used in teaching as well as research, and in industry as well as academia. Perhaps the greatest and most unique aspect of the success of Proof General is its *genericity*. It exploits the deep similarities between systems by hiding some of their superficial differences. Just as a web browser presents a similar interface to different protocols — http, ftp, or file, so Proof General presents a similar interface to different proof assistants. This genericity is no empty claim or briefly tested design goal; the system is already in common use for LEGO, Coq, and Isabelle. Support for HOL98 has recently been added.

2 A New Architecture

Although it has been a success, the present Proof General has some drawbacks and scope for improvement. For example, from the user's point of view:

1. The Emacs-centric nature puts off some people who don't use or don't like Emacs.
2. Not enough system-specific options are provided, for example, to invoke common tactics or commands in a system. The interface degenerates to offering a command-line prompt for these cases.
3. There is no allowance for interoperability with other tools, and only restricted facility for internet-based distributed development.
4. While Proof General presents a similar appearance for different proof assistants, ultimately the user must be familiar with most commands of the underlying proof assistant. Further abstraction should be possible.

From the implementors point of view, Proof General is not as easy to connect to proof assistants as we might like:

1. Customization and extension is exclusively in Emacs Lisp. (Recently it has been drastically simplified, but it still involves setting some tricky Emacs regular expressions).
2. The implementor has too much choice, in a sense: each proof assistant is free to use its own communication triggers and markup schemes.[1] This makes it complicated to configure Proof General and inhibits communication with other tools.
3. There is some duplication of information in the interface and the proof assistant, and some information is hard-coded into Emacs which would better belong in the proof assistant.

To address these drawbacks, we propose a new architecture for Proof General. First, we recognize that not everybody shares a love for Emacs, and that Emacs Lisp is not the best language for implementing some advanced aspects of the interface. To allow more generality, we envisage a collection of alternative pluggable components, implemented in various languages, which can serve as user interface elements, proof assistants, and eventually other proof tools. Second, to make this possible, we realize that the new architecture requires a common underlying interface language and interaction protocol.

To take things further, we want to deepen the abstraction that Proof General provides, so that the logic and proof language, as well as the interaction model, become generic. Ultimately the choice of logic and its syntax should fall under Proof General's organization, and we envisage implementing advanced aspects of logical frameworks: a logic morphism from the core logic of each supported proof assistant into Proof General's general form of the logic. Thus Proof General would truly live up to its name: a generic system for doing proofs, with the underlying proof engine being almost arbitrary.[2]

But that ultimate goal remains for the future; we prefer to build up towards it in stages. Right now, we are working on the first stage.

3 Proof General Kit

The Kit introduces a new architecture for Proof General. Instead of a monolithic implementation inside Emacs, Proof General will become a collection of communicating components. In the spirit of the present system, we want to use carefully designed lightweight protocols, which are easily supported by a range of present and future proof engines. The protocols and components will be developed in stages. The first stage is to develop the *PGIP protocol* and *PGML markup language*.

PGIP is the protocol that we propose for communicating the progress of an interactive proof; it is based on examining and clarifying the mechanisms currently implemented. We give a short description in Section 4 below.

PGML is the markup language used inside PGIP to annotate terms and formulae for display

[1] This is because we tried to make Proof General work with existing systems without requiring their modification. But this was only partially successful: for robustness and features like proof by pointing, it is necessary to modify the output routines of proof assistants anyway.

[2] This aspect is dubbed *Logic General*. Similar ideas appear elsewhere, e.g. in OMEGA [4].

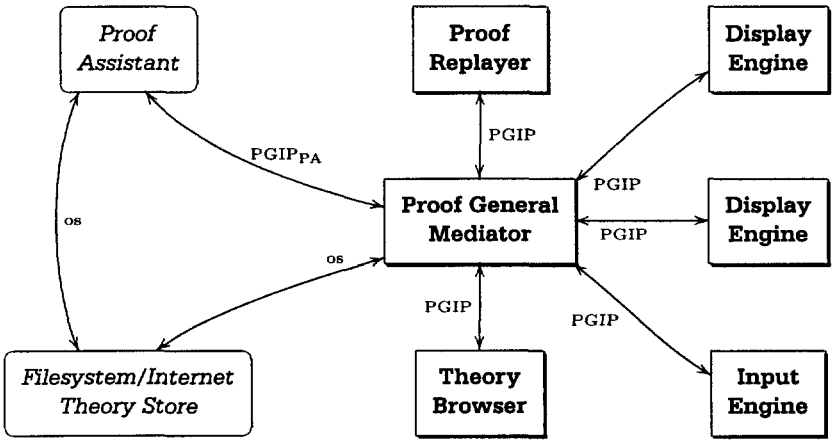

Fig. 1. Simplified Architectecture of Proof General Kit

by the interface. We give a short description in Section 5 below.

System architecture An overview of the architecture of the Proof General Kit is shown in Figure 1. The proof assistant and filesystem are outwith the Kit; the other components are part of the system.

Notice the central role of the *Mediator* here: it interfaces the various other components to the proof assistant and filesystem, whereas the other components have no direct access to either. Only the mediator and the proof assistant use operating system calls to access the file system (or internet) theory store. This separation allows the Mediator to organize the synchronization messages needed for script management. At the start of the project, the Mediator is even more important, because it provides extra interfacing mechanisms to connect to the proof assistant.[3] This is because we begin from proof assistants which do not natively or fully handle PGIP, explaining the $\mathrm{PGIP}_{\mathrm{PA}}$ label on the arrow.

The remaining components are examples only. In a particular session we might connect several *Display Engines* (perhaps running remotely, or displaying different aspects). Only one *Input Engine* is allowed to be *active for scripting* at a time, which means that it owns the current proof development.[4] The *Proof Replayer* is a restricted kind of input engine which only allows replay of previous proofs, perhaps over the internet. The *Theory Browser* is a component which allows browsing of the theory store and/or the theories which are loaded into the running proof assistant.

4 PGIP

PGIP is a protocol for conducting interactive electronic proof. It has other features to enable the Kit architecture, for example, various querying operations on the filesystem or running proof assistant.

Ultimately we hope that PGIP will be adopted as a standard by proof assistant implementors;[5] the benefit to them will be that they can connect off-the-shelf interfaces, and with less effort than it takes to connect to Proof General currently.

Essentials PGIP makes certain assumptions about the way proof is conducted with a proof assistant. The essential aspect is that proof proceeds in steps, and that the proof assistant can be "fed" a step at a time, without needing to see the whole proof. So we assume that the proof assistant itself maintains some state representing

[3] In the Proof General Kit white paper [3], the Mediator is decomposed into several subcomponents.

[4] This is because we assume that the proof assistant is single threaded and maintains only a single proof state. Later on, we may treat proof assistants as a resource, and allow the Kit to connect to several at once; for the time being we will work within the current model of just a single proof assistant instance.

[5] This stands some chance, because it is being designed in conjunction with some of those very people!

the proof development. A proof using PGIP has these parts:

- Proof begins by issuing a target *goal*
- Proof proceeds by successive *proof steps*
- To reverse the effect of a proof step, the proof assistant has a command to *undo*.
- A proof is closed, aborted, or abandoned by issuing a *save-goal*, *quit-goal*, or *forget-goal*, respectively.

We consider the goal, save-goal, and forget-goal operations to be proof steps themselves, whereas undo and quit-goal are not. Together, the proof steps form a language for writing *proof scripts*. There is much freedom on exactly what a proof script is: for example, it may well be a *declarative* rather than a *procedural* description of a proof. It matters little to Proof General whether scripts are forwards or backwards proofs, or whether they use tactics or some more readable description of proof steps. The essential aspect is that they have a textual, linear representation.[6]

Proof scripts may have additional elements outside proofs, for example, to make definitions or assumptions. (If definitions and assumptions are allowed inside proofs, we consider them as proof steps and they must be undoable). Moreover, proof scripts may have additional structure both inside and outside proofs, to allow *sections* or *block structure* which exposes some structure of the underlying proof tree and theory.

As well as controlling the progress of the interactive proof, PGIP is responsible for certain initialization and book-keeping tasks which require communication between the proof assistant and the rest of Proof General. It includes messages to configure (proof assistant specific) user-level menus and preferences; messages to display status or error dialogues, and messages to retrieve theory files.

For specifics of the PGIP format including an XML DTD, see [3].

Sending PGIP messages A PGIP message is sent in a "packet", which is an XML document with root <pgip>. We haven't yet decided on a particular transmission mechanism for packets. To begin with we are experimenting using Unix pipes and pseudo ttys; later we may use sockets and/or other mechanisms (probably http for internet connectivity; perhaps a simple CORBA interface for desktop connectivity). PGIP messages are themselves structured in XML, and may contain embedded PGML.

5 PGML

PGML is an XML-based document structure for transmitting marked-up *concrete* syntax output from proof assistants.

Most present proof assistants have their own favourite concrete syntax and syntax defining mechanisms, but little or no access to abstract syntax. So we assume that the output is already in concrete syntax; the reason to markup is to allow decoration of aspects of the syntax (e.g. special constants or variables), and also to allow proof-by-pointing style interactions via subterm position annotations. We want to use a logical markup mechanism to make these things possible, so that different renderers can display information in different ways, allowing variations in fonts, colours, special characters, hidden annotations, etc.

There are already emerging standards for XML-based document formats designed for displaying mathematics (MathML, [14]), and transferring mathematical content between applications (OpenMath, [12]). Later on, we hope to use these languages with PGML. At the moment, it is more feasible to implement our own simple markup scheme since both MathML and OpenMath go further into the *abstract* structure of terms than we need, or than we can easily accommodate in a generic way.

Term structure annotations are an effective half-way house, and can be included by those proof assistants capable of generating them. Already for systems like Isabelle it is a highly non-trivial task to relate the abstract syntax to

[6] Many interfaces like to develop proofs graphically based on trees or graphs, since their authors argue that this is the true nature of a proof. Yet the abstract syntax of a program is also tree-based, and text-based linear description and development of programs is hardly obsolete. Proof General takes the linear text to be primary, but may later offer adjuncts for browsing or developing proofs graphically, perhaps using *dependency analysis* [13] for exposing the structure of proof scripts.

```
<pgml>
<statedisplay name="Level 3" id="avocado.dcs.ed.ac.uk/951858790/12480/101">
    Level 3 (2 subgoals)
    <br/>
    <statepart kind="initialgoal">
        A & B --> B & A
    </statepart>
    <br/>
    <statepart kind="subgoal" name="1"> 1.
        <statepart kind="asms"> [| A; B |] </statepart>
        ==>
        <statepart kind="body"> A </statepart>
    </statepart>
    <br/>
    <statepart kind="subgoal" name="2"> 2.
        <statepart kind="asms"> [| A; B |] </statepart>
        ==>
        <statepart kind="body"> B </statepart>
    </statepart>
    <br/>
</statedisplay>
</pgml>
```

Fig. 2. Example of outer structure PGML markup for Isabelle

the concrete, and generate annotations. But we believe that more proof assistants will provide structured output in the future.

PGML documents Here is a proof state display in Isabelle:

```
Level 3 (2 subgoals)
A & B --> B & A
 1. [| A; B |] ==> B
 2. [| A; B |] ==> A
```

In Figure 2, the skeleton of a PGML marked-up version of the same display is shown. Isabelle has a rich structure on the subgoal display, always including the overall goal and a list of local assumptions for each subgoal. The PGML markup makes this structure apparent within the concrete syntax. It is simple to implement this markup scheme in the output routines of Isabelle. And with the marked-up version of the subgoal display, it will become easy for display engines to hide or reveal parts of the display, quickly navigate through large displays, etc.

PGML can go a bit further into the structure of terms than shown in Figure 2, in particular using these elements:

- <term pos="*p*" >*term text*</term>
- <action kind="*kd*">*action text*</action>
- <atom kind="*kd*">*name*</atom>
- <sym name="*nm*" alt="*SYM*">

Terms The purpose of <term> is to add enough structure to annotate subterm position information, for mouse-sensitivity and proof-by-pointing actions. The *pos* annotation achieves this; typically it is a position indicator in the abstract syntax tree corresponding to the surounding term. The position is not displayed or interpreted by the display engine, but should be recoverable by mouse pointing, so that it can be sent back to the proof engine.

Actions Terms may have *action* texts attached to them. Zero or more <action> texts appear immediately after an openning <term>. The idea behind actions is that they allow additional hidden annotations which may be revealed to the user (e.g. typing information, proof hints), or used to generate commands to the proof assistant to allow fine-grained proof-by-pointing style interactions.

Atoms Atoms in terms can be decorated by the display engine to indicate their logical status and to attach extra information. The logical status is suggested by the *kind* name, which may be used by the display engine to generate different colours or balloon popups. It is appropriate to

use different kinds when the atom is named from a distinct namespace (e.g. if we want to help the user distinguish between a variable "x" and a constant "x").

Symbols Proof General at present can use the XEmacs package *X Symbol* [15] to display a variety of mathematical symbols, Greek letters, etc, which are not part of ASCII. PGML will allow these extra characters to appear via the <sym> tag.

For specifics of PGML and a DTD, see [3].

6 Design and Engineering

Careful design and good engineering are both important to achieve a usable system. And even when building research prototypes, we believe that software quality deserves serious attention.

One design question for a framework like PGIP is where to make the split between interface and proof engine. One might expect the part of the system which records the state of the proof development to be closely coupled to the interface, where it is always manipulated; perhaps with a functional API for constructing proofs in the engine. But experience with Proof General shows, perhaps surprisingly, that the stateful development mechanisms already implemented in diverse provers are compatible enough to have a generic interface at a higher level, sending proof commands. This is more efficient, since the data structures associated with proof development (e.g. list of current subgoals) are rather closely coupled with the proof engine. This explains the form of PGIP described in Section 4.

An engineering factor that seems important in the success of Proof General is the incremental way it has been constructed, by *successive generalization*.[7] It began as "LEGO mode", an interface for one system. Then support for Coq was added, generalizing to "Proof Mode". In the last couple of years, support for Isabelle was added, generalizing further, and giving birth to "Proof General." Each stage of generalization involved a mix of modifying and re-engineering the basic core of the system, and adding new features, all the while carefully maintaining support for previous proof assistants. This process is synergistic: supporting a new system typically *improves* support for the other systems. Features which are useful or easy to add for the new system get added back to previous systems, as innovations there; a direct and novel cross fertilization method.

We hope that this engineering approach can help the future success of Proof General Kit. That is why we start from a restricted and "bottom up" plan, rather than a "grand design" promising the world. It means we can refine and extend a series of useful working prototypes, rather than struggle for a long time on something unwieldy.

7 Conclusions

We have described and motivated ideas for the Proof General Kit. At the moment, we are at an early stage, conducting experiments and applying for funding. More details of what was described here appear in the white paper [3], which is being discussed on the Proof General developer's mailing list. Just now, we are considering the viability of PGIP for handling interaction methods other than text-based script processing; in particular, we hope to connect to a generic direct manipulation interface developed at the Universities of Bremen and Freiburg [10].

I welcome feedback and discussion on this paper and [3]. For information about the Proof General project or to try Emacs Proof General visit `http://www.lfcs.ac.uk/proofgen`.

Related work Interoperability is in the fashion, and several folk in the theorem proving community are taking it seriously. Other diverse projects include OMEGA [4], OMRS [8], Prosper [6] and HELM [7]. Compared to these, Proof General Kit has a rather different focus, concentrating on protocols for use during interactive proof, rather than communicating or storing semantic content. That said, we hope to consider semantic content as an extension to PGIP in the future, perhaps linking up to one of the other projects. Another difference of the projects mentioned is that most grew from other communities, which favour different proof assistants to the ones we are working with. But seems that the interoperability drives are making some links between the theorem proving cliques, at last, and we hope that Proof General Kit will add to this.

[7] This is reminiscent of a *product line architecture* in software engineering.

Acknowledgements I'm grateful to the Proof General developer team, and several others, who have expressed interest in the Kit project. In particular, for discussions on the content of [3] I'm grateful to Paul Callaghan, Pierre Courtieu, Christoph Lüth, and Markus Wenzel.

Proof General Credits Thomas Kleymann began LEGO mode, also worked on by Dilip Sequeira, and had the idea of making it generic. Healfdene Goguen added Coq support. Kleymann managed development until Aspinall took over; Aspinall added support for Isabelle and helped Proof General to its first widely distributed releases in late 1998. Since then Aspinall has developed Proof General together with helpers from the camps of the supported proof assistants (a deliberate policy), including M. Wenzel, D. von Oheimb, P. Callaghan, P. Loiseleur, and P. Courtieu. We are grateful to users and testers for their feedback and suggestions.

References

1. D. Aspinall, H. Goguen, T. Kleymann, and D. Sequeira. Proof General. System documentation, see `http://www.dcs.ed.ac.uk/home/proofgen`, 2000.
2. David Aspinall. Proof General: A generic tool for proof development. In Graf and Schwartzbach [9], pages 38–42.
3. David Aspinall. Proof General Kit. White paper. See http://zermelo.dcs.ed.ac.uk/home/da/drafts/, 2000.
4. C. Benzmüller et al. ΩMega: Towards a mathematical assistant. In William McCune, editor, *14th International Conference on Automated Deduction — CADE-14*, volume 1249 of *LNAI*. Springer, 1997.
5. Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(7):161–194, February 1998.
6. Louise A. Dennis et al. The PROSPER toolkit. In Graf and Schwartzbach [9].
7. Andrea Asperti et al. Helm – an hypertextual electronic library of mathematics. Home page at `http://www.cs.unibo.it/~asperti/HELM/home.html`, 2000.
8. Fausto Giunchiglia, Paolo Pecchiari, and Carolyn Talcott. Reasoning theories: Towards an architecture for open mechanized reasoning systems. In F. Baader and K.U. Schulz, editors, *"Frontiers of Combining Systems - First International Workshop" (FroCoS'96)*, Kluwer's Applied Logic Series (APLS), pages 157–174, 1996.
9. Susanne Graf and Michael Schwartzbach, editors. *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1785. Springer-Verlag, 2000.
10. C. Lüth and B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 9(2), 1999.
11. N. Merriam and M. Harrison. What is wrong with GUIs for theorem provers? In Y. Bertot, editor, *User Interfaces for Theorem Provers UITP'97*. INRIA Sophia Antipolis. Electronic proceedings at `http://www.inria.fr/croap/events/uitp97-papers.html`, 1997.
12. The OpenMath Society. `http://www.nag.co.uk/projects/openmath/omsoc/`.
13. Olivier Pons, Yves Bertot, and Laurence Rideau. Notions of dependency in proof assistants. In *Proc. User Interfaces for Theorem Provers, UITP'98*, 1998.
14. Mathematical markup language (MathML). W3C Recommendation, 1999. `http://www.w3.org/TR/REC-MathML/`.
15. Christoph Wedler. Emacs package "X-Symbol". `http://www.fmi.uni-passau.de/~wedler/x-symbol/`.

Reasoning about Order Errors in Interaction

Paul Curzon and Ann Blandford

School of Computing Science, Middlesex University, London, UK
{p.curzon,a.blandford}@mdx.ac.uk

Abstract. Reliability of an interactive system depends on users as well as the device implementation. User errors can result in catastrophic system failure. However, work from the field of cognitive science shows that systems can be designed so as to completely eliminate whole classes of user errors. This means that user errors should also fall within the remit of verification methods. In this paper we demonstrate how the HOL theorem prover [7] can be used to detect and prove the absence of the family of errors known as order errors. This is done by taking account of the goals and knowledge of users. We provide an explicit generic user model which embodies theory from the cognitive sciences about the way people are known to act. The user model describes action based on user communication goals. These are goals that a user adopts based on their knowledge of the task they must perform to achieve their goals. We use a simple example of a vending machine to demonstrate the approach. We prove that a user does achieve their goal for a particular design of machine. In doing so we demonstrate that communication goal based errors cannot occur.

1 Introduction

People commonly make mistakes when interacting with computer-based devices. Whilst some errors cannot always be prevented, such as those caused by users behaving randomly and maliciously, there are whole classes of error that have distinct cognitive causes and are predictable [13]. Furthermore, changes to the design of systems can eliminate such errors [9, 3]. Formal verification aims to either detect system errors or show their absence. If user errors can be eliminated using appropriate design then their detection ought to be within the remit of formal verification methodologies. However, formal verification is generally done in a machine-centered way. A consequence is that avoidable user errors are not detected or corrected as part of the verification process.

In this paper, we describe a verification methodology for detecting user errors. Our approach is to formally model *rational* users as part of the system being verified. We focus here on errors resulting from a mismatch between the device design and the order a user expects to supply information or objects. This extends earlier work concerning a different class of errors known as post-completion errors [5]. Our verification approach is capable of detecting both classes of error simultaneously. The verification described has been fully machine-checked using interactive proof with the HOL theorem prover [7].

We define a generic user model that can be instantiated for different machines. This user model describes *rational* user behaviour based on established results from cognitive science [11]. The verification approach therefore detects *rational* user errors. This differs from similar approaches in which the environment of the machine is specified to provide the input required (treating users as logical as opposed to rational agents). With such an approach user errors are treated as never occurring. Our approach is also different from assuming that the environment could perform any action at any time (users as "monkeys at keyboards"). That would amount to saying that whatever the user's goal and whatever actions they perform, they will eventually achieve the goal. This is not appropriate for interactive systems as the functionality of such a system would need to be trivial for it to be considered "correct". Instead, our approach recognises that users are important but do not act randomly. The user is described in terms of the things they wish to achieve; the actions they may perform in order to achieve those goals and in terms of the device-independent knowledge they have about the task. We are interested in eliminating errors from systems that occur when users act in this way as such errors are liable to be persistent.

2 Formal User Modelling

There are, broadly speaking, two main approaches to formal reasoning about the usability of systems. One approach is to focus on formal specification of the user interface; Campos and Harrison [4] review several techniques that take this approach. However, such techniques do not support reasoning about errors. The alternative, which we take in this work, is based on formal user modelling. This involves generating a formal specification of the user in conjunction with one of the computer system, in order to support reasoning about their conjoint behaviour. It should be noted that a formal specification of the user is a description of the way the user is, rather than one of the way the user should be, since users cannot be designed in the way that computer systems can [1]. Examples of formal user modelling include the work of Duke *et al* [6], Butterworth *et al* [2], Moher and Dirda [10] and Paterno' and Mezzanotte [12]. Each of these approaches takes a distinctive focus. Duke *et al* [6] use a mathematical notation to express constraints on the channels and resources within an interactive system; this makes their 'syndetic modelling' technique particularly well suited to reasoning about multi-modal interaction (such as that combining the use of speech and gesture). Butterworth *et al* [2] use Lamport's [8] TLA to reason about behaviour traces and reachability conditions within an interaction; this approach describes behaviour at an abstract level that does not support re-use of the user model from one computer system to another, so while it can support reasoning about errors, each model has to be individually hand-crafted. Moher and Dirda [10] use Petri net modelling to reason about users' mental models and their changing expectations over the course of an interaction; this approach supports reasoning about learning to use a new computer system – which, in turn may be an important source of errors, but focuses on changes in

user belief states rather than proof of desirable properties. Finally, Paterno' and Mezzanotte [12] use LOTOS and ACTL to specify intended user behaviours and hence reason about interactive behaviour; their approach corresponds closely to that which is done in state space exploration verification, but because their user model describes how the user is intended to behave, rather than how users might actually behave, it does not support reasoning about errors.

3 Classes of User Error

A common form of error made by humans in a wide variety of situations is the *Post-completion Error* [3]. Examples include taking the cash but leaving a bank card in an Automatic Teller Machine and leaving the original on the platen and walking away with the copies when using a photocopier. Most ATM machines have been redesigned to force users to remove their cards before cash is delivered to avoid this problem, but the phenomenon persists in many other environments. There are of course other situations where a user does not complete all the subtasks associated with a main goal. For example, if a fire alarm went off whilst a person was using a photocopier, they might not take their original. However, such an error would not be a post-completion error in our sense as it would have a different underlying cause. A design that eliminated post-completion errors would not necessarily guarantee the user would not make the same surface level "mistake" for other reasons.

Post-completion errors are interesting because they are not predictable (i.e. they do not occur in every interaction) but they are persistent. They are not related to missing knowledge so cannot be eliminated by increased user training. They can, however, be eliminated with careful system design. Curzon and Blandford [5] illustrate the use of HOL to reason about such errors by considering alternative device designs. Here we develop that approach by extending the generic user model to identify a new class of errors with a distinctive cognitive cause. In particular, we look at errors that occur when there is a mismatch between the design of a device and the knowledge that a user has about the task (independent of the particular device used to complete that task). A user will often know of specific information that must be communicated to any such device for the task to be completed. They may not know precisely how or when the information must be imparted to a particular machine. They thus maintain a mental list of *communication goals*: information that they must communicate at some point. If the order that the information must be imparted to the machine is not known, or the user's mental model of the task suggests a different order then order errors can result. The user attempts to fulfill their goals in an order different to that required by the machine.

Order errors can also arise due not to information that must be communicated, but to objects that must be supplied: an ATM card, coins, etc. For example, with a vending machine, the user will know they must make a selection of chocolate and that they must supply money, but for a given machine they will not necessarily know the order. If they know exactly what they want but not

the price, they may be inclined to press the selection first (some machines would display the price at this point). Alternatively, they may have the coin in their hand and so insert it first before working out exactly which buttons to press to make their selection.

Each of the above classes of errors have distinct cognitive causes. We provide a verification approach that detects such errors in a structured way. Whilst we cannot eliminate all user errors, we *can* eliminate whole classes of error that have such specific cognitive causes.

4 Proving Usability

A proof of usability, in the sense that particular classes of errors cannot occur, involves proving a theorem of the form

```
⊢ ∀(ustate: ustate_type) (mstate: mstate_type).
  MACHINE_USER ustate mstate ∧ MACHINE_SPEC s mstate ⊃
      MACHINE_USABLE ustate mstate
```

MACHINE_SPEC is a traditional machine specification: a relation over an internal state `s` and inputs and outputs `mstate`. The latter represents the interface between the device and its users. States and signals are represented by *history functions*: functions from time to the value at that time. MACHINE_USER is also a specification of a component of the system: the user of the device. It describes the actions a *rational* user might take based on their knowledge and goals. It is a relation on an internal user state `ustate` and the inputs and outputs of the device. We will look at in more detail in the next section. The conjunction of these two relations provides a specification of the system as a whole: both device and user. The conclusion we prove about this combined device is not phrased in terms of what the device can do, or explicit properties of it. Instead it is a specification of whether the user achieves their goal in interacting with the device.

Note that the above usability theorem is of the basic form

$$\vdash \mathit{implementation} \supset \mathit{specification}$$

It can thus easily be combined with a traditional correctness theorem that an implementation of the machine meets the given specification [5].

In one sense the user model fills a similar role to an environment machine in traditional model-checking based verification. It provides inputs to the device being verified. The difference is not in the fact that such an environment is provided but in the *kind* of environment provided. Rather than providing values based on what the machine specification requires, or on other devices connected to the device, it is modelling the way people behave based on results from cognitive science. The user of course may not be providing all the inputs to the device. Thus unlike with an environment machine, the combined user-device system is not necessarily closed. We are treating the user as part of the system under verification, rather than just a test rig to verify the system. The kind of errors we

are looking for are those that result from the user component of the system, but which can be eliminated by modifying the device component of that system.

5 A Generic User Model

We could adopt the approach of providing a separate user model for each distinct device that we wish to verify. However, this approach could lead us back into a machine-centered specification approach, specifying users that do exactly what the specific device requires of them. Moreover, we wish to detect classes of user error that are widespread and not just confined to specific devices. It therefore makes sense to provide once-and-for-all a generic user model that incorporates cognitive science theory about the way people behave in general. Such a generic model can then be targeted to specific machines, simply by providing details about the machine state, the user's knowledge of the task and their goals. Higher-order logic provides an elegant framework in which to specify such a generic model. It allows functions and relations providing details of a specific interaction to be an argument to the generic user model. For example to support reasoning about post-completion errors the user model contains general machinery regarding termination conditions. This is defined in terms of a variable representing an *interaction invariant*: a relation indicating the part of the state that should be restored for the task to be considered completed. The user model takes a specific instance of such an invariant as an argument.

The generic user model is given as a relation `USER` over the user and machine states as described above. In addition however, it takes a series of other arguments representing the details of the specific machine. To instantiate the user model for a given machine, we must provide:

- concrete types for the state of the machine and of the user,
- a list of actions a user might take (inserting coins, pushing buttons, etc),
- a history function to record the communication goals of users of the device at each instant in time,
- a list giving the user's initial communication goals,
- a list pairing device outputs with user inputs, indicating relationships where the output is intended to make the user react by taking the action resulting in the input (for example, a light might be located next to a button, with the light being on indicating the button should be pressed),
- history functions recording the possessions of the user and how they change over time as the interaction progresses,
- a history function recording when the user terminates the interaction (by leaving the device) together with that signal's position in the list of possible actions,
- the goal users of the device are trying to achieve, and
- a history function describing an interaction invariant that should hold both at the start and end of the interaction.

We will discuss each of these in more detail below as we describe the definition of the user model.

The core of the user model is a series of temporally guarded statements about possible actions a rational user might take. For example, one disjunct is associated with each of the paired lights and actions, reflecting the fact that a rational user could react to a light coming on by pressing the associated button. This is specified by:

```
⊢ LIGHT user_actions light action ppos (mstate:'m) t =
      (light mstate t = T) ∧
      NEXT user_actions (action mstate) ppos t
```

This states that if the light is on at a time t then the *next* action performed by the user from the list of possible actions user_actions will be the one paired with the light (action). Since this is just one clause of a list of disjuncts, it is not guaranteed that the action will be taken. A recursive definition LIGHTS forms a disjunct of all the pairs in the given list of lights and actions. Note that mstate (similarly ustate) has a polymorphic type in this and the other definitions of this section representing the fact that we are defining a generic user model that can apply to machines and users with different states.

The relation NEXT specifies the next action to occur. To define it we first define relations LSTABLE and LF. The former is used to specify that the signals do not change in some interval. The latter then states that at the end of that interval all but one of the signals remains false.

More formally, LSTABLE is a temporal operator that states that all the history functions in the given list have a value v between the start and end time.

```
⊢ (LSTABLE [] t1 t2 v = T) ∧
  (LSTABLE (CONS a l) t1 t2 v =
           (∀t. t1 <= t ∧ t < t2 ⊃ (a t = v)) ∧
           (LSTABLE l t1 t2 v))
```

LF states that all but one of the actions in the list (that indicated by position ppos) are false at a given time. This is defined recursively on the action list.

```
⊢ (LF n [] P ppos t = T) ∧
  (LF n (CONS a l) P ppos t =
      (((n = ppos) ∨ ~(a t)) ∧ (* miss the numbered signal *)
      LF (n+1) l P ppos t))
```

Note that we can not simply use a list MEMBER function here as it would check whether the *values* in the list were equal to one being checked. We wish to identify a specific action, not the value of an action. In the absence of a syntax for user actions, we use the position in the list to identify the action.

NEXT uses the above definitions to specify that there is a time later than that given when the action identified by the position occurs (its history function is true), the other actions do not occur (their history functions are false), and for which all the actions do not occur in all the intervening time instances.

```
⊢ NEXT al P ppos t1 =
    ∃t2. t1 <= t2 ∧ (LSTABLE al t1 t2 F) ∧ (LF 0 al P ppos t2) ∧ (P t2)
```

If the temporally guarded statements that make up the user model were based only on the pairs of lights and actions as defined above, we would be specifying a *reactive* user who did exactly what was required. However, other clauses are included to reflect *rational* behaviour based on user goals and knowledge. The first such disjunct describes the fact that a rational user may terminate the interaction on achieving their goal. If this action is taken, before the user's interaction invariant is restored, a post-completion error is made.

```
⊢ COMPLETION user_actions finished finishedpos goalachieved ustate t =
    (goalachieved ustate t = T) ∧
    NEXT user_actions (finished ustate) finishedpos t
```

In this paper we are primarily concerned with errors that result from devices not taking communication goals of users into account. For more detail of verification of designs with respect to post-completion errors see [5].

As discussed earlier, a user of a device generally enters into an interaction with some knowledge about the task. Specifically they are likely to know of some of the information that must be communicated to the device, because they know the task cannot be completed, whatever the device design, unless it receives this information. They will not necessarily know the order the information must be communicated, however.

We model this using a list of actions, corresponding to the communication goals. We first extract the communication goal list from the user state for the time of interest. This allows COMMGOALS to be defined recursively on that argument.

```
⊢ COMMGOALER user_actions actions goal ustate mstate t =
    COMMGOALS user_actions (actions ustate t) goal ustate mstate t
```

This gives a list of communication goals with their position in the list of all possible actions the user could perform. We recurse on this list to produce a list of action disjuncts based on the communication goals.

```
⊢ (COMMGOALS user_actions [] goal ustate mstate t = F) ∧
  (COMMGOALS user_actions (CONS a actions) goal ustate mstate t =
      ((COMMGOALS user_actions actions goal ustate mstate t) ∨
       (COMMGOAL user_actions (FST a) (SND a) goal ustate mstate t)))
```

COMMGOAL describes a temporally guarded action similar to LIGHT and COMPLETION given earlier. A separate relation is defined for this for consistency throughout the user model: each guarded action is given by a similar definition. Provided the user's main goal has not yet been achieved, the next action they will take if this disjunct is activated (i.e. true) is the given communication goal.

```
⊢ COMMGOAL user_actions action n goal ustate mstate t =
    ~(goal ustate t) ∧
    NEXT user_actions (action mstate) n t
```

Since all the communication goals are disjuncts and all have the same guard, no ordering of them is prescribed by these definitions. The user may attempt to complete them in any order. Once a communication goal related action has been completed, it will cease to be a communication goal. We examine how this is specified below.

Each of the actions that a rational user might make when confronted with the machine are combined in a single definition GENERAL_USER_CHOICE. It contains a final default disjunct, ABORTION. It asserts that if none of the guards of the other disjuncts hold (and so no rational action is available) then the user will terminate the interaction without having achieved their goal.

```
⊢ GENERAL_USER_CHOICE user_actions commgoals lights_actions
              finished finishedpos goalachieved mstate ustate t =
    COMMGOALER user_actions commgoals goalachieved ustate mstate t ∨
    LIGHTS user_actions lights_actions 0 mstate t ∨
    COMPLETION user_actions finished finishedpos goalachieved ustate t ∨
    ABORTION user_actions finished finishedpos goalachieved commgoals
              lights_actions ustate mstate t
```

This relation describes the series of options that a user has open to them on any given cycle. There are other conditions that must apply at every instance in time, however. For example, we assume it is always the case that if the user terminates the interaction then they cannot then continue with it.

```
∀t. finished ustate t ⊃ finished ustate (t+1)
```

We similarly assume various rules about the possessions of a user. For example, we assume it is always the case that if a user gives up a possession then they have one less of that possession. These rules are encapsulated into a relation POSSESSIONS. We omit the details of this relation here.

We also assert universal properties of the communication goal list. It is not a constant over time. As the user performs the actions associated with a communication goal, that goal is discharged and so is removed from the user's internal list of things to do: it ceases to be a communication goal. This behaviour is modelled by asserting that if an action that appears on the communication goal list occurs at a time t, then that action will be removed from the communication goal list on the subsequent cycle.

```
⊢ (FILTER [] mstate t = []) ∧
  (FILTER (CONS a actions) mstate t =
    if (FST a) mstate t then (FILTER actions mstate t)
                       else (CONS a (FILTER actions mstate t)))

⊢ FILTER_HLIST mstate hlist = ∀t. hlist (t+1) = FILTER (hlist t) mstate t

⊢ FILTER_USER_HLIST ustate mstate hlist = FILTER_HLIST mstate (hlist ustate)
```

The separate relations describing universal properties are cojoined together into a single relation GENERAL_USER_UNIVERSAL.

```
⊢ GENERAL_USER_UNIVERSAL commgoals possessions finished ustate mstate =
    (∀t. finished ustate t ⊃ finished ustate (t+1)) ∧
    (POSSESSIONS possessions ustate mstate) ∧
    (FILTER_USER_HLIST ustate mstate commgoals)
```

We need two further elements to our generic user model, however. We must assert that at the start of the interaction, the user's communication goals are in fact those supplied as the initial list.

```
⊢ USER_INIT cgoals init_cgoals ustate = (cgoals ustate 0 = init_cgoals)
```

Finally we must describe the situation where the user terminates the interaction normally. We have considered the situation where a user completes their goal and leaves. However, we argued that this may lead to post-completion errors. Normal, non-erroneous termination involves leaving not just when the goal is completed, but also when any necessary house-keeping tasks have been completed. A non-device specific way of describing this is by using the notion of an interaction invariant that the user wishes to maintain. The invariant may be perturbed in the course of the interaction, but must be reinstated by the time the interaction is terminated.

If the goal is achieved and the interaction invariant satisfied, then we assume that the rational user will always terminate the interaction as the next action. If either condition is not fulfilled, the user will take some action from the set of options. This is combined with the initialisation and universal relations to give the complete generic user model.

```
⊢ USER user_actions commgoals init_commgoals lights_actions possessions
          finished finishedpos goalachieved invariant ustate mstate =
  (USER_INIT commgoals init_commgoals ustate) ∧
  (GENERAL_USER_UNIVERSAL commgoals possessions finished ustate mstate) ∧
  (∀t.
    if  ((invariant ustate t = T) ∧ (goalachieved ustate t = T))
    then  NEXT user_actions (finished ustate) finishedpos t
    else  GENERAL_USER_CHOICE user_actions commgoals lights_actions
                    finished finishedpos goalachieved mstate ustate t)
```

This user model, instantiated with the details of a specific machine, specifies aspects of a general rational user of that machine. Because all the options are modelled as guarded disjuncts, the model does not specify that users always make mistakes, just that they are capable of making mistakes of specific kinds. To verify that the modelled user always achieves their goal, the device specification must be such that the opportunities for such errors are not present. For example, if a chocolate machine design always gives out change before chocolate, the guard on the COMPLETION disjunct will only be activated when the interaction invariant has already been restored. In this way we have provided a facility which can be used to verify that whole classes of errors cannot occur with a given design.

6 Case Study: A Chocolate Machine

To demonstrate how our user model can be used to verify the absence of classes of errors we will look at a simple case study. In [5] we used an earlier, less sophisticated version of the user model to investigate the verification of simple vending machines with the potential for post-completion errors. Here we consider a similar example, but instead concentrate on communication goal related errors. The design consists of features that appear in real machines. However, it has been reduced to the simplest form with which to demonstrate our approach.

Our chocolate machine takes exact money only and it is assumed it will only take a single coin of that value. To release the chocolate a button must be pressed (this is intended as a simplified version of the selection that most machines would offer). The design of the machine could require a specific ordering: coin inserted, then button pressed, or button pressed then coin inserted. In either case order errors could result. The problem can be eliminated if either order is allowed. We verify here a machine that does allow either ordering. We will also discuss the effect of trying to verify faulty designs. We assume for the sake of simplicity that the chocolate machine always contains chocolate.

We formally specify the chocolate machine using a traditional finite state machine description (see Figure 1) within higher order logic. The specification is represented by a relation on the machine's inputs and outputs. We group these inputs and outputs into a tuple of history functions to represent the machine state. We define a new type `mstate_type` to represent this. The machine has two inputs indicating that the button has been pressed and that the coin has been inserted. It has a single output that releases chocolate. Each of the history functions is a function from time (a natural number) to booleans indicating the value of the signal at that time. We define a series of accessor functions to obtain the values of particular components of the state. For example the function `InsertCoin` extracts from a machine state the history function representing the coin slot.

We define a new enumerated type `ChocState` to represent the 4 finite state machine states (as opposed to the state representing the values input and output discussed above).

```
ChocState = RESET_STATE | COIN_STATE | CHOC_STATE | DONE_STATE
```

The RESET state is the initial state. In the DONE state the chocolate is released. The COIN state is the state in which a coin has been inserted but the button not pressed and vice versa for the CHOC state.

For each state we define a relation indicating the value on the single output in that state, together with a relation indicating the next state. These are then combined in a relation giving the full specification for that state. For a small example such as that considered here, it might be simpler to just have one definition giving the whole automaton. However such an approach would not scale: in particular the resulting specification would be much less readable.

For example when in the RESET state the machine does not release chocolate so the value of the output is false.

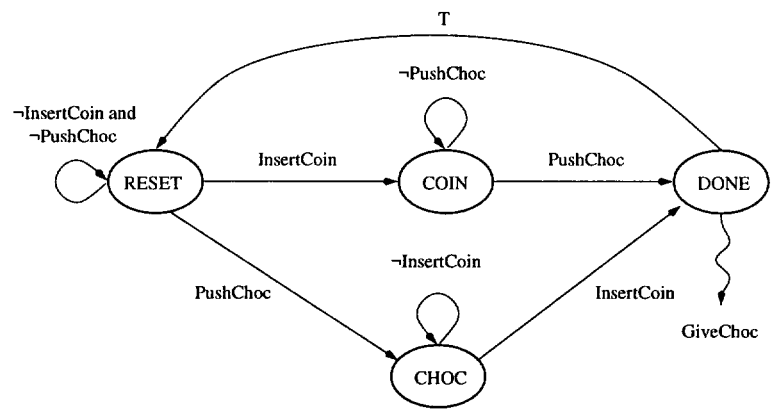

Fig. 1. Finite State Machine Specification of the Chocolate Machine

```
⊢ RESET_OUTPUTS (mstate: mstate_type) t = (GiveChoc mstate t = F)
```

We also give a relation representing the next state for each state. If a coin is entered it moves to a COIN state in the next cycle, if the button is pressed it moves to the CHOC state and otherwise it remains in the RESET state.

```
⊢ RESET_TRANSITION s mstate t =
    if       InsertCoin mstate t    then (s(t+1) = COIN_STATE)
    else if  PushChoc mstate t      then (s(t+1) = CHOC_STATE)
                                    else (s(t+1) = RESET_STATE)
```

For each state these two relations are combined in a relation that gives the whole behaviour (for example RESET_SPEC for the RESET state). A single definition then gives the full specification of the machine in terms of these definitions.

```
⊢ CHOC_MACHINE_SPEC s mstate =
    ∀t. if        (s t = RESET_STATE)    then  RESET_SPEC s mstate t
        else if   (s t = COIN_STATE)     then  COIN_SPEC s mstate t
        else if   (s t = CHOC_STATE)     then  CHOC_SPEC s mstate t
                                         else  DONE_SPEC s mstate t
```

7 Instantiating the User Model

To target the generic user model to a given machine we must provide values for all the arguments to USER except for the user state and machine state. For these we provide concrete types to instantiate the type variables given as their type.

The type of the machine state is just that used in the machine specification defined above: a tuple of history functions. For the user state we must provide a state consisting of a tuple of 6 elements. These elements are history functions that record for each time instance whether the user has chocolate, whether they

have a coin, whether they have terminated the interaction, a count of the amount of chocolate they possess, a count of the number of coins they possess, and a list of their communication goals paired with numbers giving the position of the corresponding action in the full list of actions. An accessor function for each part of the state is defined. For example, `UserCommgoals` extracts the communication goal list from the state.

The first argument we provide to `USER` is a list of all the possible user actions indicated by their history functions: the state extractor applied to the appropriate state tuple.

```
[InsertCoin mstate; PushChoc mstate; UserFinished ustate]
```

The second argument is the state extractor for the communication goals, `UserCommgoals`. We must also provide the initial communication goal list with which the user enters the interaction. In this case we assume that the user knows they must insert a coin at some point and that they must make a selection (push the chocolate button). This would be determined using a device-independent task analysis of the task of getting chocolate. We use the state extractor function to represent each communication goal. These are paired with a number giving their position in the full action list.

```
[(InsertCoin, 0); (PushChoc, 1)]
```

Note that, strictly speaking, inserting a coin is not a communication goal as it is concerned with property rather than information about a selection to be made. We intend in a later version of the user model to deal with these two kinds of knowledge separately. The main ramification for the theorem proved here is that as a communication goal no check is made in the user model as to whether the user has a coin as one of its possessions. This means the correctness theorem though not explicitly stating it says nothing about what happens if the user tries to insert a coin that they do not have.

Our particular machine provides no output to the user to indicate what must be done so an empty list is provided as the next argument for the pairings between outputs and the corresponding reactive input. A case study concerning post-completion errors where reactive pairings are provided can be found in [5].

We must also indicate the possessions of the user and how they are affected by particular actions. A relation `CHOC_POSSESSIONS` gathers this information into an appropriate form, given the history functions for the user having chocolate and coins, the machine giving chocolate, the user inserting a coin and counts of the number of coins and chocolate bars possessed.

```
CHOC_POSSESSIONS UserHasChoc GiveChoc CountChoc
                 UserHasCoin InsertCoin CountCoin
```

We specify which accessor functions to the user state indicate when the user has terminated the interaction, `UserFinished`, together with the number of its position in the list of actions (as with the communication goals). We also specify

the state accessor specifying the user's main goal in taking part in the interaction, UserHasChoc.

Finally we must provide the invariant that the user wishes to restore by the end of the interaction. For vending machines this can be based on the value of the user's possessions. After interacting with a vending machine a user does not wish the value of their total possessions to be less than they were at the start. This is described by a history predicate VALUE_INVARIANT. We omit the definition here.

The general model for the chocolate machine is specified by providing each of the arguments discussed above to the generic user model and restricting the types of the states to be the concrete types for the chocolate machine.

```
⊢ CHOC_MACHINE_USER (ustate:ustate_type) (mstate:mstate_type) =
    USER [InsertCoin mstate; PushChoc mstate; UserFinished ustate]
         UserCommgoals [(InsertCoin, 0); (PushChoc, 1)]
         ... ustate mstate
```

8 Verifying Usability

The usability correctness theorem we have proved in HOL has the following form:

```
⊢ ∀ustate mstate s.
  CHOC_MACHINE_USER ustate mstate ∧ CHOC_MACHINE_SPEC s mstate ⊃
      (s 0 = RESET_STATE) ∧ ~(UserHasChoc ustate 0)
                          ⊃ (∃t2. UserHasChoc ustate t2)
```

This is of the general form discussed earlier. The usability specification part of the theorem states that if we assume the vending machine starts in its reset state, and the user does not have chocolate but has communication goals of inserting a coin (paying money) and pushing the chocolate button (making a selection), then there will exist some time at which the user does have chocolate (i.e., has achieved their main goal).

This theorem is essentially proved using simulation by proof. An induction principle concerning the stability of a signal is used repeatedly. This essentially states that:

- if the value of some boolean signal P is stable over an interval,
- a second signal, Q, is true at the start of that interval, and
- if Q is true at some time, but P has the stable value at that time, then Q will be true at the subsequent time,
- then Q will be stable over an interval starting at the same point but extending one cycle later.

This is used to step the simulation over periods of inactivity.

In proving the usability theorem we have not proved that users using the machine will never make an error. We have, however, proved that no user will make the classes of errors with known cognitive causes specified in the user

model. In particular, we have proved that a user will not make order errors due to communication goal mismatches, provided they start with the stated communication goals. If these communication goals are identified using a device-independent task analysis then they will be consistent with the majority of users. Since such errors are both common and persistent as discussed in Section 3 the reliability of the system as a whole is consequently improved.

Consider an attempt to verify a design which requires the coin to be inserted before the button was pushed. This proof attempt would fail because the user model allows the user to do either of the communication goals first. If they pushed the button first, this action would be removed from their list of goals: they would believe the selection made. On then inserting a coin to complete their other goal, there would be no longer anything in the user model to compel them to press the button. We thus would be required to prove that they pushed the button, with no assumptions with which to do this. Of course a real user would in this case eventually work out the problem and go on to complete the interaction. However, the user error has already occurred.

9 Conclusions

We have described a formal verification methodology which detects classes of user error. In particular we have so far considered order errors based on communication goal mismatches and post-completion errors. These classes of errors are considered because they can be eliminated by appropriate design.

Our approach involves defining a generic user model which describes the behaviour of rational users. As with real users, erroneous behaviour is not specified to occur during every interaction. It is just specified as a potential behaviour. Given that potential behaviour exists, if it can be proved that the user does eventually achieve their goal, then it has been proved that the erroneous behaviour cannot manifest itself with the device under verification.

The use of a generic user model reduces the work required to produce a user model for each new device considered. More importantly, it reduces the chances that the user model is created in a device-centered way, specifying that the user behaves as expected by the designer of the device. It is based only on cognitive science theory that is generally applicable.

As alternative approach would be to write liveness properties corresponding to a list of known user errors for each system to be verified. However, to do so would require informal reasoning to determine the manifestation of the error from rational behaviour for every new device considered. For example, the order errors considered here are errors because the user does not have perfect knowledge of the design. Post completion errors are errors dependent on the user's goals. It is only by reasoning about the user's goals and knowledge that we determine the actions for which the ordering is important and determine what that ordering should be. In our approach, this reasoning is formalised and machine-checked. The general rational behaviour is specified once and the errors emerge.

The fact that a common user model is used means that the proofs for different devices are very uniform, increasing the possibilities for automation of the proof. For examples as simple as that presented here to illustrate the ideas it is likely that fully automated model checking/state-space exploration based verification tools could be used. However, when more realistic devices are considered it is likely that the additional power of an interactive theorem prover will be required. Furthermore, higher-order logic provides an elegant way in which a generic user model can be specified. It seems likely that this kind of proof would be a good application for a combined verification tool. The instantiated user model would be instantiated in HOL and exported to the automated system. Higher level details of the proof would be dealt with in HOL, with state exploration conducted in the automated tool. HOL could also be used to combine the usability correctness theorem with more traditional system verification theorems.

We used a very simple example of a chocolate machine to demonstrate the approach. We instantiated the generic user model with the details of a specific machine designed to avoid order errors. Despite the machine giving no indication of the steps required, because its design works with the communication goals of the task, it is usable. We also discussed how the proof would fail if other erroneous designs were considered. The design works because it has a permissive interface, allowing users to supply information in any order. It might be argued that such an approach could always be used. However, post-completion errors occur if the ordering of actions by the user is such that the user can complete their main goal before other required actions have been completed. Thus to avoid post-completion errors we must do the opposite of making the interface permissive. We must instead force a specific order. For example, if a machine dispensed change, it would be important that it was not dispensed before the chocolate. We investigated the verification of post-completion errors in an earlier paper [5]. There we investigated vending machines with and without post-completion errors. Our present user model has the ability to simultaneously detect order errors and post-completion errors. In future work we will investigate more complex machines and other classes of user errors. We will also look at machine designs with the potential for making multiple classes of errors. When considering a single class of error in isolation, it is relatively easy to ensure it is not present. When multiple kinds of errors are considered it is very easy to remove one kind of error, only to introduced another. This is where having a single generic user model is beneficial, since it ensures errors are not missed. It is in this situation that our verification approach will be of most use.

Acknowledgements This work is funded by EPSRC grants GR/M45221 and GR/L00391. The work was done in part whilst the first author was visiting Cambridge University Computer Laboratory.

References

1. R. Butterworth, A. Blandford, and D. Duke. Demonstrating the cognitive plausibility of interactive system specifications. Submitted to FACS journal. Available

from http://www.cs.mdx.ac.uk/puma/ as working paper WP25.
2. R.J. Butterworth, A.E. Blandford, and D.J. Duke. Using formal models to explore display based usability issues. *Journal of Visual Languages and Computing*, 10:455–479, 1999.
3. M. Byrne and S. Bovair. A working memory model of a common procedural error. *Cognitive Science*, 21(1):31–61, 1997.
4. J.C. Campos and M.D. Harrison. Formally verifying interactive systems: a review. In M. D. Harrison and J. C. Torres, editors, *Design, Specification and Verification of Interactive Systems '97*, pages 109–124. Wien : Springer, 1997.
5. Paul Curzon and Ann Blandford. Using a verification system to reason about post-completion errors. Presented at Design, Specification and Verification of Interactive Systems 2000. Available from http://www.cs.mdx.ac.uk/puma/ as working paper WP31.
6. D.J. Duke, P.J. Barnard, D.A. Duce, and J. May. Syndetic modelling. *Human-Computer Interaction*, 13(4):337–394, 1998.
7. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.
8. L. Lamport. The temproal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16:872–923, 1994.
9. W-O Lee. The effects of skills development and feedback on action slips. In Monk, Diaper, and Harrison, editors, *People and Computers VII.* Cambridge University Press, 1992.
10. T.G. Moher and V. Dirda. Revising mental models to accommodate expectation failures in human-computer dialogues. In *Design, Specification and Verification of Interactive Systems '95*, pages 76–92. Wien : Springer, 1995.
11. A. Newell. *Unified Theories of Cognition.* Harvard University Press, 1990.
12. F. Paterno' and M. Mezzanotte. Formal analysis of user and system interactions in the CERD case study. In *Proceedings of EHCI'95: IFIP Working Conference on Engineering for Human-Computer Interaction*, pages 213–226. Chapman and Hall Publisher, 1995.
13. J. Reason. *Human Error.* Cambridge University Press, 1990.

Verification of Synthesized Analog designs using a Theorem Prover and a Computer Algebra System*

Abhijit Ghosh† and Ranga Vemuri
Digital Design Environments Laboratory
Department. of ECECS, University of Cincinnati
Cincinnati, OH 45221-0030
{aghosh,ranga} @ececs.uc.edu

Abstract

This paper presents novel techniques for functional verification of synthesized analog designs. The work focuses on extraction of formal models of the implementation from circuit net-lists (Spice net-list), extraction of specification from a behavior description (VHDL-AMS) and development of a formal reasoning system to prove the relationship between the extracted *implementation* and *specification* models. Implementation model of the circuit consists of characteristic component equations and equations depicting current-voltage laws. Some of these equations can be nonlinear. Specification model of the circuit typically consists differential and algebraic equations (DAEs). In order to validate the design we have to establish an expected relationship between the two models in a formal reasoning system. Traditional theorem provers (like PVS) support linear algebra and can be used to prove such relationships in a linear domain, but they fail to handle nonlinear equations. This paper shows that by using the support of the nonlinear reasoning mechanism of a computer algebra system,CAS (like Mathematica) in a restricted manner, a theorem prover can be used to automatically verify functional properties of analog designs described over nonlinear equations.

1 Introduction

The process of hardware design is error prone. Simulation is the principal tool used to detect design errors. Several simulation tools are available to verify a design at different levels of abstraction. Simulators have also been developed for different kinds of designs (digital, analog and mixed). However, simulators suffer from two fundamental shortcomings as they may leave design errors undetected and they are compute intensive in nature. The quality of the simulation is essentially determined by the design coverage of the excitation signals and exhaustive simulation of circuits is often not feasible. Formal hardware verification methods overcome these difficulties by proving the expected functionality as a symbolic relationship which holds *forall* possible values of the symbols and are found to be extremely useful tools for detection of design errors. Most of the formal verification effort has concentrated on digital designs and have found reasonable success. Design verification of analog designs using simulation suffers from the problems discussed before and a formal verification technique would greatly aid the design process but there have been relatively negligible attempts to apply such techniques to verify analog and mixed signal designs [1], [2]. These existing techniques use decision procedures (solvers) based on linear algebra and so they use piecewise linear (PWL) and rectilinear approximations to represent structural description of

*This work is sponsored by USAF, Air Force Research Laboratory, WPAFB under contract number F33615-96-C-1911
†Currently at Cadence Designs Systems. Work done at University of Cincinnati.

analog circuits using linear algebraic equations. Approximating nonlinear analog component behavior by linear equations can lead to inconsistency and it is difficult to make any judgment about nonlinear behavior in the absence of nonlinear reasoning mechanism.

We can see that in order to prove functional properties about analog designs using nonlinear component models we need nonlinear reasoning support. There has been significant effort by researchers to integrate theorem provers (typically written over higher order logic) with CASs [3], [4], [5], [6]. Theorem provers and CASs are powerful mathematical tools and are extremely useful for their domain specific applications. Logical soundness and proof management of theorem provers find their usefulness in proving complicated properties expressed in their underlying logic. There are popular theorem provers which provide user interaction to guide the proof process as well as proof script languages to automate proof processes. CASs on the other hand are useful to establish numerical and symbolic equivalence in a variety of mathematical and engineering applications.

There are specific application domains where we are aware of the nature of symbolic expressions we are dealing with and it is possible to integrate these two tools supplementing each other such that we have a stronger reasoning system. In this paper we present analog circuit verification as such an application where we can use the combined power of theorem provers and CASs. We begin our discussion by motivating the reader to the issues involved in analog verification in section 2, where we mention that a verification tool must have model extraction, proof management and reasoning mechanisms. We illustrate our model extraction strategy for implementation and specification models in section 3. In section 4, we present the detailed comparison between the theorem prover and CAS and the issues involved in combining them. This is followed by the discussion about our proof effort and the nature of support we need from the CAS where we describe the nature of nonlinear reasoning obtained from CAS and how it aids the rewriting ability of our theorem prover. We also present the details of integrating our specific theorem prover (PVS [7]) and CAS (Mathematica [8]). Section 5 presents results of application of our technique.

2 Formal Verification of Analog Designs: Motivation and Challenges

Verification of analog designs needs a formal system in which we can represent the specification and implementation and a reasoning mechanism in the formal system which will prove the relation between them. In the following subsection we discuss issues involved in analysis of analog systems and then we will illustrate the properties of the required formal system.

2.1 Analyzing the Analog circuit

Analog components exhibit continuous time behavior often represented as an algebraic function of several parameters. Theory of analog circuit analysis is well established [9]. There are a number of well understood techniques such as nodal analysis and tableau analysis which have been automated and used widely [10]. The basic idea of these methods is as follows:

- Model a circuit in terms of primitive components such as independent voltage and current sources, controlled sources and resistive elements,
- Setup equations relating the above using Kirchoff's Voltage and Current Laws,
- Solve these equations using Cramer's rule (if the equations are linear) and use Numerical techniques like Newton Raphson (for nonlinear equations).

Active analog components have different regions of operation and correspondingly different models for them. This kind of behavior is expressed with linear/nonlinear algebraic functions and is often dependent on the frequency of signals applied. This leads to each component's behavior being abstracted into a set of models according to operating frequencies. For example, an operational amplifier has different models for different operating frequencies. Many analog devices have inherent nonlinearity in their behavior. For example, a transistor has at least three regions of operation (cutoff, linear, and saturation) distinct from one another [11], [9]. A greater difficulty of understanding arises when several such components interact with each other in an analog system. It becomes quite difficult to determine which mode a component is operating on. In order to correctly certify the behavior of a circuit we must analyze all these possible regions of operation. The task of analog verification boils down to proving that one or more components never go into certain regions of operation based on the prevailing circuit conditions.

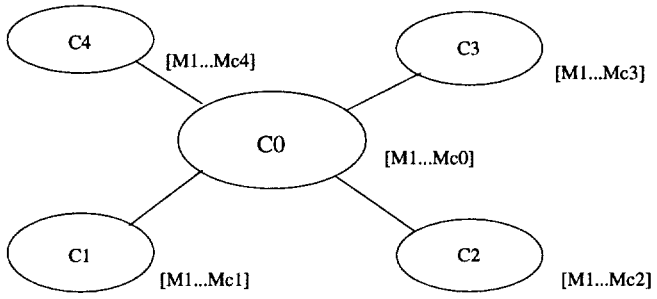

Figure 1: A Component in a General Circuit

For example consider an analog circuit shown in Figure 1. It consists of several components $C_0, \cdots, C_k$ where each component can be in several modes of operation $[M_1, \cdots, M_j]$. Here the mode in which a component goes into depends upon the modes of operation of other components to which it is connected. The circuit works correctly for particular configurations of modes among the components. In order to guarantee the correctness of the circuit we must be able to prove that those are the only configurations possible for the circuit.

2.2 Formal system to verify Analog Designs

We know that differential and algebraic equations (DAEs) represent analog components and some analog components have conditional behavior. Analog circuit components typically exhibit algebraic relations among their inputs and outputs. Interconnections of such components follow Kirchoff's Voltage and Current Laws (KVL, KCL) which are again algebraic equations. Hence any analog circuit *implementation* can be captured as a set of algebraic equations. (Differential equations are used to represent some circuits and we can Laplace transform them to algebraic domain). At the same time, *specification* (or expected properties) of the circuit is also a set of algebraic equations. Here we can conclude that an analog circuit verification tool must be able to prove that the set of algebraic equations representing the *specification* (the expected properties) are derivable from another set of algebraic equations representing the *implementation*. If $ComponentProperty_c$ is the characteristic algebraic property of component c in terms of the voltage V_c and current I_c appearing at its terminals and its physical attributes A_c, $VoltageCurrentLaws$ are the algebraic relations representing various KVL and KCL being followed in the circuit then the set of such equations over all components in the circuit must imply the algebraic relation $ExpectedProperty$ representing the circuit property we are interested to verify in terms of the voltages and currents V, I at the external terminals of the circuit.

$$[\forall_c\ ComponentProperty_c(V_c, I_c, A_c)\ \wedge\ VoltageCurrentLaws(V, I)] \Longrightarrow ExpectedProperty(V, I)$$

Here it can be seen that such the verification effort consists of proving a logical relation (implication in the above case) between sets of algebraic equations. From the above discussion we can conclude that in order to verify analog designs we need a formal system $VerifyAnalog$ defined over arithmetic algebra as

$$VerifyAnalog < A_s, A_p, LogicRelation >$$

where A_s is the set of equations corresponding to the structural implementation of the circuit, A_p is the set of properties of the circuit and $LogicRelation$ is the expected relation between A_s and A_p which are sets of algebraic equations. Traditional ways of proving such a relation is to solve the set of simultaneous equations and show that those solutions satisfy the expected relation.

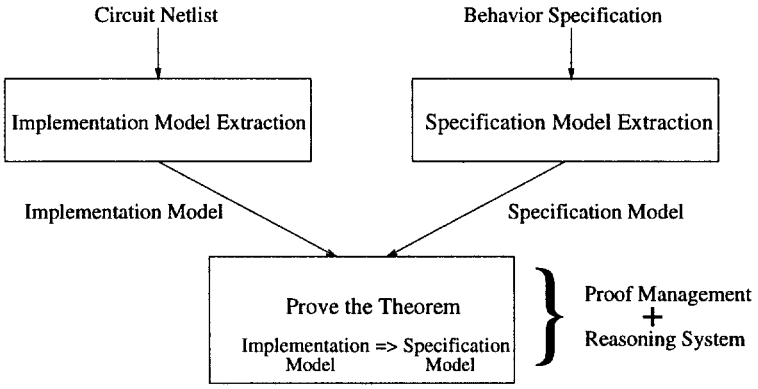

Figure 2: Verification Technique

There have been very few efforts for formal verification of such circuits. Difficulties lie in the reasonable modeling of such components and properties. In the above discussion we have seen that a formal verification system should have the ability to capture the *specification* and *implementation* as sets of algebraic equations and a reasoning mechanism to prove the expected *logic relations* between the two sets. We have also seen that analog components can have conditional behavior and the expected behavior of the analog circuit can also be conditional, our formal system needs to address this issue. In other words our verification tool must have a proof manager to examine every possible mode of operation of the analog circuit and in each one of them it must be able to prove that the implementation has the expected logic relation with the specification. Thus we can summarize that the proposed analog verification tool must have

- model extraction mechanism for the implementation and specification as sets of algebraic equations,
- a proof manager which handle conditional cases,
- a reasoning system (decision procedures) to prove relationships between sets of algebraic equations.

We have depicted such a technique in figure 2. We can see that a theorem prover is an ideal candidate for such a verification system. A theorem prover has a proof manager and a set of decision procedures (over linear algebraic equations) which can prove the expected relationships. Decision procedures of a theorem prover apply for linear algebraic equations and so we are handicapped to use piecewise linear models and are not able to prove any non linear relations. We can overcome such an handicap by enhancing the decision procedures to support nonlinear algebraic equations.

2.3 Nonlinear reasoning mechanism

Algebraic decision procedures supported in theorem provers are linear in nature. This had prompted researchers to use piecewise linear (PWL) models for non-linear analog components [1], [2]. Due to

inherent approximating nature of PWL models they introduce errors in our component modeling. Although PWL behavior models have been extensively used for study, analysis and simulation of analog components for their ease of understanding and availability of mathematical techniques to handle them, they lead to inconsistencies in the structural modeling of analog circuits. Inconsistent modeling in a formal verification effort like ours can lead to *false positives* and in our verification attempt we have to be extremely careful to avoid them.

Typically functionality is verified for known values of circuit paprameters, it is possible to verify functionality of analog designs written over symbols representing terminal values and circuit parameters. These are stronger relationships which are valid over all values of circuit parameters. Algebraic equations describing such circuit implementations are nonlinear and proving such relationships will again require support for nonlinear reasoning. Hence we see that, in order to use correct models of analog components and prove more generic properties we need the support of nonlinear reasoning. Our formal verification tool $VerifyAnalog < A_s, A_p, LogicRelation >$ needs to be extended with a new reasoning mechanism (nonlinear decision procedures). In this paper we have discussed such a new formal system $VerifyAnalog_{nonlinear} < A_{s,nonlinear}, A_{p,nonlinear}, LogicRelation >$ which can produce judgments about $LogicRelation$ between sets of nonlinear algebraic equations $A_{s,nonlinear}$ and $A_{p,nonlinear}$.

3 Formal Model Extraction

It is our attempt to compare a circuit implementation (given as a netlist) against a specification (given in a behavioral specification language). In the previous section we mentioned that formal model extraction was the first step in a verification attempt. We have attempted to extract the implementation and specification models from a synthesis environment, VASE [12]. CAD tools are indispensable in modern design environment and they adapt well for the purpose of verification as they provide hardware description language (HDL) for behavior specification and a well characterized component library whose components can be expressed in logic. In the following subsections, we briefly present strategies to extract models for the implementation and specification. Details of such model extraction strategies can be found in [2], [13].

3.1 Implementation Model Extraction

A circuit is given as a hierarchical net-list of components such that primitive components appear as the leaf nodes. In the following subsections we describe the modeling of such a circuit. We first need to characterize the primitive components. This is followed by the process of capturing the Kirchoff's Voltage and Current Laws. Finally we compose the hierarchical components consisting of primitive ones and continue the process upwards in the hierarchy till we can describe the complete circuit.

3.1.1 Characterization of the primitive components

Each component is modeled by a Boolean-valued function whose parameters are the voltages and currents at its terminals and the attributes of the component. Body of the function must be the relation which must be satisfied by the parameters. Suppose we want to describe a simple analog device like a resistor. It has two terminals, say a and b. The voltage at the two terminals are V_a and V_b, there is a current of I flowing through it and it has a non zero resistance of R, then it will satisfy the following Boolean predicate.

```
resistor(Va,Vb,I,R:real) :  bool = Va-Vb= I*R
```

Following is the description of a capacitor in terms of the Laplace variable s. Here s is a complex number

and this description can be expanded into equivalent real and imaginary parts over real variables.

```
capacitor(Va,Vb,I,C:real) :  bool = sC(Va-Vb)= I
```

We use the above description of the capacitor for frequency domain analysis of analog circuits.

```
opamp-nonideal(V1,V2,Vout,Iin,A,Rin:real) :  bool = Vout=A*(V2-V1) and V2-V1 = Iin*Rin
```

We have the above boolean function *opamp-nonideal* representing a simplified low frequency small signal behavior of the non-ideal operational amplifier. Here V_1 and V_2 are the voltages at the input terminals, V_{out} at the output terminal, I_{in} is the current into the input terminals, A is the open loop gain and R_{in} is the input resistance. A and R_{in} are the imperfections of the opamp and in its ideal behavior $A = \infty$ and $R_{in} = \infty$. An ideal opamp is expressed below is frequently used in circuit analysis,

```
opamp-ideal(V1,V2,Iin:real) :  bool = V2=V1 and Iin=0
```

Following is the description of a transistor having Cutoff, Linear and Saturation regions of operation.

```
transistor(Vg,Vd,Vs,Ids,Vt,beta:real) :  bool =
        if Vg-Vs-Vt <= 0
          then Ids=0                                          /*Cutoff*/
               elsif Vg-Vs-Vt > Vd-Vs
                    then Ids= beta*(Vd-Vs)*(Vg-Vs-Vt)         /*Linear*/
                    else Ids= beta*(Vg-Vs-Vt)*(Vg-Vs-Vt)      /*Saturation*/
        endif
```

V_g, V_d, V_s are the terminal voltages at the gate, drain and source respectively, I_{ds} is the drain to source current and threshold voltage V_t and *beta* are the physical attributes of the transistor. It can be noticed that transistor is described by non linear algebraic equations in Linear and Saturation modes of operation.

3.1.2 Interconnection of Components

In the previous sections primitive analog components were described as the relations being satisfied by the attributes of the components with respect to the terminal voltages and currents. A circuit is a collection of such components where in addition to the laws of the individual components being satisfied, Kirchoff's Voltage and Current Laws of the circuit must also be followed.

Let us consider a simple adder, as shown in Figure 3. The analog components Resistors (R_1,R_2,R_3,R_f) and Opamp (*op*) are connected to the external terminals (a,b,Out) and internal nodes (1,2). We can see that for the circuit to function correctly, each of the components must satisfy their own behavior described by the functions of the previous section.

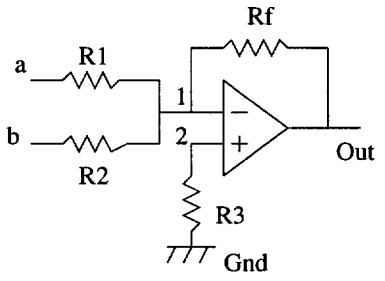

Figure 3: An Example to explain Interconnection of Components

In Figure 3 an expanded view of the adder circuit has been depicted. By Kirchoff's Current Law, we

know that the sum of all currents at a node is zero. Hence at the internal nodes, we must have

```
Currentlaw1(I1,I2,If,Iin:real) :  bool = I1 + I2 + If + Iin = 0          /* At Node 1 */

Currentlaw2(Iin,I3:real) :  bool = Iin + I3 = 0                          /* At Node 2 */
```

The Boolean functions Currentlaw1 and Currentlaw2 must be satisfied by the components of the adder.

3.1.3 Composition of Components

There are many ways to represent circuits. We will view a circuit (see Figure 4) as consisting of;

- a set of nodes $(n_1 \cdots n_k)$;
- a set of components $(c_1 \cdots c_k)$ whose terminals (called the internal terminals of the circuit) are connected to the nodes;
- a set of external terminals $(e_1 \cdots e_k)$ also connected to the nodes.

A sub-circuit can be written as a predicate over all voltages and currents at its external terminals such that there exist some voltages and currents at the internal nodes satisfying the constraints imposed by the characteristic behaviors of the individual subcomponents and the conserving the Current and Voltage Laws. Here we have used current at a node to indicate the currents flowing into the node from all connected branches.

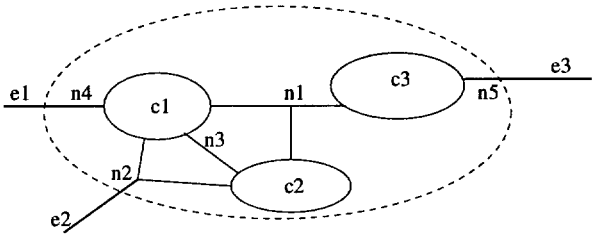

Figure 4: A Sub-Circuit

A circuit or a sub-circuit called Hcomp is shown in Figure 4. Attaching (voltage, current) tuples with all nodes, external as well as internal, Hcomp can be described as,

```
Hcomp(Ve1,Ie1,Ve2,Ie2,Ve3,Ie3,A1,A2,A3) :  bool =
  Exists(Vn1,In1,Vn2,In2,Vn3,In3):
    C1(Ve1,Ie1,Vn1,In1,Vn2,In2,Vn3,In3,A1) and C2(Vn1,In1,Vn2,In2,Vn3,In3,A2) and
    C3(Ve3,Ie3,Vn1,In1,A3)and VoltageCurrentLaws(Ve1,Ie1,Ve2,Ie2,Ve3,Ie3,Vn1,In1,Vn2,In2,Vn3,In3)
```

In the above description, attribute of a component C_j is given by A_j. We have shown only one attribute per component where as a component can have any number of them. Using our implementation model extraction technique discussed above, we can extract the structure of the adder circuit shown in Figure 3 as the following boolean function. We have not used any explicit voltage laws here, but we have associated each node in the circuit with a single voltage variable which follows the Kirchoff's Voltage law.

```
adder (Va,I1,Vb,I2,Vg,I3,Vout:real,R1,R2,R3,Rin,Rf,A:posreal):  bool =
 Exists (V1,V2,Iin,Ifb:real):
   resistor(Va,V1,I1,R1)and resistor(Vb,V1,I2,R2)and resistor(V2,Vg,I3,R3)and Currentlaw2(Iin,I3)
   and resistor(Vout,V1,Ifb,Rf) and opamp(V1,V2,Vout,Iin,A,Rin) and Currentlaw1(I1,I2,If,Iin)
```

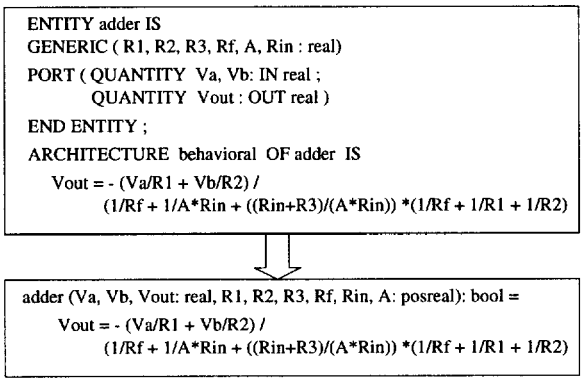

Figure 5: Specification Model Extraction

3.2 Specification Model Extraction

A behavior specification of the circuit is given by the user in a specification language whose formal semantics is understood. A subset of VHDL-AMS [14] is used as the input specification language for some Analog Synthesis Tools. There has been effort on part of researchers [12] to use a subset of this specification language for synthesis purposes. The idea here is to give the user ability to specify a wide range of analog behaviors and retain the ease of transformation of the language constructs into analog circuit. We have used a similar subset of VHDL-AMS for behavior specification. In our subset a set of DAEs and conditional statements can describe a wide range of analog behavior. Algebraic equations are capable of describing time-invariant behavior which are constant algebraic relation between input and output. Differential equations are used to describe frequency dependent behavior. We have identified some constructs of VHDL-AMS which can be translated to a formal behavioral model,

- external terminals of an ENTITY are the PORTS;
- body of the ARCHITECTURE of an ENTITY consists of simple simultaneous statements which can be DAEs or conditional statements;

Translation of such a specification of an adder into a formal model is shown in Figure 5. PORTS are translated as the external terminals, GENERICS are attributes of the design, CONSTANTS are variables having a constant value, local QUANTITIES used inside the ARCHITECTURES are existentially quantified in the boolean functions and simultaneous statements in the body of the ARCHITECTURE are directly translated as conjuncted algebraic relations.

We verify the behavior written in terms of differential equations by frequency domain analysis. In order to facilitate this we apply Laplace transformation to the differential equations. Laplace transformation of a differential equation is an algebraic equation in terms of the Laplace variable s. A specification having differential equations is Laplace transformed into equivalent s-domain algebraic equations which are then translated into behavioral models as boolean functions in real and imaginary domains.

Apart from top level behavior specification we can also specify properties we might be interested to verify. We can look at the verification of such properties as query evaluation over the circuit implementation. We can specify these properties in our subset of VHDL-AMS and extract *property* models from such specifications by our technique discussed in this section.

4 Proof Management

In figure 2 we have illustrated that the verification tool must prove the theorem that there is a relationship between the extracted implementation and specification models. We know that these models contain boolean predicates written over conditional cases. A theorem prover has the ability to manage such prove attempts. It provides user interface by which user can guide a proof attempt using basic proof commands and it also supports scripts which help automate such attempts. Theorem prover maintains a proof tree where each node is a proof goal. Through the process of proving, user has to complete the tree such that all the leaves are true. Each node or proof goal is a sequent which is a sequence of formulas in the antecedents and consequents. Conjunction of antecedents must imply the disjunction of consequents. Theorem we have to prove has the following sequent :

$\vdash \forall$ e ($\exists$ i,a : Implementation(e,i,a) $\Rightarrow$ Specification(e))

Here e represents the external variables, i represents the internal variables and a represents the attributes of the implementation. Our proof strategy essentially is,

- Remove the existential quantification in the antecedent using *skosimp*, so that we have Implementation(e!,i!,a!) $\vdash$ Specification(e!)
- Recursively use *expand* to rewrite the functions at the circuit, block and component levels.
- Use *split, lift-if* to remove the conditional cases. Now we have a set of algebraic equalities and inequalities in the sequent.
- Invoke the decision procedures to finally prove the consequents.

We have used PVS theorem prover as our proof manager and also mentioned the basic PVS proof commands used.

5 Non linear reasoning mechanism

In the previous section we saw that last step of the proof strategy is to invoke a decision procedure to prove the sequent. If we only had linear algebraic equations in our sequent then we could have invoked the theorem prover's own decision procedures to prove the relationship but we are dealing with nonlinear algebraic equations and so we need support of an external nonlinear reasoning mechanism. In this section we illustrate how a theorem prover can be interfaced with a CAS to develop a stronger reasoning system. We first compare the theorem prover and CAS as separate systems and various types of integrations possible depending on specific problem domains. In the following subsection we describe our problem domain and illustrate how they can be interfaced in such a scenario to form a sound reasoning system.

5.1 Comparing theorem provers and computer algebra system

Problem solving in mathematics often requires the application of both procedural algebraic knowledge (algorithms) and deductive knowledge (theorems). The advantages of combining both strategies have been recognized by both communities: symbolic computation and analytical reasoning. Some of these advantages concern the introduction of mathematical theories and arithmetics, in particular real numbers, into provers, as well as providing logical languages and justifications to symbolic calculators. Two aspects must be further investigated: (i) the problem of combining algorithms and theorems into one system, (ii) the heterogeneous integration of several packages.

On one hand classical CAS like Mathematica usually offer a straightforward programming language with ad-hoc implementations of rewriting. On the other hand theorem proving has shown to be an important field interfacing artificial intelligence and mathematics. It attempts to perform symbolic calculations of mathematical proofs by computers. However, there are no environments integrating theorem provers and CAS which consistently provide the interface capabilities of the first and the powerful arithmetic of the later systems.

5.1.1 Interaction between theorem provers and computer algebra systems

The major aspect is the integration of several systems in a common environment. Different possibilities to integrate symbolic calculators and theorem provers are given in [3]. We discuss some of them here

- A well known approach towards introduction of theorem proving into CAS is Analytica [4], a Mathematica package to prove theorems in elementary analysis. It is able to deal with internal mathematical knowledge of mathematica and garauntee correctness of certain operations.
- [15] introduces a architecture for open mechanized reasoning systems which consists of reasoning theory as well as control and interaction component.
- [5] describes a bridge between the theorem prover HOL and a CAS, whose interaction can be classified as master-slave (HOL as master). CAS is used as a oracle guiding the proof rigorously done in HOL.
- [6] describes an interaction between theorem prover Isabelle and CAS Maple as a master-slave relation. This is a first step in a more general direction of open heterogeneous environment.

We have described different forms of interaction between theorem provers and CASs. These interactions are based on the notion of *trust* between the two systems. In some of approaches there is *complete trust* between the two components, where as other approaches work on more domain specific restricted trust. Typically, theorem provers are used as the front-end proof handlers and CASs are used as back-end support reasoning systems. The results from the CASs are verified by the theorem provers for correctness.

5.1.2 Computer algebra systems as term rewrite systems

Computer algebra systems contain equation solvers. It would be desirable to use them as unification procedures for algebraic equation based theories. Unfortunately, this is not possible in all cases. This is the reason why such decision procedures are not available as part of theorem provers rewrite procedures. Conditional rewrite rules can be used to model the calculations performed by the CASs and CASs can be considered as term rewrite systems. Premises in the rules are important to prevent the CASs from incorrect calculations. In design of a hybrid system these premises should be carefully selected. In order to use CASs for term rewriting we have to study the problem domain.

5.2 Our problem scenario

From our discussions before, we know that in order to use the support of a CAS in formal reasoning we must understand the problem domain in which we are trying to use it and the nature of support we need from the CAS. Based on the above we can design an interaction between the theorem prover and the CAS.

Our proof attempt (as shown in Figure 6) is to formally establish a relation between two sets of algebraic equations. The first set consists of equations representing the circuit implementation and the second set represents the specification (or expected behavior). The set of equations representing the implementation are in terms of both internal and external terminal symbols, where as the specification is only in terms of the external symbols. In our proof attempt we have to symbolically eliminate the internal symbols such that we can establish the desired relation amongst the external symbols. This kind of variable elimination by algebraic manipulations is possible if the equations are linear in nature. In absence of any support for sound non linear reasoning we piece wise linearized some non linear equations in the structural model. We can make sound judgment using linear equations but they do not correctly represent the implementation. Linearization can also lead to false positives and jeopardize our proof attempt.

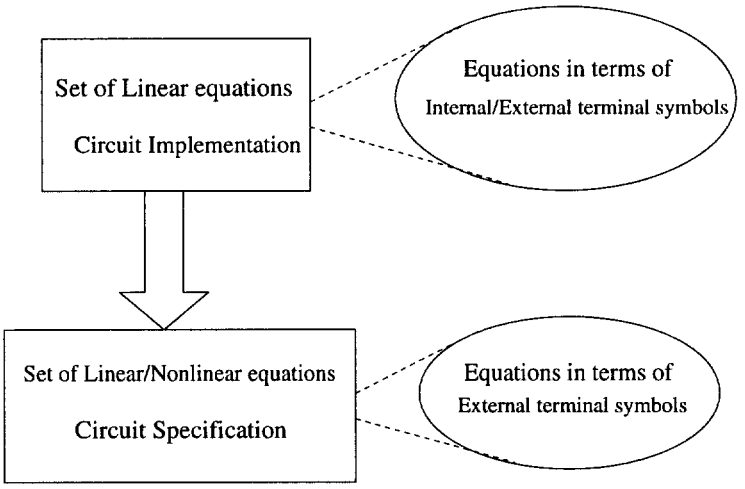

Figure 6: Our Basic Proof Attempt

Looking carefully at the nature of non linear equations we can notice that they are quadratic in nature. These are the equations denoting the DC transistor behavior. We know that any quadratic equation is equivalent to two linear equations. A quadratic equation like $ax^2 + bx + c = 0$ can be written as $x = -b/2a + (b^2 - 4ac)^{1/2}/2a$ and $x = -b/2a - (b^2 - 4ac)^{1/2}/2a$. If a, b, c are constants then x can take real or complex values but if they are variables then we cannot obtain such transformations for x where it can take exact (real or complex) values. At the same time we must remember that we are not trying to solve independent quadratic equations but these equations are part of a set of other linear/ quadratic equations. The symbols a, b, c can be constants or other free variables which are internal/external symbols. We have already mentioned that it is our attempt to eliminate the internal symbols. Hence given a set of linear/quadratic algebraic equations in terms of internal/external symbols (SetA), we can try to eliminate the internal symbols with the aid of a CAS to get an equivalent set of equations in terms of the external symbols (SetB). This is illustrated in figure 7. Examining the set of equations SetB returned by the CAS we can see that following three cases are possible,

- Equations in SetB are decomposed linear equations among external variables such that all constants are real. In this case we can completely replace SetA by SetB. Using the set of linear equations in SetB we can proceed in our proof attempt. The real constants here can be rationals or irrationals. In this case we can use the CAS again to check if the solution represented in SetB satisfies the equations representing the specification.
- Equations in SetB are linear equations among external variables such that some of the constants are complex. As we understand that all relations or values for the external symbols are real, so we

can use this case as a *false*. The symbols typically represent voltage/current values can we know that should always assume real values. A complex relation in SetB can only mean that equations in SetA represent a circuit not working correctly as the equations are inconsistent with each other.

- Equations in SetB are nonlinear equations among external variables. In this case we can check if that is the nonlinear relation (in the specification) which we wanted to verify. Using the CAS we can check the equivalence of such nonlinear equations. If they are equivalent then we say that specification is derivable from the implementation otherwise we cannot conclude anything from the manipulations provided by the CAS and the verifier cannot say anything in that case.

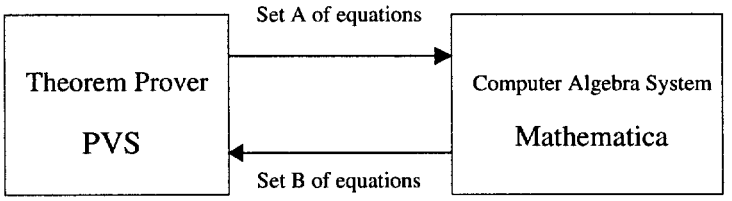

Figure 7: Theorem prover interacting with CAS

5.3 Integrating PVS and Mathematica

In the previous section we discussed the issues of using a CAS as a support reasoning system. We presented the peculiarities of our particular problem domain and described how we are going to interpret the results provided by the CAS. In this section we describe the implementation of the technique using PVS as the theorem prover and Mathematica as the CAS.

We know that the decision procedures of the theorem provers rewrite symbolically equivalent expressions. Rewrite rules of typical theorem provers like PVS are based on equality of expressions. These equality rules can be provided by the user given axioms or can be generated by underlying decision procedures of the theorem prover. When we aid the theorem prover's decision procedures with CAS support, we are using CAS as a slave in the master slave configuration. PVS sends the set of equations to Mathematica asking it to eliminate the internal variables. Mathematica returns the manipulated set back which is interpreted based on the cases discussed before. In figure 8 we show such an interface between PVS and Mathematica through an Interface Unit.

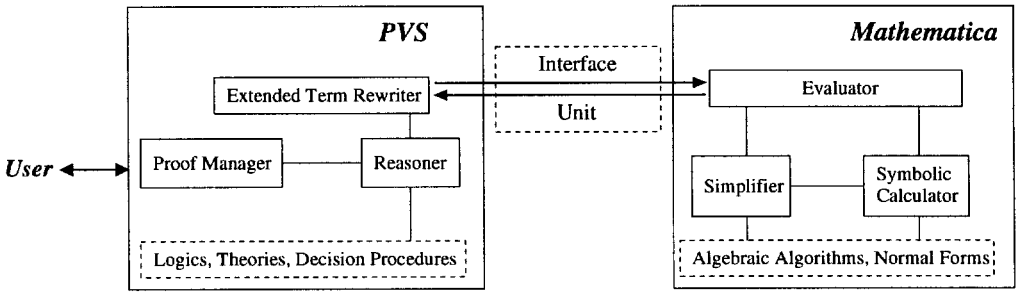

Figure 8: PVS interacting with Mathematica

User interacts with the PVS proof manager. Term rewriting is performed by the reasoner which rewrites expressions based on axioms and decision procedures. This rewriting has been extended using an Extended Term Rewriter which on invocation of certain proof commands uses Mathematica to rewrite a set of equations by an equivalent set. It interacts with Mathematica over an Interface Unit by dumping the sequents into a prefixed file, say IntFile. Mathematica on the other hand has an evaluator which

works over the simplifier and symbolic calculator. There are underlying algebraic algorithms to aid the process of simplification.

Interface Unit is a process which waits on IntFile to be written by PVS. Once IntFile has been written by PVS with the new set of equations, Interface Unit translates it into the equivalent syntax of Mathematica. It uses the Mathlink library interface to interact with Mathematica and asks it to solve the given set. Mathematica solves the given set of equations and returns the equivalent set. Now the interface unit examines the returned set of equations for the three cases and writes the appropriate response into a predetermined file (ResFile) on which the PVS proof manager in waiting. PVS then proves the current sequent with the help of this response. Interface Unit algorithm has been explained in figure 9.

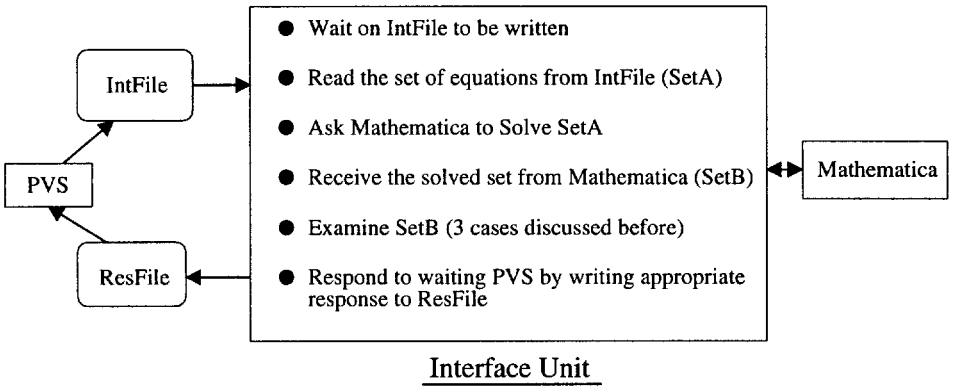

Figure 9: The Interface Unit

6 Results

In this section, we illustrate how interaction of PVS and Mathematica discussed in this paper can be used for functional property verification. Firstly, we present functionality verification of the adder discussed in section 3. We show how a complicated expected functionality can be proven by our technique. In the next subsection we apply our technique to other familiar circuits with well known transfer functions. We further present application of our technique for DC analysis of transistor netlists using nonlinear models of constituent transistors. We had extracted the implementation and specification models of the adder and it is our aim to prove the theorem: *implementation* $\Rightarrow$ *specification*. Following is one of the sequents generated during the proof of the adder_theorem.

```
{-1}  resistor(Va!1, V1!1, I1!1, R1!1)
{-2}  resistor(Vb!1, V1!1, I2!1, R2!1)
{-3}  resistor(V2!1, 0, I3!1, R3!1)
{-4}  resistor(Vout!1, V1!1, Ifb!1, Rf!1)
{-5}  opamp_nonideal(V1!1, V2!1, Vout!1, Iin!1, A!1, Rin!1)
{-6}  Currentlaw1(I1!1, I2!1, Ifb!1, Iin!1)
{-7}  Currentlaw2(Iin!1, I3!1)
  |-------
[1]   Vout!1
       = -(Va!1 / R1!1 + Vb!1 / R2!1)/
          (1 / Rf!1 + 1 / (A!1 * Rin!1)
             + 1 / R1!1 * ((R3!1 + Rin!1) / (A!1 * Rin!1))
             + 1 / R2!1 * ((R3!1 + Rin!1) / (A!1 * Rin!1))
             + 1 / Rf!1 * ((R3!1 + Rin!1) / (A!1 * Rin!1)))
```

Expanding the boolean functions in the sequent, we can extract the following set of algebraic equations to represent this implementation.

```
Va-V1= I1*R1  Vb-V1= I2*R2  V2-V1= Iin*Rin  Vout-V1= If*Rf  Vout= A*(V2-V1)  V2= I3*R3
I1+I2+If+Iin=0  I3+Iin=0
```

Using the above equations we will be able to show the following relation for the Adder

$$Vout = -\frac{\frac{Va}{R1} + \frac{Vb}{R2}}{\frac{1}{Rf} + \frac{1}{A*Rin} + \frac{Rin+R3}{A*Rin} * (\frac{1}{Rf} + \frac{1}{R1} + \frac{1}{R2})}$$

Now we present the proof steps in verification of the above theorem. We can notice that we have used a special proof command *ASSERTALL*. This is a proof strategy written in using PVS's script language. It invokes the interface unit by sending the current sequent from PVS, waits for the interface unit to give the simplified formula (obtained from Mathematica) and replaces the formulas in the antecedent with the simplified formula. The goal in the sequent is thus proven with Mathematica's assistance.

```
(|adderproof|
 (|adder_theorem| "" (SKOLEM!)
  (("" (FLATTEN)
   (("" (EXPAND "adder_behavior")
    (("" (EXPAND "adder_structure")
     (("" (SKOLEM!)
      (("" (FLATTEN)
       (("" (EXPAND "resistor")
        (("" (EXPAND "opamp_nonideal")
         (("" (EXPAND "Currentlaw1")
          (("" (EXPAND "Currentlaw2")
           (("" (FLATTEN)
            (("" (ASSERTALL)
             (("" (PROPAX) NIL)))))))))))))))))))))))
```

6.1 Proving known symbolic relations

We have applied our verification technique to other analog designs. We present them in Table 1. The examples presented here are typical opamp-resistor networks, similar to the adder example presented before. The circuits are categorized according to the number of different analog components (opamps, resistors, capacitors) used in them. The last column indicates the number of universal variables over which their functionality can be expressed.

The circuits presented here have well understood behaviors, in other words their functionality can be expressed as algebraic or differential equations over symbols representing voltages, currents at their external terminals and physical attributes (like resistance, capacitance etc).

Behavior of circuits (1-5) can be expressed as pure algebraic equations. Circuits (6-12) have been verified in the Laplace domain as the DAEs representing their behavior are Laplace converted to equivalent algebraic equations in real and imaginary domains. Implementation models of these circuits are again extracted as algebraic equations in real and imaginary domains. The execution time for verification of these examples is of the order of less than a minute on Sun Sparc Ultra 2 workstation (296 MHz, 128Mb RAM). The execution time consists of proof management by PVS and interaction of the interface unit with Mathematica.

CktNo	Input Circuit	Opamps	Resistors	Capacitors	ExtVariables
1	AdderIdeal	1	3	0	6
2	AdderNonIdeal	1	4	0	9
3	Subtractor	1	4	0	7
4	BridgeAmplifier	1	3	0	6
5	Instrumentation Amplifier	1	4	0	7
6	Integrator	1	1	1	7
7	Differentiator	1	1	1	7
8	PhaseShifter	1	3	1	9
9	TapePreAmplifier	1	3	2	10
10	HighPassFilter	1	2	1	8
11	LowPassFilter	1	2	1	8
12	WidePassFilter	1	2	2	9

Table 1: Proving characteristic relations of well known circuits

6.2 DC Analysis of Transistor Netlists

We know that transistors behave nonlinearly in linear and saturated modes of operation. In order to make judgment about them using linear algebra we have to use piecewise linearization to approximate such behavior. We have used our integration of the theorem prover with CAS for establishing DC conditions of transistor net-lists using nonlinear models of transistors. In table 2 we have presented some of transistor net-lists whose DC conditions were checked. Columns 4,5 present the number of unknown variables and equations in these circuits. We know that transistors have three possible regions of operation and so a

CktNo	Input Circuit	Transitors	Variables	Equations
1	Invertor	2	4	9
2	Nand	4	6	8
3	Nor	4	6	8
4	Differential Amplifier	4	6	9
5	Current Mirror	4	6	9

Table 2: DC Analysis of Transistor Netlists

circuit with four transistor has 81 possible configurations. The circuit has been constructed to perform correctly for a few of these configurations. Exhaustively checking the correctness of these circuits in all such modes is an exponentially complex task. Simulators like Spice [10] are used to establish DC conditions of such circuits. Simulators take user inputs, make guesses, use convergence algorithms and employ several heuristics to determine the stable DC operating modes for prevailing circuit conditions. Our approach is certainly more correct as we try to solve the set of equations completely in each possible circuit configuration but at the same time number of possible configurations increase exponentially with circuit size. The execution time for verification of these examples is of the order of a few minutes on Sun Sparc Ultra 2 workstation (296 MHz, 128Mb RAM). The execution time consists of proof management by PVS and interaction of the interface unit with Mathematica.

7 Summary

In this paper we have seen that integration of a computer algebra system with a theorem prover enhances its ability to handle more complicated proofs. The theorem prover is still used as a proof manager where as the CAS provides additional power of nonlinear reasoning to its decision procedures. We have to realize that we cannot provide CAS support for all situations and results obtained from CAS has to be

correctly interpreted. Nonlinear equations have multiple solutions which need to be analyzed correctly. We have studied the type of algebraic equations we are dealing with and have provided ways to interpret the results obtained from the CAS. We have presented interaction of a theorem prover PVS with a CAS Mathematica and have found such an integrated approach useful for verification of analog design functionalities.

References

[1] Keith Hanna. "Automatic Verification of Mixed-Level Logic Circuits". *Proceedings of Formal Methods in Computer-Aided Design*, pages 133–148, 1998.

[2] Abhijit Ghosh and R. Vemuri. "Formal Verification of Synthesized Analog Designs". *Proceedings of International Conference on Computer Design*, October 1999.

[3] K.Homann and J.Calmet. Combining theorem proving and symbolic mathematical computing. In J.Calmet and J.A.Campbell, editors, *Artificial Intelligence and Symbolic Mathematical Computing (AISMC-2)*, Lecture Notes in Computer Science. Springer-Verlag, 1994.

[4] E.Clarke and X.Zhao. Combining theorem proving and theorem proving: Some problems of ramanujam. In A.Bundy, editor, *Automated Deduction (CADE-12)*, number 814 in Lecture Notes in Computer Science, pages 758–763. Springer-Verlag, 1994.

[5] J.Harrison and L.Thery. Extending the hol theorem prover with a computer algebra system to reason about the reals. In J.J.Joyce and C-J.H.Seger, editors, *Higher Order Logic Theorem Proving and its Applications (HUG'93)*, number 780 in Lecture Notes in Computer Science, pages 174–184. Springer-Verlag, 1993.

[6] Clemens Ballarin, Karsten Homann, and Jacques Calmet. "Theorems and Algorithms: An Interface between Isabelle and Maple". *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*, July 1995.

[7] N.Shankar, S.Owre, and J.M.Rushby. *"The PVS Proof Checker: A Reference Manual (Beta Release)"*, March 1993.

[8] S.Wolfram. *Mathematica: a System for Doing Mathematics by Computer.* Addison-Wesley, 1991.

[9] Jacob Millman and Christos C. Halkias. *Integrated Electronics.* Tata Mcgraw Hill, 1991.

[10] L.W.Nagel. "SPICE2: A Computer Program to Simulate Semiconductor Circuits". Technical Report Memo ERL-M520, Berkley,Calif, May 1975.

[11] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design.* Addison Wesley Publishing Company, 1993.

[12] Alex Doboli, N. Dhanwada, A. Nunez-Aldana, Sree Ganesan, and R. Vemuri. "Behavioral Synthesis of Analog Systems using Two-Layered Design Space Exploration". *Proceedings of 36th Design Automation Conference*, June 1999.

[13] Abhijit Ghosh and R. Vemuri. "Formal Verification of Synthesized Mixed Signal Designs Using *BMDs". *Proceedings of 13th International Conference on VLSI Design*, January 2000.

[14] *"IEEE Standard VHDL Language Reference Manual (Integrated with VHDL-AMS changes)".* IEEE Std.1076.1.

[15] F.Giunchiglia, P.Pecchiari, and C.Talcott. "Reasoning Theories- Towards an Architecture for Open Mechanized Reasoning Systems". 1994.

Extracting formal models from synthesised Verilog

(Category B submission)

Mike Gordon, David Greaves, Konrad Slind

University of Cambridge Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG
U.K.
mjcg@cl.cam.ac.uk
http://www.cl.cam.ac.uk/~mjcg

Abstract. The verification of implemented algorithms can be decomposed into (i) showing properties of an abstract representation of the algorithm and (ii) showing the abstract representation is correctly implemented. In this paper we discuss an approach to (ii) based on automatically extracting formal logical models from the output of the CV3 Verilog compiler. CV3 output is processed by inference using HOL, but the use of HOL is completely hidden from the user. The model extraction is packaged as a Unix tool that reads a Verilog source file and creates a HOL theory file. Command line options are used to select the kind of model produced and the form in which it is represented. This work illustrates the approach being explored by the Prosper project [9] in which user level tools make use of an internal proof engine.

1 Introduction

Formal verification is often applied to algorithms represented abstractly. For example, Harrison [5] verifies the correctness of floating point algorithms expressed in a simple imperative programming language semantically embedded in HOL, and Paulson [8] verifies properties of cryptographic protocols modelled directly in Isabelle/HOL[1].

There are several approaches to ensuring that abstract representations of algorithms are correctly realised by concrete implementations, for example:

- formally refine the algorithm to implementable code or hardware;
- use a verified compiler;
- use an unverified compiler, but check for each run that the output is equivalent to the input.

The approach described here is similar to the last of these: we automatically extract logical models from the output of an unverified (and rapidly evolving) Verilog compiler. These models are in higher order logic and are suitable for further analysis by theorem proving or model checking. Application scenarios include post synthesis checking of properties of the implementation and showing that the synthesised implemention matches previously formulated abstract models.

Industrial tools exists to check the equivalence between HDL input and the results of synthesis, but as far as we know these use ad hoc semantics of the input HDL and are mainly intended to ensure that if synthesisable HDL is verified by simulation, then this verification will also apply to the results of synthesis.

The rest of the paper proceeds as follows: first an overview of the relevant aspects of CV3 is given, next the modelling in HOL of the result of compiling Verilog is discussed, then the various options for processing by HOL of the synthesis output are described.

[1] Isabelle/HOL is Isabelle's instantiation to simply typed higher order logic [7].

2 CV3

CV3 is a tool written by David Greaves that reads a Verilog source file and generates output in a variety of formats. It is implemented in a version of Standard ML[2] and "supports compilation of nearly the whole Verilog language but has some bugs".[3]

The output of CV3 is determined by a technology library. The library used here is called `cv2.100` and consists of various combinational components and a positive edge-triggered Dtype. The example in this paper only uses inverters (`INV`), 2-input exclusive-or gates (`XOR2`) and Dtype flip-flops (`DFF`). These components have simulation models written in Verilog. The models of the combinational components all have small delays[4] to avoid asynchronous (zero-delay) loops.

```
module INV(o, i);
  output o;
  input i;
  assign #2 o = ~i;
endmodule

module XOR2(o, i1,i2);
  input i1, i2;
  output o;
  assign #3 o = i1 ^ i2;
endmodule
```

The Dtype has a more complex model that includes additional variables (`last_d`, `last_clk`) and tasks for generating and displaying simulation output.

```
module DFF(q, d, clk, ce, ar, spare);
  output q;
  reg q;
  initial q = 0;
  input clk;          // Clock (positive edge triggered)
  input d;            // Data input
  input ce;           // Clock enable
  input ar;           // Asynchronous reset
  input spare;

  integer last_d, last_clk;

  always @(posedge clk or posedge ar)
        if (ar) q <= #10 0;
        else if (ce)
                begin
                if ($time - last_d < 5)
                  $display("Time %t,DFF %m violated set-up time",$time);
                last_clk = $time;
                q <= #10 (d & 1);
                end

  always @(d)
        begin
        last_d = $time;
        if ($time - last_clk < 4)
          $display("Time %t,DFF %m violated hold time",$time);
        end
endmodule
```

[2] CV3 is written in a version of ML implemented by David Greaves.

[3] `http://www.cl.cam.ac.uk/users/djg/localtools/oldindex.html`

[4] `#`n specifies a delay of n units of simulation time

One possible output from CV3 is an unflattened (i.e. module hiererarchy preserving) Verilog netlist (`vnl`), another output form is an ML datatype representing the unflattened netlist (`mlout`). The former is more readable, but the latter is used in the interface to HOL.

To illustrate CV3, suppose the file `COUNT2.cv` contains the following Verilog module definitions:

```
module CLK_DIV(clk,ce);
  input clk;
  output ce;
  reg ce;
  always @(posedge clk) ce = !ce;
endmodule

module COUNT2(clk,out);
  input clk;
  output [1:0] out;
  reg [1:0] out;
  wire ce;
  CLK_DIV M1(clk,ce);
  always @(posedge clk) if (ce) out = out+1;
endmodule
```

Executing the command

```
cv3core cv2.100 -root CLK_DIV -vnl $PWD/COUNT2.cv -o CLK_DIV.vnl
```

will write the file `CLK_DIV.vnl` with the following Verilog netlist:[5]

```
module CLK_DIV (clk, ce);
  supply0 LGND; supply1 LVCC;
  input clk;
  output ce;
  wire I100;
  DFF  ce(ce, I100, clk, LVCC, LGND, LGND);
  INV  I100(I100, ce);
endmodule
```

Executing the command

```
cv3core cv2.100 -root COUNT2 -vnl $PWD/COUNT2.cv -o COUNT2.vnl
```

will write the file `COUNT2.vnl` with the following Verilog netlist:

```
module COUNT2 (clk, out);
  supply0 LGND; supply1 LVCC;
  input clk;
  output [1:0] out;
  wire g102,I100;
  wire ce;
  CLK_DIV  M1(clk, ce);
  DFF  i1out103(out[1], g102, clk, ce, LGND, LGND);
  XOR2  g102(g102, out[1], out[0]);
  DFF  i0out101(out[0], I100, clk, ce, LGND, LGND);
  INV  I100(I100, out[0]);
endmodule
```

[5] Automatically generated comments have been removed and the format of the output Verilog has been made more compact.

3 Importing CV3 output into HOL

It is very easy to import the netlists output by CV3 into HOL, because CV3 can generate its output as an ML datatype. The standard representation of structure in HOL [3] associates predicates with components, variables with wires and then expresses the structure using conjunction and existential quantification. For example, when COUNT2 is imported into HOL the Verilog module declaration becomes the following definition.

```
|- COUNT2 CLK_DIV (clk,out_1,out_0) =
    ∃I100 g102 ce.
      CLK_DIV (clk,ce) ∧
      DFF (out_1,g102,clk,ce,LGND,LGND) ∧
      XOR2 (g102,out_1,out_0) ∧
      DFF (out_0,I100,clk,ce,LGND,LGND) ∧ INV (I100,out_0)
```

The components in the `cv2.100` technology library (e.g. XOR2, DFF) are predefined with a user-selected semantics that is discussed below. Any module that has not been predefined (e.g. CLK_DIV) is made a paramenter.

The translation to a HOL-netlist is achieved by the command[6]

```
cv2hol COUNT2.cv COUNT2 COUNT2
```

The first argument is the Verilog source file (`COUNT2.cv`). The second argument (COUNT2) is the name of the Verilog module in the source file that is to be imported into HOL, and the final argument (COUNT2) is the name of the theory to be created. This call to `cv2hol` creates a theory `COUNT2Theory` (which is represented by two files: `COUNT2Theory.sml` and `COUNT2Theory.sig`) containing the definition above.

Often one wants to read in a sequence of modules and create a theory containing them all. If a module M1 is defined and then used in a subsequently defined module M2, then M1 will not be a parameter to M2. However, if M2 is defined first then M1 will be a parameter. The order in which modules are defined is specified by the order they are listed when `cv2hol` is invoked. For example

```
cv2hol COUNT2.cv CLK_DIV COUNT2 COUNT2
```

first reads CLK_DIV and then COUNT2 from the file `COUNT2.cv` and creates a theory `COUNT2Theory` containing

```
|- CLK_DIV (clk,ce) =
    ∃I100. DFF (ce,I100,clk,LVCC,LGND,LGND) ∧ INV (I100,ce)

|- COUNT2 (clk,out_1,out_0) =
    ∃I100 g102 ce.
      CLK_DIV (clk,ce) ∧
      DFF (out_1,g102,clk,ce,LGND,LGND) ∧
      XOR2 (g102,out_1,out_0) ∧
      DFF (out_0,I100,clk,ce,LGND,LGND) ∧ INV (I100,out_0)
```

Since CLK_DIV was defined when COUNT2 was processed, it is treated as a predefined constant and not made a parameter. To get CLK_DIV as a parameter to COUNT2 use

```
cv2hol COUNT2.cv COUNT2 CLK_DIV COUNT2
```

The general form of a command to create a netlist theory is

`cv2hol` *<source>* *<module>* ⋯ *<module>* *<theory>*

where *<source>* is a Verilog source file, each *<module>* is the name of a module declared in the source file and *<theory>* is the name of the theory to be created.

[6] See Section 8 for current status of implementation.

4 Extracting models from netlists

The meaning of a term like `COUNT2(clk,out)` defined by `cv2hol` depends on the meaning of the components `INV`, `XOR2`, `DFF` etc. This is determined by the parent theory that predefines these constants. Currently four theories are provided, each giving a different model of the components.

`simTheory` approximates the "golden" simulation semantics, complete with combinational delays;
`edgeTheory` gives all combinational components zero delay and `DFF` a unit delay on a rising edge;
`tickTheory` gives all combinational components zero delay, shrinks clock cycles to a single 'tick' and models `DFF` as a unit delay on a tick;
`cycleTheory` cycle-based semantics: clock lines are ignored and `DFF` is modelled as a pure unit delay.

The default theory is `cycleTheory`. The other three theories are selected by giving `cv2hol` the argument `-sim`, `-edge` or `-tick` before the Verilog source file.

4.1 The theory `simTheory`

Example definitions of the combinational components in `simTheory` are shown below. Note that the delays correspond to the Verilog simulation models of the components.

```
LVCC(t)            = T
LGND(t)            = F
INV(o,i)           = ∀t. o(t+2) = ¬(i t)
XOR2(o,i1,i2)      = ∀t. o(t+3) = i1 t xor i2 t
```

The HOL model of `DFF` in theory `simTheory` approximates the simulation model, though the waveform monitoring is ignored. A rising edge is defined by

```
rise clk t = ¬(clk t) ∧ clk(t + 1)
```

and then `DFF` is defined by

```
DFF(q, d, clk, ce, ar, spare) =
 (∀t. t<10 ⇒ (q t = F))
 ∧
 ∀t. if ((rise clk t) ∨ (rise ar t))
       then (if ar(t+1)
              then q(t+10) = F
              else if ce(t+1) then q(t+10) = d t
                              else q(t+10) = q t)
       else (q(t+10) = q t)
```

Although the HOL models in theory `simTheory` of the `cv2.100` components use the same delay values as the Verilog simulation models, it is far from clear how the representation of behaviour in HOL corresponds to that generated by the Verilog simulation cycle. This is an important question, since one would like verification by simulation and formal verification to produce consistent results [4]. Attempts so far at reconciling simulation and formal verification semantics are at best rather preliminary (e.g. [2]).

The combinational delays in the Verilog models of the `cv2.100` components are in practice mainly to ensure well behaved simulation, rather than to support timing analysis. The model supports the implementation of storage via combinational loops and the implementation of simulators is such that the delays do not get in the way of efficient modelling.

The HOL theory `simTheory` leads to rather messy formal models that are hard to analyse. In particular, additional state variables are needed to define transition relations (see Section 6). Thus the model `simTheory` is really only of academic interest.

A more tractable model is `edgeTheory` in which there are no combinational delays, but clock edges are still explicit, and transparent latches and gated and derived clocks can be represented.

4.2 The theory edgeTheory

The theory edgeTheory is obtained from simTheory by setting all combinational delays to zero, and giving DFF unit-delay.

```
LVCC(t)              = T
LGND(t)              = F
INV(o,i)             = ∀t. o t = ¬(i t)
XOR2(o,i1,i2)        = ∀t. o t = i1 t xor i2 t

DFF(q, d, clk, ce, ar, spare) =
 (q 0 = F)
 ∧
 ∀t. if ((rise clk t) ∨ (rise ar t))
       then (if ar(t+1)
              then q(t+1) = F
              else if ce(t+1) then q(t+1) = d t else q(t+1) = q t)
       else (q(t+1) = q t)
```

This model is appropriate when one wants to model transparent latches, or flip-flops triggered on rising and falling edges. If every register is clocked on the positive edge of a single clock line, then a simpler representation is obtained by merging the steps between a positive edge and the following negative edge into a single abstract 'tick'. Thus there is no sequence of times when the clock is high.

4.3 The theory tickTheory

The theory tickTheory is obtained from edgeTheory by regarding the clock as a sequence of abstract ticks: clk t = T means there's a tick at time t and clk t = T means no tick at time t. There is no distinction between positive and negative edges and no interval between successive edges of the same clock phase. tickTheory is a temporal abstraction [6] from from edgeTheory. As with edgeTheory all combinational delays are zero. The model of DFF is

```
DFF(q, d, clk, ce, ar, spare) =
 (q 0 = F)
 ∧
 ∀t. if clk t
       then (if ar t
              then q(t+1) = F
              else if ce t then q(t+1) = d t else q(t+1) = q t)
       else (q(t+1) = q t)
```

This model is appropriate when only one kind of edge is used to trigger flip-flops and there are no transparent latches, but gated or derived clocks need to be modelled. With tickTheory a clock cycle is atomic and is associated with a single time. With edgeTheory a clock cycle can take several unitis of time (e.g. between rising edges).

If every register is clocked on a single clock line, then a simpler representation is obtained by abstracting all signals to their values at successive ticks. This corresponds to a 'cycle-based' interpretation. The theory cycleTheory in the next section interprets the cv2.100 components at this abstraction.

4.4 The theory cycleTheory

If all registers have the same clock line, then the following simplified model of DFF can be used.

```
DFF(q, d, clk, ce, ar, spare) =
 (q 0 = F)
 ∧
 ∀t. if ar t
      then q(t+1) = F
      else if ce t then q(t+1) = d t else q(t+1) = q t
```

Note that the clock line `clk` is ignored[7] – time is modelling succesive cycles. There is thus a further temporal abstraction from the timescale used in theory `tickTheory`. If flip-flops are clocked by more than one clock then translation to HOL will give an incorrect model.

4.5 Selecting a parent theory

The component model to be used is specified as the first argument to `cv2hol`

`cv2hol -`<*model*> <*source*> <*module*> ⋯ <*module*> <*theory*>

where <*model*> is one of `sim`, `edge`, `tick` or `cycle`. If no model is specified, then `cycle` is assumed. The theory named <*theory*>`Theory` that is created will have <*model*>`Theory` as a parent.

5 Deriving equations from netlists

When `cv2hol` is invoked, the default is to create a theory just with the translated netlists. If the argument `-eqn` is given, then each translated module is unwound using the definitions of the `cv2.100` components and other modules in the source that are defined earlier. For example, invoking

```
cv2hol -sim -eqn COUNT2.cv CLK_DIV CLK_DIV
```

creates a theory `CLK_DIVTheory` that, as well as the HOL netlist of `CLK_DIV`, also contains the automatically proved theorem

```
|- CLK_DIV (clk,ce) =
    ∃I100.
      ∀t.
        ((t < 10 ⇒ ¬ce t) ∧
         (if ¬clk t ∧ clk (t + 1) then
            ce (t + 10) = I100 t
          else
            ce (t + 10) = ce t)) ∧
        (I100 (t + 2) = ¬ce t)
```

Invoking

```
cv2hol -edge -eqn COUNT2.cv CLK_DIV CLK_DIV
```

creates a theory containing

```
|- CLK_DIV (clk,ce) =
    ¬ce 0 ∧
    ∀t.
      (if ¬clk t ∧ clk (t + 1) then
         ce (t + 1) = ¬ce t
       else
         ce (t + 1) = ce t)
```

Invoking

```
cv2hol -tick -eqn COUNT2.cv CLK_DIV CLK_DIV
```

creates a theory containing

[7] Currently the clock line is retained as an input variable, but ignored. This is inefficient for model checking, so in the future the clock variable may be eliminated.

```
|- CLK_DIV (clk,ce) =
    ¬ce 0 ∧
    ∀t.
      (if clk t then ce (t + 1) = ¬ce t else ce (t + 1) = ce t)
```

and invoking

```
cv2hol -cycle -eqn CLK_DIV CLK_DIV
```

creates a theory containing

```
|- CLK_DIV (clk,ce) = ¬ce 0 ∧ ∀t. ce (t + 1) = ¬ce t
```

If several modules are specified and the `-eqn` argument given, then all the modules are unwound. For example, invoking

```
cv2hol -cycle -eqn COUNT2.cv CLK_DIV COUNT2 COUNT2
```

creates a theory `COUNT2Theory` containing the HOL netlists of `CLK_DIV` and `COUNT2` and the theorems

```
|- CLK_DIV (clk,ce) = ¬ce 0 ∧ ∀t. ce (t + 1) = ¬ce t

|- COUNT2 (clk,out_1,out_0) =
    ∃ce.
      ¬ce 0 ∧ ¬out_1 0 ∧ ¬out_0 0 ∧
      ∀t.
        (ce (t + 1) = ¬ce t) ∧
        (if ce t then
           out_1 (t + 1) = ¬(out_1 t = out_0 t)
         else
           out_1 (t + 1) = out_1 t) ∧
        (if ce t then
           out_0 (t + 1) = ¬out_0 t
         else
           out_0 (t + 1) = out_0 t)
```

6 Deriving state transition systems

The equations produced using `-eqn` are useful for theorem proving. For model checking, it is convenient to derive a state transition system. To support this `cv2hol` can automatically derive a predicate giving the initial state (which has all the variables initialised to `F`) and a transition relation $\mathcal{R}$ defined so that if s is the vector of boolean state variables and s' is the corresponding vector of primed variables then $\mathcal{R}(s, s')$ means that s' is a possible successor of s.

The state vector of the transition system for a module consists, in general, of a pair (s_1, s_2) where s_1 is a vector of the inputs and outputs of the module and s_2 is a vector of the local variables. If there are no local variables, as in `CLK_DIV`, then the state vector is just s_1. For `COUNT2` the vector s_1 is `(clk,out_1,out_0)` and s_2 is `ce`.

Invoking `cv2hol` with argument `-trans` generates definitions of <*module*>`Init` and <*module*>`Trans` for each module specified. For example, invoking

```
cv2hol -cycle -trans COUNT2.cv CLK_DIV COUNT2 COUNT2
```

puts the following definitions and theorems into `COUNT2Theory`

```
|- CLK_DIVInit(clk,ce) = ¬ce

|- CLK_DIVTrans((clk,ce),clk',ce') = (ce' = ¬ce)

|- COUNT2Init((clk,out_1,out_0),ce) = ¬ce ∧ ¬out_1 ∧ ¬out_0

|- COUNT2Trans(((clk,out_1,out_0),ce),(clk',out_1',out_0'),ce') =
    (ce' = ¬ce) ∧
    (if ce then out_1' = ¬(out_1 = out_0) else out_1' = out_1) ∧
    (if ce then out_0' = ¬out_0 else out_0' = out_0)
```

In addition to these definitions, for each module $\mathcal{M}$ a theorem is automatically proved of the form

```
|- ∀P.(∀s1 s2. Reachable MTrans MInit (s1,s2) ⇒ P s1)
      ⇒
      ∀v1···vn. M(v1, ..., vn) ⇒ ∀t. P(v1 t, ..., vn t)
```

This shows that if P is true of all reachable states of the derived transition system then P(v1 t,...,vn t) holds at each time t. This theorem is a bridge from model checking to theorem proving [1]. For example, invoking

```
cv2hol -cycle -trans COUNT2.cv CLK_DIV COUNT2 COUNT2
```

puts the following theorems into COUNT2Theory

```
|- ∀P.(∀s. Reachable CLK_DIVTrans CLK_DIVInit s ⇒ P s)
      ⇒
      ∀clk ce. CLK_DIV (clk,ce) ⇒ ∀t. P (clk t,ce t)

|- ∀P. (∀s1 s2. Reachable COUNT2Trans COUNT2Init (s1,s2) ⇒ P s1)
       ⇒
       ∀clk out_1 out_0.
         COUNT2(clk,out_1,out_0) ⇒ ∀t. P(clk t,out_1 t,out_0 t)
```

7 Discussion and future research

cv2hol packages an off-the-shelf hardware compiler (CV3) and a proof engine (HOL) into an easy-to-use turnkey semantics extractor for Verilog. The output can be loaded into other tools (including HOL) for further processing.

In the future it is hoped that the various models could be derived from a single model, ideally a representation in logic of the HDL simulation cycle. The current tool is a pragmatic compromise: the different models correspond to different definitions of the cv2.100 primitives and are not formally related. However, the implementation methodology of cv2hol does insure that if you trust the specified model, then the other derived representations (as selected by -eqn, -trans) are guaranteed to be logically consistent with it. This is a step towards ensuring the different representations needed for different purposes are consistent with each other.

The long term goal is to provide a platform supporting the easy scripting of bespoke checkers that automatically verify special purpose properties. cv2hol is an initial experiment and is part of Cambridge's contribution to the Prosper project [9].

8 Implementation status

Currently cv2hol is implemented as a Moscow ML standalone executable that parses the command line arguments, creates an ML script, and then loads the script into HOL to create the desired theory. The

dynamically created script loads one of `simTheory`, `edgeTheory`, `tickTheory` or `cycleTheory` according to the argument given to `cv2hol`. Currently the `-eqn` and `-trans` options are not available if `-sim` (i.e. `simTheory`) is specified. It is possible that this implementation strategy might change in the future (e.g. to use `Holmake` and/or the Prosper tool integration mechanisms).

9 Acknowledgements

This work was also supported by ESPRIT Framework IV LTR 26241 project entitled *Proof and Specification Assisted Design Environments* (PROSPER), EPSRC grant GR/L35973 entitled *A Hardware Compilation Workbench* and EPSRC grant GR/L74262 entitled *A uniform semantics for Verilog and VHDL suitable for both simulation and formal verification.*

References

1. Michael J. C. Gordon. Reachability programming in HOL using BDDs. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 180–197. Springer-Verlag, 2000.
2. Mike Gordon. Synthesizable verilog: syntax and semantics. Draft available at `http://www.cl.cam.ac.uk/users/mjcg/Verilog/V/V.ps.gz`.
3. Mike Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. North-Holland, 1986.
4. Mike Gordon. The semantic challenge of Verilog HDL. In *Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 136–145. IEEE Computer Society Press, 1995.
5. John Harrison. Floating point verification in HOL Light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1997. Available on the Web as `http://www.cl.cam.ac.uk/users/jrh/papers/tang.html`.
6. T. F. Melham, editor. *Higher Order Logic and Hardware Verification.* Cambridge Tracts in Theoretical Computer Science 31. Cambridge University Press, 1993.
7. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.
8. Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
9. See web page `http://www.dcs.gla.ac.uk/prosper/`.

Towards the Integration of Model Checking and Theorem Proving: Embedding a Subset of Promela into HOL

Elsa L. Gunter[1], Davor Obradovic[2]

Bell Laboratories[1],
600 Mountain Ave.,
Murray Hill, NJ, 07974, USA
email: elsa@research.bell-labs.com

University of Pennsylvania[2],
Department of Computer and Information Science,
200 South 33rd Street,
Philadelphia, PA 19104, USA
email: davor@saul.cis.upenn.edu

Abstract. We describe how to interface a model checker SPIN with a theorem prover HOL through a language called GAS. GAS programs specify concurrent transition systems with guarded assignments. The language is expressively a subset of Promela. We formally define the GAS semantics in HOL, define a semantics of LTL formulae on GAS programs and provide a translation from GAS to Promela. We use this to add SPIN as an external decision procedure to HOL for proving properties of the form "GAS program G satisfies the LTL formula φ".

1 Motivation

When doing a verification of a property of a protocol or a concurrent program, often some significant reasoning is required to reduce the problem to one that is suitable for model checking. Abstractions may need to be performed on the data space to reduce it from an infinite space to perhaps a two point space. We may need to prove a fact about a protocol involving an arbitrary number of processes, but where it is possible to argue that it suffices to show that the protocol works when there are at most, say, 3 or 5 processes. For an example of such a reduction see [2]. Once these reductions are performed, then a model checker is the most efficient way of finishing the verification. However, these reductions are outside the scope of a model checker such as SPIN, but require as much rigor as the parts that are handled by the model checker. Therefore it is often appropriate that they should should be done with an interactive theorem prover.

By having the theorem prover directly call the model checker, instead of the human verifier recoding the problem, we get higher assurance that the same program is model checked as was reasoned about in the theorem prover. Also the correctness of the translation may be examined once for all translations, instead of on a per translation basis.

We are not the first to see a need to connect model checkers to theorem provers. PVS has incorporated a BDD-based decision procedure for the relational μ-calculus for use in the verification of protocols [8, 12, 13]. Isabelle/HOL also provides a connection [11] to a model checker (the Eindhoven model checker) based on the μ-calculus. To the best of our knowledge, this is the first time such a connection has been made with SPIN.

The language GAS (for Guarded Assignments) is the bridge between the model checker SPIN and the interactive theorem prover HOL. While GAS is fairly similar to other languages, such as Unity [10], it is designed to be a close fit to the execution model of SPIN, and at the same time to be reasonable to write programs in.

2 Syntax of GAS and LTL

In this section we describe the syntax of the GAS language. GAS programs are very similar to SPIN automata, the intermediate representation used for translating Promela programs into Büchi automata.

A GAS program consists of a collection of concurrently running processes. A process is a transition system, whose nodes represent control points. We assume that there is a collection of integer variables, all of which are shared among the processes. Each transition has two control points associated with it: *origin* and *destination*. Additionally, each transition is labeled with a *guard* and a *parallel assignment*. A guard is a boolean predicate on the space of all variables. Parallel assignment can change the value of one or more variables. The formal grammar is shown in Table 1.

Table 1. GAS Grammar

$<GASProgram> ::= [<Process>, \ldots, <Process>]$

$<Process> ::= (<ControlPoint>, <TransitionSet>)$

$<TransitionSet> ::= \{<Transition>, \ldots, <Transition>\}$

$<Transition> ::= (<ControlPoint>, <Guard>, <ParAssignment>, <ControlPoint>)$

$<ControlPoint> ::= 0, 1, 2, \ldots$

$<Guard> ::= \mathsf{true} \mid \mathsf{false}$
$\mid <Guard> \vee <Guard>$
$\mid <Guard> \wedge <Guard>$
$\mid <Guard> \rightarrow <Guard>$
$\mid \neg <Guard>$
$\mid <Expr> < <Expr>$
$\mid <Expr> > <Expr>$
$\mid <Expr> \leq <Expr>$
$\mid <Expr> \geq <Expr>$
$\mid <Expr> = <Expr>$
$\mid <Expr> \neq <Expr>$

$<Expr> ::= <Var> \mid <Const>$
$\mid <Expr> + <Expr>$
$\mid <Expr> - <Expr>$

$<Var> ::= \mathsf{x}, \mathsf{y}, \mathsf{z}, \ldots$

$<Const> ::= 1, 2, 3, \ldots$

$<ParAssignment> ::= [<Assignment>, \ldots, <Assignment>]$

$<Assignment> ::= <Var> := <Expr>$

Initially, all the variables are set to zero and every process has a distinguished starting control point. At every step, only *one* process can make a move. A move consists of taking an enabled transition whose origin is the current control point of the process. More precisely, a process P currently at control point c_1 can take the transition $c_1 \xrightarrow{g;a} c_2$ if the guard g evaluates to `true`, given the current state of all the variables. As a result, P will change its current control point to c_2 and update the variables according to the parallel assignment a. Variables are updated in two steps:

1. Evaluate the right-hand sides of all the singleton assignments in a.
2. Assign those values to the corresponding variables on the left-hand side in the order from left to right.

We can represent GAS programs as transition diagrams. Figure 1 shows graphical representation of the GAS program $GP = [P_1, P_2]$, where

$$P_1 = (1,\ \{\ (1, \mathsf{x} < 2, [\mathsf{y} := \mathsf{y}+1], 2),\quad (2, \mathsf{y} \geq 3, [\mathsf{x} := \mathsf{x}+1], 1),\quad (2, \mathsf{true}, [\mathsf{y} := \mathsf{y}+1], 2)\ \})$$

$$P_2 = (1,\ \{\ (1, \mathsf{x} \geq 2, [\,], 2),\quad (2, \mathsf{y} < 3, [\mathsf{x} := \mathsf{x}-1], 1),\quad (2, \mathsf{true}, [\mathsf{y} := \mathsf{y}-1], 2)\ \}).$$

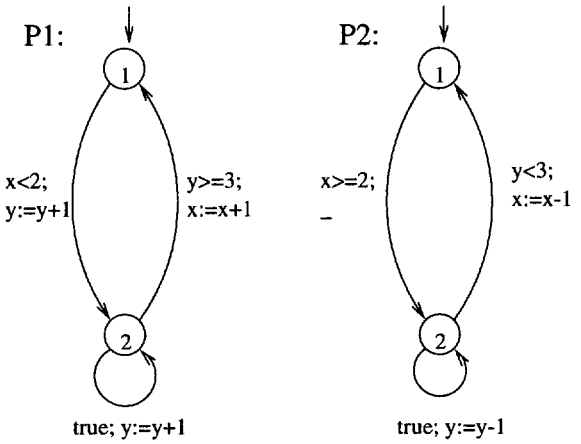

Fig. 1. Graphical representation of a sample GAS program

A sample execution fragment of the above program might go as follows. First a process is selected, say P_1. The current control point for P_1 is 1. There is one transition in P_1 with an initial control point of 1, namely the first one. Since in the initial state all variables have 0 for a value, we have $\mathsf{x} < 2$, and so the guard condition is satisfied. Therefore, the transition is enabled and we generate a new state in which y now has the value 1 and the control point for the first process is 2. The control point for the second process is still 1. In the next move, if we were to select the process P_2, no transition is enabled so no change of state would occur. If we select the process P_1 again, only the third transition is enabled. This transition increments y and retains the control point for P_1 at 2. As long as the control point for P_1 is at 2, this transition may execute an arbitrary number of times. Once it has executed enough times to cause y to be at least 3, then the second transition for P_1 also becomes enabled. Should this tranistion be chosen, then x is incremented, y is left alone, and the control point for P_1 is returned to 1. Since the value of x is now 1, we still have that only P_1 has an enabled transition, which is the first one. After the first transition of P_1 executes for a second time, both the second and third transitions of P_1 are enabled. If at this point the second transition is chosen, then x is incremented a second time and the control point for P_1 is returned to 1. This time no transition of P_1 is enabled, but finally the first transition of P_2 is enabled. This transition doesn't change the state, but sets the control point for P_2 to 2. Now only the third transition of P_2 is enabled. It decrements y and retains the control point for P_2 at 2. After it has executed enough times, eventually the second transition will also become enabled. Once the second transition of P_2 executes, the control point for P_2 is set to 1. At this point P_2 once again becomes blocked, but the first transition of P_1 is once more enabled.

Some facts that we can prove about this program in HOL include that always exactly one of P_1 or P_2 is enabled, and that the value of x is always between 0 and 3.

The main statement that we are interested in making about a GAS program is that it satisfies a given linear temporal logic formula [9]. Linear temporal logic formulae are constructed in layers. Atomic linear

temporal logic formulae are the boolean expressions over the program variables that may occur as *guard* statements in programs. These may then be combined with the standard propositional connectives, $\vee$, $\wedge$, $\neg$ and $\rightarrow$, together with the special temporal connectives $\Box$ (always), $\Diamond$ (eventually), $\bigcirc$ (next), $\mathcal{U}$ (until), and $\mathcal{W}$ (waiting, or weak until).

3 Semantics of GAS and LTL

To be able to reason about GAS programs in the theorem prover HOL, we need to give a semantic embedding of GAS programs as HOL terms. We choose to do so using a method referred to as "shallow" embedding. In a shallow embedding, constructs of the object logic are implicitly interpreted directly as logical operators in the meta-language (HOL). The alternative approach is to encode the syntax of GAS programs as datatypes of abstract syntax trees, and then give an explicit evaluation or interpretation function in the meta-language explicitly describing the meaning of the various constructs. This method is usually called "deep" embedding. Deep embedding is more general. It facilitates reasoning about the whole programming language which is being embedded, while shallow embedding allows only reasoning about the meaning (i.e. embedding) of individual programs. We choose to do a shallow embedding, because it is simpler, yet expressive enough for our purpose.

The method we use for giving the shallow embedding is to lift the various constants and operators to functions over states for guard expressions, and to functions over sequences of states for temporal formulae. This method is fairly standard, similar to the one used in [3] and [1]. We will interpret GAS programs as sets of state sequences. A state is a mapping from program variables to values, integers in our case. To begin, we view constants as constant functions over states, and variables as variable lookup (i.e. we apply the state to the variable to find its value). From here we can define expressions and guard statements by applying a lifting operation (which we shall write as a superscripted asterisk) to the operators for constructing expressions and guards. Thus, for example, the guard statement

$$(\mathsf{x} =^* 1) \wedge^* (\mathsf{y} =^* 2)$$

becomes

$$\lambda state.((state\ \mathsf{x}) = 1) \wedge (state\ \mathsf{y}) = 2).$$

The result type of a guard expression is an HOL boolean, and the result type of an expression is a value (integer). The reason for parameterizing expressions and guard statements fundamentally is to have state available for calculating variable values in assignment statements and guards. Similarly, an assignment is viewed as an update function from states to states.

Just as assignments, expressions, and guard statements are viewed as functions parameterized by states, each process is viewed as a relation parameterized by a pair of a control point and a state. To make this definition precise, we will associate with each process a transition relation defined on (control point, state)-pairs.

Let T be the set of transitions corresponding to some process, let c and d be control points and let s_1 and s_2 be states. The transition relation $\xrightarrow{T}$ describes one step of process execution. Each step changes the current control point and the current state by executing an enabled transition. The relation is formally defined by the following rule:

$$\frac{(c, g, a, d) \in T, \quad g(s_1) = \mathsf{true}, \quad a(s_1) = s_2}{(c, s_1) \xrightarrow{T} (d, s_2)}.$$

Notice that the above definition treats guards and assignments as functions parameterized by a state, as described earlier. This transition relation has a natural extension to *lists* of processes. Consider a GAS program $G = [(c_1, T_1), \ldots, (c_n, T_n)]$. Let $\overline{T} = (T_1, \ldots, T_n)$ be the vector of transition sets and $\overline{c} = (c_1, \ldots, c_n)$ be the vector of control points (one for each process), shortly called a *control vector*.

We now define the relation $\xrightarrow{\overline{T}}$ between (control vector, state)-pairs. The first rule says that the state changes when a single process executes a transition:

$$(Step) \quad \frac{(c_i, s_1) \xrightarrow{T_i} (d_i, s_2)}{((c_1, \ldots, c_i, \ldots c_n), s_1) \xrightarrow{\overline{T}} (c_1, \ldots, d_i, \ldots c_n), s_2))}.$$

There is one more rule that takes care of the situations where no transition is enabled. If this is the case, the state and the control vector stay unchanged:

$$(Block) \quad \frac{\forall i, g.\ (c_i, g, -, -) \in T_i \ \Rightarrow\ g(s) = \mathsf{false}}{(\overline{c}, s) \xrightarrow{\overline{T}} (\overline{c}, s)}.$$

This describes an asynchronous execution model, similar to that of Promela [6, 7, 5], [1] which facilitates an easy syntax-directed translation of GAS, discussed in Section 4. Another important correspondence is between semantics of the arithmetic. Instead of using the exact arithmetic already defined in HOL, we use the standard 32-bit signed word integer arithmetic that exists in Promela [6].

We now define *executions* of GAS programs as certain infinite sequences of $(\overline{c}, s)$-pairs. We say that a sequence $\{(\overline{c}^k, s_k)\}_{k \geq 0}$ of (control vector, state)-pairs is a *behavior* of a GAS program $G = [(c_1, T_1), \ldots, (c_n, T_n)]$ if adjacent elements in the sequence are connected through the transition relation:

$$\forall k \geq 0.\ (\overline{c}^k, s_k) \xrightarrow{\overline{T}} (\overline{c}^{k+1}, s_{k+1}),$$

while having $\overline{c}^0 = (c_1, \ldots, c_n)$, $\overline{T} = (T_1, \ldots, T_n)$, and s_0 being the state where all variables evaluate to zero.

By projecting behaviors to the second component (pointwise), we get *admissible state traces*. We say that a sequence of states $\{s_k\}_{k \geq 0}$ is an *admissible state trace* of G if there exists a sequence of control vectors $\{\overline{c}^k\}_{k \geq 0}$, such that $\{(\overline{c}^k, s_k)\}_{k \geq 0}$ is a behavior of G.

Finally, we define the semantics of LTL formulae on GAS programs. LTL has a natural interpretation on state traces [9], which defines when a state trace satisfies an LTL formula. We say that a GAS program G satisfies an LTL formula φ and write

$$G \models^{HOL} \varphi$$

if *every* admissible state trace of G satisfies φ.

We defined all of the above semantic relations (including both the standard and GAS semantics for LTL) in HOL. In addition, we proved soundness of the several proof rules for LTL found in [9]. This is helpful for HOL reasoning about GAS programs in the context of LTL formulae. However, our main goal is to show how to use automated tools, such as SPIN, to help us carry out the proofs. This is the topic of the next section.

4 Connecting SPIN and HOL via GAS

This section describes the way we used GAS language to connect SPIN and HOL. There are three essential parts to this process:

1. Provide a meta-function P which, given an HOL term containing a well-formed GAS program t, computes a "faithful" translation $P(t)$ of t in Promela.
2. Provide an HOL tactic `SPIN_TAC` that can be used for proving theorems of the form t satisfies φ, where t is a GAS program and φ is an LTL formula. The tactic should invoke the SPIN verifier.
3. Provide a way to incorporate the result returned by SPIN back to HOL. For this task, we use a method called *semantic tagging* [4], designed to support safe addition of external decision procedures to HOL.

[1] The only difference is synchronous communication, where a 'send' in one process can execute simultaneously with a 'receive' in another process. However, this does not appear in our translations.

4.1 Translation from GAS to Promela

For a Promela program p and an LTL formula φ, we denote by $p \models^{SPIN} \varphi$ the fact that p satisfies φ (i.e. every execution trace of p is a model of φ). The annotation is motivated by the fact that the SPIN verifier is a decision procedure for $\models^{SPIN}$. Our translation function P must have the following faithfulness property: for every GAS program t and an LTL formula φ,

$$t \models^{HOL} \varphi \text{ iff } P(t) \models^{SPIN} \varphi.$$

This property says that the translation function P preserves the LTL semantics. P is a meta-function, written in a meta-language (ML, in our case), so its faithfulness is established informally. It may seem that this is a potential source of integrity problems for HOL. However, this is not the case—one of the fundamental properties of semantic tagging is that incorrect results from the external decision procedures, while often useless, can not introduce inconsistency.

Because of the large number of technical details, we do not present the full code for our translation function P. Rather, we will concentrate on describing the essential aspects of the translation. Each GAS process will be translated into a Promela process. Control points will be translated into Promela labels and program variables will become Promela integer variables. To illustrate the translation of transitions, consider the following set of transitions with the common origin c:

$$T = \{(c \xrightarrow{g_1;a_1} c_1),\ (c \xrightarrow{g_2;a_2} c_2), \ldots,\ (c \xrightarrow{g_n;a_n} c_n)\}.$$

Guards and assignments are translated into Promela expressions. We denote this translation by $P[\ ]$. Using this convention, the following Promela fragment is a translation of T:

```
c:
if
  :: P[g_1] -> d_step { P[a_1] } goto c_1;
  :: P[g_2] -> d_step { P[a_2] } goto c_2;
        . . .
  :: P[g_n] -> d_step { P[a_n] } goto c_n;
fi;
```

The fact that GAS is shallowly embedded in HOL makes the task of translating from GAS to Promela somewhat tricky. In HOL, we operate only on semantic GAS objects. We do not have access to the original syntactic objects whose embedding created their HOL representations. For instance, it is not quite clear how to tell if a given HOL term is a representation of a well-formed GAS program, let alone to translate it into Promela. We solve this problem by relying on the specific syntax of HOL terms that represent GAS programs. In order to do that, we define, for every GAS construct, the corresponding HOL constant that encodes its meaning. Only HOL terms which correctly use those constants (and nothing else) to represent GAS programs are considered valid for the purpose of translation. Obviously, this can be established only by looking at HOL terms from the meta-level, since HOL itself can not distinguish between the terms with different syntax, but identical meaning in the logic (e.g. $\lambda x.\lambda y.\ x + y$ and $\lambda x.\lambda y.\ y + x$). Essentially, translation as our major goal forces us to perform a "syntax-preserving" embedding, which is a special, "deeper" case of shallow embedding.

4.2 SPIN as an External Decision Procedure for HOL

The general form of an HOL theorem is

$$[as_1, \ldots, as_n] \vdash s,$$

where $as_1, \ldots, as_n$ are the assumptions and s is the statement of the theorem. Such a theorem can be established by providing a proof or by deriving it from already existing theorems by using built-in

inference rules, such as Modus Ponens. Both methods are safe, in the sense that they can not generate false theorems. However, accepting external results into a theorem prover has the obvious problem concerning preservation of integrity. The theorem prover does not know whether the result is correct or not. If it does, there is no point in using an external decision procedure in the first place. Semantic tagging solves this problem by tagging every externally produced result by a special tag. Every theorem which is subsequently proved using the external result will have this tagged result among its assumption. The only way to eliminate tagged results from the assumption list is to actually prove them.

We designed an HOL tactic called `SPIN_TAC`, which can be applied to HOL goals whose statement is of the form t satisfies φ, for a GAS program t and an LTL formula φ. The tactic first computes the Promela program $P(t)$, then invokes the SPIN LTL verifier with inputs $P(t)$ and φ. The tactic succeeds if all of the following conditions hold:

1. The goal is of the required form.
2. Term t is a well-formed GAS program.
3. φ is an LTL formula not containing the next operator ($\bigcirc$). [2]
4. SPIN completes the verification and finds no counterexamples.

If all this holds, HOL generates a theorem with the tagged assumption t satisfies φ added to the assumption list. In principle, the tagged assumption can later be eliminated, using the HOL semantics for GAS and LTL described in Section 3.

5 Example

This section describes a GAS program that implements Peterson's Mutual Exclusion Protocol. Graphical version of the program is shown on Figure 2. Control points and variables are given descriptive names for the sake of clarity.

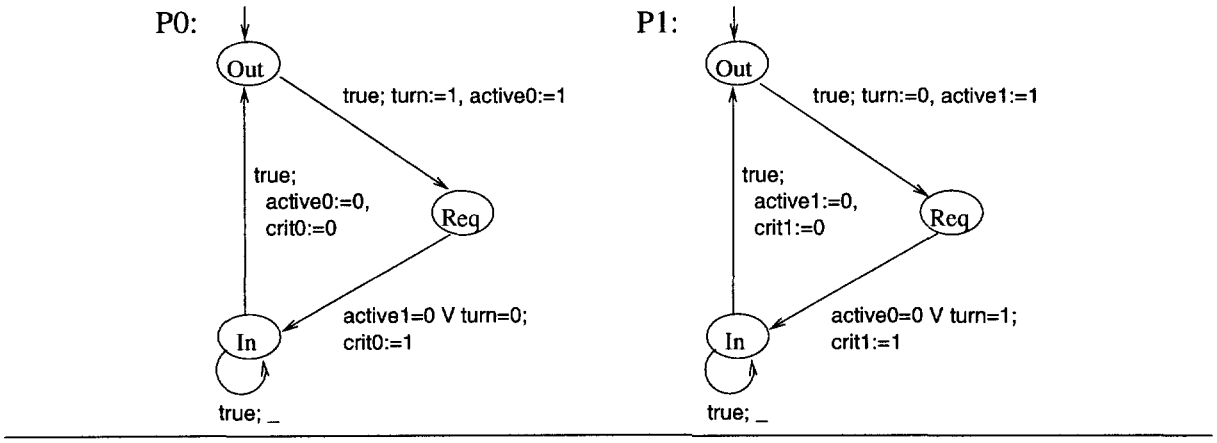

Fig. 2. Peterson's Mutual Exclusion Protocol in GAS

Processes $P0$ and $P1$ are trying to enter a shared critical region. If a process is outside of the critical region and wants to get in, it first goes through the request (Req) state. Flag `active0` (resp. `active1`) is set to 1 if $P0$ (resp. $P1$) is requesting to enter the critical region or is already inside the critical region.

[2] This is to ensure stutter-closedness of the formula, which enables SPIN to use powerful optimizations.

Variable turn is supposed to prevent the situation in which both processes are in the critical region at the same time. Flags crit0 (for $P0$) and crit1 (for $P1$) denote whether the corresponding process is inside the critical region.

GAS code for the protocol is given below:

$$P0 = (0,\ \{\ (0, \mathsf{true}, [\mathsf{x4} := 1, \mathsf{x0} := 1], 1),\\ (1, \mathsf{x1} = 0 \vee \mathsf{x4} = 0, [\mathsf{x2} := 1], 2),\\ (2, \mathsf{true}, [\,], 2),\\ (2, \mathsf{true}, [\mathsf{x0} := 0, \mathsf{x2} := 0], 0)\ \ \})$$

$$P1 = (0,\ \{\ (0, \mathsf{true}, [\mathsf{x4} := 0, \mathsf{x1} := 1], 1),\\ (1, \mathsf{x0} = 0 \vee \mathsf{x4} = 1, [\mathsf{x3} := 1], 2),\\ (2, \mathsf{true}, [\,], 2),\\ (2, \mathsf{true}, [\mathsf{x1} := 0, \mathsf{x3} := 0], 0)\ \ \}).$$

Variables and control points in the textual version of the code are renamed, since in HOL we represent them with indices. In the code above, a variable with index i is denoted as x_i. Indices are assigned according to the following table:

Variable / Control Point	active0	active1	crit0	crit1	turn	Out	Req	In
Index	0	1	2	3	4	0	1	2

Since GAS is shallowly embedded in HOL, the above GAS program is represented as an HOL term, shown in Table 2. The term is a pair consisting of the initial state and process list. The initial state assigns constant 0 to every variable. The process list contains two processes, each represented as the initial control point paired with the set of transitions. Initial control points for both processes are 0 (corresponding to Out in Figure 2). Transitions are encoded using HOL-defined constructs that closely follow the GAS syntax. For instance, functions for reading and assigning variable values are `read` and `assign`, constant representing the empty assignment is `I`, boolean disjunction operator for guards is `||`, and so on.

Table 2. Peterson's Mutual Exclusion Protocol: GAS represented in HOL

```
val Peterson =
``(\x. INT 0),
  [0,
   {0,True,assign [Var 4,con (INT 1); Var 0,con (INT 1)],1;

    1,read (Var 1) eq con (INT 0) || read (Var 4) eq con (INT 0),
    assign [Var 2,con (INT 1)],2;

    2,True,I,2;
    2,True,assign [Var 0,con (INT 0); Var 2,con (INT 0)],0};

   0,
   {0,True,assign [Var 4,con (INT 0); Var 1,con (INT 1)],1;

    1,read (Var 1) eq con (INT 0) || read (Var 4) eq con (INT 1),
    assign [Var 3,con (INT 1)],2;

    2,True,I,2;
    2,True,assign [Var 1,con (INT 0); Var 3,con (INT 0)],0}]``
: term
```

Table 3 shows the Promela code generated for this GAS program. Notice that there are two declared variables for every index i. One of them, v_i, is used to store the value of the right-hand side of the assignment, while the final assignment is performed on r_i. This is to ensure that individual assignments inside a parallel assignment do not interfere (i.e. to ensure that the assignment is indeed parallel).

Consider the three LTL properties given in Table 4. Property *MutEx* states that processes are never in the critical region at the same time. When we invoke SPIN_TAC on the HOL goal $Peterson \models^{HOL} MutEx$, the system invokes SPIN which successfully verifies the claim.

Table 3. Peterson's Mutual Exclusion Protocol: generated Promela translation

```
int v4, r4,v0, r0,v1, r1,v2, r2,v3, r3;
proctype process_0 ()
{
goto label_0;
label_0: if
::true-> d_step {v4 = 1;v0 = 1;r4 = v4;r0 = v0;} goto label_1;
fi;
label_1: if
::((r1==0)||(r4==0))-> d_step {v2 = 1;r2 = v2;} goto label_2;
fi;
label_2: if
::true-> d_step {skip;} goto label_2;
::true-> d_step {v0 = 0;v2 = 0;r0 = v0;r2 = v2;} goto label_0;
fi;
}

proctype process_1 ()
{
goto label_0;
label_0: if
::true-> d_step {v4 = 0;v1 = 1;r4 = v4;r1 = v1;} goto label_1;
fi;
label_1: if
::((r1==0)||(r4==1))-> d_step {v3 = 1;r3 = v3;} goto label_2;
fi;
label_2: if
::true-> d_step {skip;} goto label_2;
::true-> d_step {v1 = 0;v3 = 0;r1 = v1;r3 = v3;} goto label_0;
fi;
}

init
{
run process_0 ();
run process_1 ();
}
```

Table 4. LTL properties of the protocol

Name	LTL property	LTL property with Promela variables
MutEx	$\Box\neg(\text{crit0} \wedge \text{crit1})$,	$\Box\neg(\text{r0} = 1 \wedge \text{r3} = 1)$
Progress	$\Box(\text{active0} \Rightarrow \Diamond\text{crit0})$	$\Box(\text{r0} = 1 \Rightarrow \Diamond(\text{r2} = 1))$
CondProgress	$\Box\Diamond\neg\text{crit1} \Rightarrow$ *Progress*	$\Box\Diamond(\text{r3} = 0) \Rightarrow \Box(\text{r0} = 1 \Rightarrow \Diamond(\text{r2} = 1))$

However, the tactic fails if we attempt to prove *Progress*, which states that if $P0$ requests permission to enter the critical region, it would eventually get it. This is because the property is violated if $P1$ decides to stay inside the critical region indefinitely.

CondProgress states that this is exactly the only case when *Progress* is violated. Precisely, it states that *Progress* for $P0$ is guaranteed as long as we know that $P1$ is out of the critical region infinitely often. `SPIN_TAC` managed to prove this goal as well.

6 Conclusions and Future Work

We have demonstrated a way to interface a SPIN model checker with the interactive theorem prover HOL through a common sub-language called GAS. GAS has the four essential properties:

1. It is expressive enough to write interesting and nontrivial programs in it.
2. It can be straightforwardly compiled into Promela.
3. It has a trace semantics which can be subjected to LTL verification.
4. Its semantics can be relatively easily encoded into HOL.

SPIN can be invoked from HOL to verify a conjectured satisfaction of an LTL formula by a GAS program.

If a theorem to be proved in HOL can be reduced to some finite state transition system, our framework enables an appropriate split of the work between HOL and SPIN—HOL can be used to prove the reduction and SPIN can be used to finish a potentially tedious case analysis. Communication between HOL and SPIN is automatic and safe, in the sense that it does not compromise the integrity of the verification.

Looking at the expressive power, GAS is a subset of Promela. It is a proper subset, meaning that there are Promela programs that can not be naturally represented in GAS. GAS was designed primarily as an experimental language and as such it does not support many of Promela's features. Some of those features can be simulated. An example is asynchronous channel communication, which can be simulated by using shared variables. Others, such as dynamic process creation, can not be easily simulated. A possible future work direction is extension of GAS with more Promela constructs.

Another idea is to consider extensions of the LTL semantics for GAS which would incorporate notions of weak and strong fairness. For instance, in some cases we may want to ignore the behaviors in which a particular transition stays enabled indefinitely, but is never taken.

Finally, interesting extensions are possible concerning the HOL/SPIN interaction. The current system accepts only positive verification answers from SPIN. In cases where verification fails due to a counterexample found by SPIN, a proof of the negation of the original conjecture could be automatically generated from the counterexample.

References

1. F. Andersen, U. Binau, K. Nyblad, K. D. Petersen, , and J. S. Pettersson. The HOL-UNITY verification system. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development: 6th International Conference: Proceedings*, volume 915 of *Lecture Notes in Computer Science*, pages 795–796. Springer-Verlag, 1995.
2. Karthikeyan Bhargavan, Carl A. Gunter, and Davor Obradovic. RIP in SPIN/HOL, August 2000. To appear in: 13th International Conference on Theorem Proving in Higher-Order Logics TPHOLs 2000.
3. M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer-Verlag, 1989.
4. Elsa L. Gunter. Adding external decision procedures to HOL90 securely. In *Theorem Proving in Higher Order Logics, 11th International Conference TPHOLs '98*, volume 1479. Springer-Verlag LNCS, September 1998.

5. Gerard J. Holzmann. SPIN-formal verification. Web Page. Available at http://netlib.bell-labs.com/netlib/spin/whatispin.html.
6. Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991. http://cm.bell-labs.com/cm/cs/what/spin/Doc/Book91.html.
7. Gerard J. Holzmann. The Spin Model Checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
8. Patrick Lincoln and John Rushby. The formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304, Elounda, Greece, June/July 1993. Springer Verlag.
9. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1991.
10. J. Misra and K.M. Chandy. *Parallel Program Design:A Foundation.* Addison-Wesley, 1988.
11. Olaf Müller, Jan Philipps, and Robert Sandner. Invoking model checkers in isabelle/hol. Web Page. Available at http://isabelle.in.tum.de/dist/Isabelle99/src/HOL/Modelcheck/.
12. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
13. John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. In Mario Dal Cin, Catherine Meadows, and William H. Sanders, editors, *Dependable Computing for Critical Applications—6*, volume 11 of *Dependable Computing and Fault Tolerant Systems*, pages 203–222, Garmisch-Partenkirchen, Germany, March 1997. IEEE Computer Society.

Congruence Classes with Logic Variables
Category B Paper

Joe Hurd*

Computer Laboratory
University of Cambridge
joe.hurd@cl.cam.ac.uk

Abstract. We are improving equality reasoning in automatic theorem-provers, and congruence classes provide an efficient storage mechanism for terms, as well as the congruence closure decision procedure. We describe the technical steps involved in integrating logic variables with congruence classes, and present an algorithm that can be proved to find all matches between classes (modulo certain equalities). An application of this algorithm makes possible a *percolation* algorithm for undirected rewriting in minimal space; this is described and an implementation in `hol98` is examined in some detail.

1 Introduction

There has been a long-standing difficulty in theorem-proving of blending together deduction steps (such as Modus-Ponens or specialization) with equality steps (Leibniz' rule of substituting equals for equals). On the equality side, there are useful rewriters in most interactive theorem-provers: these are able to call decision procedures and perform 'simplifying' deductive steps. On the deduction side, there are powerful resolution theorem-provers such as Gandalf [13]: these include special deductive rules for handling equality but still this is where they are weak in practice.

We are interested in combining these two worlds, and in this paper we present a method to improve the handling of equality in deductive provers, using congruence classes. To motivate this work, consider the following example from group theory that a deductive prover would find difficult to handle:

$$\begin{aligned} &\{\forall x\, y\, z.\ [(x * y) * z = x * (y * z)] \wedge [e * x = x] \wedge [i(x) * x = e]\} \\ \Rightarrow\quad &\{\forall x.\ x * i(x) = e\} \end{aligned}$$

Here the antecedent is enough to axiomatize the group operation ($*$), identity (e) and inverse (i), and the consequent is the right inverse group law. Here is a proof in the form of a chain of equalities [2]:

$$\begin{aligned} e &= i(x * i(x)) * (x * i(x)) \\ &= i(x * i(x)) * (x * (e * i(x))) \\ &= i(x * i(x)) * (x * ((i(x) * x) * i(x))) \\ &= i(x * i(x)) * ((x * (i(x) * x)) * i(x)) \\ &= i(x * i(x)) * (((x * i(x)) * x) * i(x)) \\ &= i(x * i(x)) * ((x * i(x)) * (x * i(x))) \\ &= (i(x * i(x)) * (x * i(x))) * (x * i(x)) \\ &= e * (x * i(x)) \\ &= x * i(x) \end{aligned}$$

* Supported by an EPSRC studentship

The problem is that the chain is long, and to find the proof a deductive prover would have to generate a very large number of terms using the given equalities. Note that this example would also pose a problem for a basic rewriter because the rewrite set requires completion to solve the problem.

Using congruence classes to store a set of terms maximizes subterm sharing and performs the congruence closure decision procedure, thus reducing the memory and time requirements of the deductive prover. There is however an incompatibility that needs addressing: deductive proving relies on logic variables that can be instantiated to arbitrary terms during the proof search, while congruence classes treat all variables in the terms as constants (and hence congruence closure does also).

The contribution made in this paper is an exploration of some of the ramifications of using congruence classes to store terms with logic variables, and a demonstration of how the underlying data structure may be exploited to find all possible term matches modulo certain equalities. An immediate application of this is a *percolation* algorithm that performs undirected rewriting on congruence classes; we also describe this and characterize what it can and cannot prove. As a secondary contribution, the algorithms presented in this paper have all been implemented in the `hol98` theorem-prover, and some preliminary results with the percolation algorithm are examined.

The structure of the paper is as follows: Section 2 defines a set of congruence classes with the basic operations, Section 3 is on matching: we precisely define the problem, present the match-finding algorithm and prove that it always terminates with all matches. We describe the percolation algorithm in Section 4, and in Section 5 we briefly examine the functionality and performance of the HOL implementation. We draw some conclusions from the work in Section 6, and finally in Section 7 relate our approach to others. Appendix A contains the details of the HOL implementation.

Note: this paper has been submitted to the Journal of the IGPL.

1.1 Notation

Terms in HOL are either constants, variables, function applications $app(t_1, t_2)$ of terms to terms, or lambda abstractions $lambda(v, t)$ of variables from terms.

2 Congruence Classes

2.1 Introduction

A congruence class set is a data structure that allows a set of terms to maximize the sharing of their subterms (and hence stores them in the minimum possible space). This property provides the basis of the congruence closure decision procedure [1] for the theory $\mathcal{E}$ in which the only interpreted function symbols are $=$ and $\neq$.

We define a **congruence class** to be a set of terms and a representative, so we can write a set of classes like this:

$\{\ rep_1\ \{\ elt_{(1,1)},\ elt_{(1,2)},\ \ldots,\ elt_{(1,m_1)}\ \},$
$\quad rep_2\ \{\ elt_{(2,1)},\ elt_{(2,2)},\ \ldots,\ elt_{(2,m_2)}\ \},$
$\quad \ldots$
$\quad rep_n\ \{\ elt_{(n,1)},\ elt_{(n,2)},\ \ldots,\ elt_{(n,m_n)}\ \}\ \}$

This class set $\mathcal{C}$ has n classes; we write C_i for the ith class, which here has m_i elements. The set of elements of a class can never be empty; the representative is always an element. We also insist that there is a distinguished **true class** containing $\top$ and a **false class** containing $\bot$. One more piece of notation: given an element elt in class C with representative rep, we will sometimes write $class(elt)$ for C and $[elt]$ for rep.

2.2 Normal Form

A term t is in **normal form** with respect to a congruence class set $\mathcal{C}$ if t is an element of a class in $\mathcal{C}$, and all the proper subterms of t are representatives of classes in $\mathcal{C}$. A congruence class set is in **normal form** if all its elements are in normal form.

In practice it is convenient to keep the class set $\mathcal{C}$ in normal form at all times. This is not a burden; the empty class set is already in normal form, and we add an arbitrary term t to $\mathcal{C}$ by the following procedure:

1. We may assume by recursion on the term structure that the immediate subterms of t are equal to terms in normal form.
2. Thus since the immediate subterms are equal to elements of $\mathcal{C}$ we may rewrite them to be their class representatives, to get t'.
3. If t' is an element of $\mathcal{C}$ then we are done, otherwise we add the new class $t'\{t'\}$.

This procedure adds t to the class set $\mathcal{C}$, and as a side effect produces the theorem $t = t'$ where t' is in normal form.

Henceforth we will assume that congruence class sets are always in normal form, unless we explicitly state otherwise.

2.3 Congruence Closure

We define a **close** operation on a class set $\mathcal{C}$: whenever two classes C_i and C_j in $\mathcal{C}$ have an element in common, then merge the two classes using the following algorithm:

1. First choose rep to be the smaller term of rep_i and rep_j.[1]
2. Replace C_i and C_j in $\mathcal{C}$ with a new class C with representative rep and which contains all of the elements of C_i and C_j.
3. Finally we must rewrite all occurrences (in proper subterms) of rep_i and rep_j within $\mathcal{C}$ to rep.

So far the effect of this operation is the same as the standard congruence closure decision procedure, but we also need two more features in our close operation:

- Look for terms of the form $A = B$ in the true class, where A is not the same term as B, and merge the classes with representatives A and B.
- Look for terms of the form $A = A$ in all classes except the true class, and merge all the classes where we find such terms with the true class.

We now present a small example of congruence closure in action:[2]

1. Start with a minimal set of classes:
   $\{ \top \{ \top \},$
   $\quad \bot \{ \bot \} \}$
2. Add the fact $f(f(f(a))) = a$ to the true class:
   $\{ \top \quad \{ \top, \quad a = a \quad \},$
   $\quad \bot \quad \{ \bot \quad \},$
   $\quad f \quad \{ f \quad \},$
   $\quad a \quad \{ a, \quad f(f(f(a))) \},$
   $\quad f(a) \quad \{ f(a) \quad \},$
   $\quad f(f(a)) \{ f(f(a)) \quad \} \}$

[1] Unfortunately, this is not quite enough to make our matching algorithm complete. In addition, for every logic variable X in rep that is not in both rep_i and rep_j, we must instantiate X in rep to an arbitrary value. This will always be possible, since if X is in rep_i but not rep_j, say, since the classes contain a common element we must have that $\forall x.\ rep_i[x/X] = rep_j$, so $rep_i[arb/X] = rep_j$ and $\forall x.\ rep_i[x/X] = rep_i[arb/X]$.

[2] The same example is also in the appendix with more implementational details.

3. Now when we add the fact $f(f(f(f(f(a))))) = a$, we get a neat collapse:
   $\{ \top \{ \top, a = a \},$
   $\quad \bot \{ \bot \quad \},$
   $\quad f \{ f \quad \},$
   $\quad a \{ a,\ f(a) \} \}$

There is nothing novel so far; the congruence closure decision procedure has already been implemented in HOL by Boulton [4] using term graphs, as part of his implementation of the Nelson-Oppen combination of decision procedures [11, 12]. We preferred the current data structure because it seemed easier to integrate logic variables and perform our matching algorithm on HOL terms, where we could use the standard HOL functions for operating on them. However, the underlying closure algorithm (using Union/Find) is the same in both cases, and both implementations perform a fully expansive proof.

2.4 Terms with Logic Variables

We assume that our terms contain logic variables from a set V. To avoid confusion between the two types of variable, we will always write logic variables in upper case and normal 'HOL' variables in lower case. Thus the term $f(X)$ contains a logic variable but the term $f(x)$ does not.

We store terms in a congruence class set treating logic variables as normal variables. This has the effect that t_1 and t_2 will be in the same congruence class if and only if for every variable instantiation σ we have $\sigma t_1 = \sigma t_2$.

We are now in a position to define **equality modulo** $\mathcal{C}$ for a class set $\mathcal{C}$, which we write $=_{\mathcal{C}}$. If $t_1 =_{\mathcal{C}} t_2$, then t_1 can be transformed to t_2 by a sequence of rewriting operations which always replace an element of $\mathcal{C}$ with another element in the same class, *treating logic variables as constants.*

To illustrate this central definition, if we have a class in $\mathcal{C}$ containing both X and $X+0$, then we can certainly say that $X =_{\mathcal{C}} X+0$ and $7+((X+0)+0) =_{\mathcal{C}} 7+X$, but not that $5+0 =_{\mathcal{C}} 5$ or that $Y+0 =_{\mathcal{C}} Y$.

We also define $t \in_{\mathcal{C}} C_i$ to mean $t =_{\mathcal{C}} rep_i$.

Note that $=_{\mathcal{C}}$ is an equivalence relation, and if $\mathcal{C}$ is closed then for all t there will be at most one C_i such that $t \in_{\mathcal{C}} C_i$.

3 Matching Algorithm

Suppose we have a congruence class set $\mathcal{C}$ that is closed and in normal form. The precise version of the matching problem that we would like to solve is: given classes C_i and C_j, what variable instantiations σ allow C_i to match to C_j? A particularly important example of this problem for deductive proving occurs when C_i contains the goal term and C_j is the true class, when the question is equivalent to asking what instantiations of variables in the goal term make it true. First we define these terms, and then present a solution.

3.1 Definitions

We define a **variable instantiation** $\sigma : V \to \mathcal{T}$ to be a function from logic variables to terms (which may themselves contain logic variables). Given a term $t \in \mathcal{T}$, we write σt to mean the instantiation of variables in t according to σ.

For a congruence class set $\mathcal{C}$ containing classes C_i and C_j, we say that σ **allows** $\mathbf{C}_i$ **to match to** $\mathbf{C}_j$ if

$$\exists\, t_i\, t_j.\ t_i \in_{\mathcal{C}} C_i \ \wedge\ t_j \in_{\mathcal{C}} C_j \ \wedge\ \sigma t_i = t_j$$

We define a function $\chi : V \times \mathcal{C} \to \mathcal{P}(V \to \mathcal{T})$ from variable-class pairs to sets of variable instantiations

$$\chi(X, C) = \{\sigma : V \to \mathcal{T} \mid \sigma(X) \in_{\mathcal{C}} C\}$$

and let ϕ be

$$\phi(S) = \bigcap_{(X,C)\in S} \chi(X,C)$$

So $\phi(S)$ is the set of all functions from V to $\mathcal{T}$ that, for every (X,C) pair in S, map variable X to a term 'provably in C'. We call sets of variable-class pairs **substitutions**.

3.2 The Algorithm

The reason that we defined the function ϕ in the previous subsection is that for every pair (C_i, C_j) of classes the algorithm we present finds substitutions S such that every variable instantiation in $\phi(S)$ allows C_i to match to C_j. Let $match(C_i, C_j)$ be the current set of substitutions between (C_i, C_j).

We initialize each $match(C_i, C_j)$ as follows:

1. First set $match(C_i, C_j) = \{\}$.
2. If rep_i and rep_j have different HOL types, then finish.
3. For every logic variable X with $class(X) = C_i$, add $\{(X, C_j)\}$.
4. If $i = j$ then add $\{(X, class(X)) \mid X \text{ a logic variable in } rep_i\}$.

And now we inductively build up *match*, terminating when it is no longer possible to add anything new to any $match(C_i, C_j)$:

- Given an element in class C_j of the form $lambda(v, rep_b)$ and a substitution $S \in match(C_B, C_b)$; if there exists an element $lambda(v, rep_B)$ in some class C_i then add S to $match(C_i, C_j)$.
- Given an element in class C_j of the form $app(rep_f, rep_a)$, and substitutions $S \in match(C_F, C_f)$ and $S' \in match(C_A, C_a)$; if both
  - S and S' are **compatible**, (i.e., for every variable X there is at most one element in $S \cup S'$ of the form (X, C)).
  - there exists an element $app(rep_F, rep_A)$ in some class C_i

  then add $S \cup S'$ to $match(C_i, C_j)$.

Note that here we are using the fact that $\mathcal{C}$ is in normal form.

We will illustrate the algorithm on an example. Unfortunately, all real-life examples are too large to represent, so the example will necessarily have to be rather artificial. We invent an 'if-and-only-if' operator, f, which return true exactly when its two boolean arguments are equal. The example shows what matches occur on the term $f\ \top\ (f\ \top\ \top)$.

1. Here is the result of adding the fact $f\ X\ X$ and the term $f\ \top\ (f\ \top\ \top)$ to a minimal class set:

$\{\top$	$\{\top,$	$f\ X\ X\},$
$\bot$	$\{\bot$	$\},$
f	$\{f$	$\},$
X	$\{X$	$\},$
$f\ X$	$\{f\ X$	$\},$
$f\ \top$	$\{f\ \top$	$\},$
$f\ \top\ \top$	$\{f\ \top\ \top$	$\},$
$f\ \top\ (f\ \top\ \top)$	$\{f\ \top\ (f\ \top\ \top)$	$\}\ \}$

2. The initial matches are easy to calculate, every class matches itself, and the boolean logic variable X matches all the boolean classes:

$$
\begin{aligned}
match(\top,\ \top) &= \{\{\}\} \\
match(\bot,\ \bot) &= \{\{\}\} \\
match(f,\ f) &= \{\{\}\} \\
match(X,\ X) &= \{\{(X,\ X)\}\} \\
match(f\ X,\ f\ X) &= \{\{(X,\ X)\}\} \\
match(f\ \top,\ f\ \top) &= \{\{\}\} \\
match(f\ \top\ \top,\ f\ \top\ \top) &= \{\{\}\} \\
match(f\ \top\ (f\ \top\ \top),\ f\ \top\ (f\ \top\ \top)) &= \{\{\}\} \\
match(X,\ \top) &= \{\{(X,\ \top)\}\} \\
match(X,\ \bot) &= \{\{(X,\ \bot)\}\} \\
match(X,\ f\ \top\ \top) &= \{\{(X,\ f\ \top\ \top)\}\} \\
match(X,\ f\ \top\ (f\ \top\ \top)) &= \{\{(X,\ f\ \top\ (f\ \top\ \top))\}\}
\end{aligned}
$$

All other *match* sets are empty.

3. After one inductive step, we gain the following extra match:
   $match(f\ X,\ f\ \top) = \{\{(X,\ \top)\}\}$
   This comes from the application inductive step, using the compatible substitutions $\{\}$ and $\{(X, \top)\}$, contained in $match(f, f)$ and $match(X, \top)$ respectively.
4. After two inductive steps, we gain the following extra match:
   $match(\top,\ f\ \top\ \top) = \{\{(X,\ \top)\}\}$
   Again from the application inductive step: this uses the substitution from the previous step, and the compatible substitution $\{(X, \top)\}$ contained in $match(X, \top)$. Since $f\ X\ X$ has representative $\top$, this is how the match appears. Notice that there are no matches from $\top$ to $f\ \top\ (f\ \top\ \top)$, because the relevant substitutions are incompatible.
5. The algorithm now terminates, because no more substitutions can be added to any *match* set. All the match sets are returned to the calling application.

We will return to this example in Section 4, to show how the percolation algorithm makes use of the returned match sets.

There are two significant optimizations that can be made to the theoretical algorithm. Firstly, we do not add a substitution S that is less general than a match S' that we already have (i.e., $\phi(S) \subseteq \phi(S')$), though the algorithm above adds anything that is different. This costs nothing and promotes faster convergence.

Secondly, we divide the inductive stage into passes, so that on pass $n + 1$ we add matches that arise from the result of pass n. This allows us to optimize by considering only the matches that were new at pass n, instead of every match. This is most significant for the application inductive step, where if there are m new and M old matches at pass n, it only has to consider $2mM + m^2$ combinations instead of $(m + M)^2$.

3.3 Proofs

We will prove three theorems about the algorithm: firstly, it terminates on all inputs; secondly, a soundness result that every match that it finds is valid; thirdly, a completeness result that every possible match is found. The purpose of these theorems is to show that the theoretical algorithm is logically sound[3] and also to provide some useful information on the scope of the ideas. They can be used in proving facts about procedures that make use of the Matching Algorithm: we will see in Section 4 that the Percolation Algorithm relies on the termination and completeness results.

Theorem 1 (Termination). *The matching algorithm terminates on all inputs.*

[3] Though in `hol98`, the design of the theorem-prover forces all inferences to be expressed as combinations of primitive inferences and these are checked by the logical kernel, so in this environment we are protected against proving false theorems.

Proof. Let c be the number of classes in $\mathcal{C}$, and let v be the number of logic variables in $\mathcal{C}$. Note that the algorithm does not create any new classes or logic variables, so c and v are fixed.

The number of variable-class pairs that may arise in the matching algorithm is bounded above by vc, and so the number of possible sets of variable-class pairs, i.e., substitutions, is bounded above by 2^{vc}.

Define the termination measure to be the sum of the number of substitutions in all the match sets. Since there are c^2 match sets, this is bounded above by $c^2 2^{vc}$.

The algorithm terminates when it cannot add any new substitutions to any match set, so the termination measure must increase on every inductive step. But it is bounded above, so the algorithm must terminate.

Lemma 1. *For all substitutions S and S', we have that $\phi(S \cup S') = \phi(S) \cap \phi(S')$.*

Proof.

$$\begin{aligned}\phi(S \cup S') &= \bigcap_{(X,C) \in S \cup S'} \chi(X, C) \\ &= \left(\bigcap_{(X,C) \in S} \chi(X, C) \right) \cap \left(\bigcap_{(X,C) \in S'} \chi(X, C) \right) \\ &= \phi(S) \cap \phi(S')\end{aligned}$$

Theorem 2 (Soundness). *For every substitution $S \in match(C_i, C_j)$, for every variable instantiation $\sigma \in \phi(S)$, we have that σ allows C_i to match to C_j.*

Proof. We proceed by structural induction on match, and assume that we have just added substitution S to $match(C_i, C_j)$, and that $\sigma \in \phi(S)$.

Firstly suppose S is one of the initial substitutions; for step 3 it is clear from the definitions that $X \in_{\mathcal{C}} C_i$ and $\sigma(X) \in_{\mathcal{C}} C_j$, similarly for step 4 we have that $rep_i \in_{\mathcal{C}} C_i$ and $\sigma rep_i \in_{\mathcal{C}} C_i$.

Now consider the lambda inductive step. By the induction hypothesis there must exist $t_B \in_{\mathcal{C}} C_B$ and $t_b \in_{\mathcal{C}} C_b$ with $\sigma t_B = t_b$, so therefore we have that $lambda(v, t_B) \in_{\mathcal{C}} C_i$, $lambda(v, t_b) \in_{\mathcal{C}} C_j$ and $\sigma lambda(v, t_B) = lambda(v, t_b)$.

Finally consider the application inductive step. There must exist classes C_F, C_f, C_A and C_a such that $app(rep_F, rep_A)$ is an element in C_i and $app(rep_f, rep_a)$ is an element in C_j. In addition, there must exist compatible substitutions $S_f \in match(C_F, C_f)$ and $S_a \in match(C_A, C_a)$ such that $S = S_f \cup S_a$, and so by Lemma 1 we know $\sigma \in \phi(S_f)$ and $\sigma \in \phi(S_a)$. Now we may apply the induction hypothesis to get $t_F \in_{\mathcal{C}} C_F$, $t_f \in_{\mathcal{C}} C_f$, $t_A \in_{\mathcal{C}} C_A$, $t_a \in_{\mathcal{C}} C_a$ with $\sigma t_F = t_f$ and $\sigma t_A = t_a$. Therefore we have that $app(t_F, t_A) \in_{\mathcal{C}} C_i$, $app(t_f, t_a) \in_{\mathcal{C}} C_j$, and $\sigma\, app(t_F, t_A) = app(t_f, t_a)$, as required.

Lemma 2. *For every class C_i, all the logic variables in rep_i are also in every element of C_i.*

Proof. This is true when the class initializes, and the case when two classes merge is taken care of in the footnote to Subsection 2.3.

Lemma 3. *For every $t \in_{\mathcal{C}} C$, there is an element x in C such that one of the following holds:*

- *t is a logic variable, constant or normal variable and $t = x$.*
- *$t = lambda(v, t_b)$, $x = lambda(v, rep_b)$ and $t_b \in_{\mathcal{C}} C_b$.*
- *$t = app(t_f, t_a)$, $x = app(rep_f, rep_a)$, $t_f \in_{\mathcal{C}} C_f$ and $t_a \in_{\mathcal{C}} C_a$.*

Proof. Consider a counter-example t with minimal term depth. $t =_{\mathcal{C}} rep$, so consider the first time we meet an element x in C in the rewriting chain between t and rep. If x is a logic variable, constant or normal variable then we must have that $t = x$ because there can be no previous step in the chain. If x is a lambda or application term, then it must always have been so, and now we can appeal to the minimality of t to show that the proper subterms must satisfy the necessary conditions.

Theorem 3 (Completeness). *For every variable instantiation σ that allows C_i to match to C_j, there exists a substitution $S \in match(C_i, C_j)$ such that $\sigma \in \phi(S)$.*

Proof. Assume the result is false and pick a counterexample σ. There must exist $t_i \in_{\mathcal{C}} C_i$ and $t_j \in_{\mathcal{C}} C_j$ with $\sigma t_i = t_j$, and we may assume that the term depth of t_i is minimal over all counterexamples. Using Lemma 3 we perform a case split on t_i:

Suppose t_i is a logic variable X. Then we will have that $\sigma \in \phi(\{(X, C_j)\})$, and since the HOL type of t_i is the same as the HOL type of σt_i this substitution was added to the set in step 3 of the initialization. Contradiction.

Suppose t_i is a constant or normal variable. Then $t_i = t_j$, and since $\mathcal{C}$ is closed we have that $C_i = C_j$. By Lemma 2 the representative of C_i contains no logic variables at all, and so the set $\{\}$ was added to $match(C_i, C_i)$ in step 4 of the initialization. $\phi(\{\})$ contains every variable instantiation, so certainly contains σ. Contradiction.

Suppose $t_i = lambda(v, t_B)$, so $t_j = lambda(v, t_b)$, and we must have classes such that $t_B \in_{\mathcal{C}} C_B$ and $t_b \in_{\mathcal{C}} C_b$. Since $\sigma t_B = t_b$ and t_B has strictly smaller term depth than t_i (by the normal form property), we must have a substitution $S \in match(C_B, C_b)$ such that $\sigma \in \phi(S)$. Therefore S must have been added to $match(C_i, C_j)$ in the lambda inductive step. Contradiction.

Finally suppose $t_i = app(t_F, t_A)$, so $t_j = app(t_f, t_a)$, and we must have classes such that $t_F \in_{\mathcal{C}} C_F$, $t_A \in_{\mathcal{C}} C_A$, $t_f \in_{\mathcal{C}} C_f$ and $t_a \in_{\mathcal{C}} C_a$. Since $\sigma t_F = t_f$ and t_F has strictly smaller term depth than t_i, we must have a substitution $S \in match(C_F, C_f)$ such that $\sigma \in \phi(S)$. Similarly we must have $S' \in match(C_A, C_a)$ such that $\sigma \in \phi(S')$. By Lemma 1 we have that $\phi(S) \cap \phi(S') = \phi(S \cup S')$, so $\sigma \in \phi(S \cup S')$, and hence S and S' are compatible. This implies that $S \cup S'$ was added to $match(C_i, C_j)$ in the application inductive step. Contradiction.

4 Percolation Algorithm

The Percolation Algorithm performs undirected rewriting on the congruence classes, and is an easy application of the Matching Algorithm. Here is how it works:

1. Perform the Matching Algorithm.
2. For every substitution S in $match(C_i, C_j)$ and for every element t in C_i, create the element t' by applying the substitution S to t, and add t' to C_j. This step is illustrated in Figure 1.
3. Perform a close operation.

This is one iteration of the percolation algorithm. There is no hope here of always reaching a fixed-point after many iterations; this would allow us to decide the word problem for the equalities in $\mathcal{C}$, which is undecidable [2]. In general our algorithm is a semi-decision procedure for the word problem, and in particular cases where a fixed-point is reached it is a full decision procedure.

Returning to the example used in Section 3 to illustrate the matching algorithm, suppose the percolation algorithm had performed step 1 and been returned the *match* sets we created in the example. In step 2 it would be able to make exactly one addition: adding $\top$ to the class with representative $f\ \top\ \top$. After the close operation in step 3, this is how the classes look:

$$
\begin{array}{lll}
\{\top & \{\top, & f\ X\ X, f\ \top\ \top\}, \\
\ \ \bot & \{\bot & \}, \\
\ \ f & \{f & \}, \\
\ \ X & \{X & \}, \\
f\ X & \{f\ X & \}, \\
f\ \top & \{f\ \top & \}\ \}
\end{array}
$$

In practice we give each element a level number, and we define the level of a substitution S in $match(C_i, C_j)$ to be the maximum level of: the element in C_j that was matched; and the substitutions used to create S. When we add new elements to C_j we give them the level of the match plus one. Now if

we insist that the level of all elements added is less than some maximum then we can run the percolation algorithm until a fixed-point is reached, since now it is guaranteed to exist. This method of allocating level numbers was chosen so that we can give our rewrite rules (like $X + Y = Y + X$) an initially high level number, which will then stop them being rewritten like everything else!

5 Results

The examples we chose to test the program are listed in Table 1, and the results for these examples are detailed in Table 2. The Percolation Algorithm is not the most efficient way to prove the test theorems, and gets bogged down when the maximum level is set above about 4 (depending on which rewriting theorems are used). The first example is a triviality that is impossible for standard congruence closure [2]; the second example (from the introduction) derives the right inverse law from the group axioms; the other examples depend on the usual arithmetic rewrites and are designed to show what happens in such a rich theory. We give each example in Table 1 a number which is used in Table 2, and also show the minimum level necessary to prove the result. The columns in Table 2 in order are: example number, iteration number, number of classes, number of elements, number of matches, time to find the matches, time to add the new elements (assimilation), time to perform congruence closure, and total time spent on the iteration. We measured the number of elements and classes before each iteration. At the end of all the iterations we write in bold the total time spent in each phase.[4] Note that the standard first order prover in HOL, `MESON_TAC`, proves examples 1, 3 and 4 in under 2 seconds, but can't prove the others at all (at least, not in any reasonable time).

The first thing that can be noticed from Table 2 is that the times to perform the three phases are of the same order: no one phase relatively dominates or vanishes. The second thing is that there is an awfully large number of matches for even small maximum levels, bringing in a correspondingly large number of new classes and elements. The congruence closure phase can reduce the size of the class set by a factor of 10 after an assimilation, and if we weren't using congruence classes this reduction in space would not be possible. However, despite this saving, undirected rewriting causes a subterm explosion; all congruence classes can really do is postpone the point beyond which we lose control.

6 Conclusions

We have described the technical steps involved in integrating logic variables with congruence classes, and presented an algorithm to discover matches between classes. This problem is analogous to the problem of finding the shortest path between two points on a weighted graph: in both cases the solution algorithm computes seemingly more than is required. Dijkstra's Algorithm [6] solves the shortest path problem but must compute the shortest path from the source point to every other point; and the matching algorithm presented here computes all matches between every pair of classes.

This 'side-effect' is exploited in the Percolation Algorithm: finding every match is exactly what we want for undirected rewriting, and congruence classes provide a efficient storage mechanism. However, the results show that it is not efficient enough to stem the tide in theories such as arithmetic with large sets of equalities, and it cannot compete with a rewriter when there exists a directed rewrite set. This suggests a natural compromise: for all equalities that make up such a rewrite set, let the rewriter work and feed the resulting simplifications to the congruence classes, we can then run the Percolation Algorithm on these and the rest of the equalities.

This further integration will form the basis of future work, as well as the original aim of building a deductive proof procedure on top of the congruence classes. Unification is frequently used in deductive

[4] All times are in seconds, using version Taupo 2 of `hol98` running on an Intel Pentium III 600MHz. The memory use for the examples is less than 10Mb, and garbage collection accounts for about 10% of the matching time, 5% of the assimilation time, and 20% of the closing time.

Table 1. Examples Chosen to Test the Percolation Algorithm.

Example	Level	Theorem
1	1	$(\forall x.\ f(f(x)) = g(x)) \Rightarrow (f(g(a)) = g(f(a)))$
2	2	$(\forall x\, y\, z.\ ((x * y) * z = x * (y * z)) \wedge (e * x = x) \wedge (i(x) * x = e)) \Rightarrow (x * i(x) = e)$
3	1	$abc = cba$
4	2	$abcd = dcba$
5	2	$abcde = edcba$
6	3	$(a+1)(a+1) = aa + a + a + 1$

Table 2. Detailed Profiles of the Percolation Algorithm on the Examples.

Ex	It	#C	#E	#M	Match	Assim.	Close	Total
1	1	17	21	26	0.003	0.005	0.007	0.015
	2	15	22	24	0.004	0.005	-	0.009
					0.007	**0.010**	**0.007**	**0.024**
2	1	28	36	46	0.008	0.011	0.012	0.031
	2	28	40	126	0.040	0.125	0.069	0.234
	3	67	101	754	0.738	1.034	0.121	1.893
	4	85	134	857	1.308	1.283	0.079	2.670
	5	76	122	994	1.962	1.138	0.088	3.188
	6	77	129	951	1.738	1.242	0.068	3.048
	7	80	145	857	1.578	1.179	-	2.757
					7.372	**6.012**	**0.437**	**13.821**
3	1	51	70	109	0.027	0.054	0.075	0.156
	2	59	98	118	0.035	0.054	-	0.089
					0.062	**0.108**	**0.075**	**0.245**
4	1	56	75	143	0.033	0.083	0.108	0.224
	2	76	129	1047	0.499	1.348	2.633	4.480
	3	189	401	1168	0.826	1.169	-	1.995
					1.358	**2.600**	**2.741**	**6.699**
5	1	61	80	177	0.045	0.126	0.161	0.332
	2	89	154	1413	0.665	2.053	6.289	9.007
	3	188	477	1562	1.074	1.976	-	3.050
					1.784	**4.155**	**6.450**	**12.389**
6	1	53	72	131	0.033	0.092	0.076	0.201
	2	69	114	770	0.399	1.034	0.957	2.390
	3	83	179	3507	5.929	8.950	10.914	25.793
	4	96	228	2609	5.617	4.995	0.607	11.219
	5	200	406	2501	5.403	6.837	-	12.240
					17.381	**21.908**	**12.554**	**51.843**

proof search, and perhaps there exists a unification algorithm on classes analogous to the matching algorithm presented in this paper. We have briefly investigated this, and the problem seems to be more difficult: at the separation of variables phase we may have to create new classes to accommodate the new variables, so even our termination proof fails in the new context. There are also some improvements to be made to the current congruence class component, including allowing type variables to match (which will be useful in polymorphic theories), and allowing higher-order matching by including conversions on the underlying lambda calculus. This could perhaps work in a similar way to the Percolation Algorithm, looking for terms of the form $app(lambda(v, C_i), C_j)$, and then producing a new term by replacing all occurrences of v in C_i with C_j.

7 Related Work

McAllester [10] has also used congruence classes for storing terms, in order to perform rewriting in non-confluent theories. The application is different but the theory is very similar, since it is motivated by the same goal of reducing the space needed to store terms. One notable difference is that McAllester uses context-free grammars to represent the current state of the congruence classes, whereas we use an ad-hoc representation. However, in higher-order logic this doesn't really matter, because there are only two term constructors. Another difference is that in our system the equations for rewriting come from the congruence classes, whereas in McAllester's setup they are prescribed in advance. This is in keeping with our respective applications, McAllester is seeking to improve rewriting performance, and we want to improve the handling of equality in deductive proof procedures.

Apart from this, the most frequent use of congruence classes has been in the heart of the Nelson-Oppen combination of decision procedures [11, 12], which uses congruence closure to propagate the equalities generated by the component decision procedures. Since traditional congruence closure treats term variables as constants, most of the work in equality reasoning has concentrated on rewriting. There is much theory on this (see Baader & Nipkow [2] for a comprehensive overview), and many systems (powerful simplifiers exist in many theorem provers, including HOL[5], Isabelle[6] and PVS[7]; and the OBJ family of languages is based on rewriting logic [7]).

There have been many attempts to integrate equality reasoning with deductive proving, with varying degrees of sophistication. The first-order prover Gandalf [13] includes paramodulation as a basic proof step, as do many others, while Harrison's implementation in HOL of Model Elimination [8] axiomatizes equality and relies on deductive proof alone. A comparison of the two [9] suggests that paramodulation is more effective than pure deduction. In this paper we have argued that even more infrastructure would improve performance again.

On the more general note of interleaving equality and deductive steps, we observe that Boyer and Moore [5] have argued for a tighter integration of decision procedures into provers, and Zammit [14] has made use of rewriting interleaved with deductive proof steps by generalizing proof rules with simplifiers.

Acknowledgements

I had many fruitful discussions with Konrad Slind, and my supervisor, Mike Gordon, while engaging on this research. Donald Syme and Michael Norrish also gave me a helping hand to produce the work. The comments of the anonymous referees of the TPHOLs conference and the IGPL journal improved the paper enormously.

[5] `http://www.ftp.cl.cam.ac.uk/ftp/hvg/hol98/`
[6] `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/`
[7] `http://www.csl.sri.com/sri-csl-pvs.html`

References

1. Wilhelm Ackermann. *Solvable Cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
3. Richard J. Boulton. Lazy techniques for fully expansive theorem proving. *Formal Methods in System Design*, 3(1/2):25–47, August 1993.
4. Richard J. Boulton. Combining decision procedures in the HOL system. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 75–89, Aspen Grove, UT, USA, September 1995. Springer-Verlag.
5. Robert S. Boyer and J Strother Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. *Machine Intelligence*, 11:83–124, 1988.
6. Thomas H. Cormen, Charles Eric Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, Cambridge, Massachusetts, 1990.
7. J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs.* MIT Press, Cambridge, Mass., 1st edition, 1996.
8. John Harrison. Optimizing proof search in model elimination. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, volume 1104 of *Lecture Notes in Computer Science*, pages 313–327, New Brunswick, NJ, 1996. Springer-Verlag.
9. Joe Hurd. Integrating Gandalf and HOL. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321. Springer, September 1999.
10. David McAllester. Grammar rewriting. In Deepak Kapur, editor, *Automated Deduction, CADE-11: 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15–18, 1992: Proceedings*, number 607 in Lecture Notes in Artificial Intelligence, pages 124–138. Springer-Verlag, 1992.
11. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
12. Greg Nelson and Derek C. Oppen. Fast decision procedures bases on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, April 1980.
13. Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, April 1997.
14. Vincent Zammit. *On the Readability of Machine Checkable Formal Proofs.* PhD thesis, University of Kent at Canterbury, March 1999.

A HOL Implementation

A.1 Notation

Terms in HOL are either constants, variables, function applications of terms to terms ($app(t_1, t_2)$), or lambda abstractions ($lambda(v, t)$). Thus `5 + 3` is really app(`$+ 5`, 3), which in turn is $app(app$(`$+`, 5), 3). Note that infix operators gain the prefix `$` when they are not in their usual infix position.

The HOL constants `T` and `F` represent the logical *true* and *false*. We use HOL variables called `lv0, lv1, lv2, ...` to denote logic variables.

HOL theorems can be created only using primitive inferences, and are printed in the form `[..] |- f lv3 = lv3`. The `|-` represents the logical $\vdash$ to mean syntactic derivability of the conclusion on the right from the assumptions on the left. `[..]` represents two assumptions, which by default HOL does not show explicitly.

There is a fixed set of HOL primitive inferences, and some that are relevant to this paper are: `ASSUME`, `REFL`, `SYM`, `TRANS`, and `MK_COMB`.

- `ASSUME` takes a term t and returns the theorem $t \vdash t$.
- `REFL` takes a term t and returns the theorem $\vdash t = t$.
- `SYM` takes a theorem of the form $\Gamma \vdash t_1 = t_2$ and returns the theorem $\Gamma \vdash t_2 = t_1$.

- `TRANS` takes two theorems of the form $\Gamma \vdash t_1 = t_2$ and $\Delta \vdash t_2 = t_3$, and returns the theorem $\Gamma \cup \Delta \vdash t_1 = t_3$.
- `MK_COMB` takes two theorems of the form $\Gamma \vdash f_1 = f_2$ and $\Delta \vdash a_1 = a_2$, and returns the theorem $\Gamma \cup \Delta \vdash app(f_1, a_1) = app(f_2, a_2)$.

A.2 Congruence Classes

To implement congruence classes in HOL we define the following ML datatypes:

```
datatype elt = Elt of term * thm;
datatype class = Class of term * elt list;
```

Note firstly that `type1 * type2` is the way ML represents a cartesian product type. The `term` in the `class` type is the class representative; within `elt` the `term` is the class element and the `thm` is the equality theorem 'the element is equal to the class representative'. The reason we include the term separately when it is always the left hand side of the theorem is to speed up access.

To give a feeling for the set-up, we give a simple example. *Here is the initial class set to which* we will be adding terms:

```
- classes_null;
> val it =
    [Class(T, [Elt(T,  [] |- T = T)]),
     Class(F, [Elt(F,  [] |- F = F)])]
    : class list
```

Now we are going to add a fact to the class set using `introduce_fact`, a function which adds the conclusion of the theorem to the class set (involving a conversion to normal form), then performs a close with the information that the fact should be in the true class.

```
- val cs = introduce_fact (ASSUME 'f (f (f lv3)) = (lv3:'a)') classes_null;
> val cs =
    [Class(lv3, [Elt(lv3, |- lv3 = lv3),
                 Elt(f (f (f lv3)),  [.] |- f (f (f lv3)) = lv3)]),
     Class(f lv3, [Elt(f lv3, |- f lv3 = f lv3)]),
     Class(f (f lv3), [Elt(f (f lv3), |- f (f lv3) = f (f lv3))]),
     Class(f, [Elt(f, |- f = f)]),
     Class($=, [Elt($=, |- $= = $=)]),
     Class(T, [Elt(lv3 = lv3,  [.] |- (lv3 = lv3) = T),
               Elt(T, |- T = T)]),
     Class($= lv3, [Elt($= lv3,  [.] |- $= lv3 = $= lv3)]),
     Class(F, [Elt(F, |- F = F)])]
    : class list
```

Note that we have precisely one logic variable (`lv3`); the others are normal variables. We now add another fact which produces the following collapse:

```
- val cs' = introduce_fact (ASSUME 'f (f (f (f lv3)))) = (lv3:'a)') cs;
> val cs' =
    [Class(lv3, [Elt(lv3, |- lv3 = lv3),
                 Elt(f lv3,  [..] |- f lv3 = lv3)]),
     Class(T, [Elt(lv3 = lv3,  [.] |- (lv3 = lv3) = T),
               Elt(T, |- T = T)]),
     Class($=, [Elt($=, |- $= = $=)]),
     Class(f, [Elt(f, |- f = f)]),
     Class($= lv3, [Elt($= lv3,  [.] |- $= lv3 = $= lv3)]),
     Class(F, [Elt(F, |- F = F)])]
    : class list
```

Finally we add a new term to the class set:

```
add_term 'f (f (f (f (lv3:'a))))' cs';
> val it =
    ( [..] |- f (f (f (f lv3))) = lv3,
     [Class(lv3,
      ...same classes as last time...
      Class(F, [Elt(F, |- F = F)])])
    : Thm.thm * class list
```

Notice the theorem that we get back in addition to the new class set, which tells us the normal form.

A.3 Matching Algorithm

In our implementation, we combine all the $match(C_i, C_j)$ sets for each class C_j, and define a `match` datatype to store the substitutions:

```
datatype match = Match of (term * (int * term) list) * (thm * thm);
```

The first `term` is rep_i, and the `(int * term) list` represents a substitution (logic variables are numbered, and the `term` is the class representative). Finally the first theorem is of the form $t = rep_i$ and the second is $t' = rep_j$, where if we apply the substitution to t we will get t'. We need this to prove to HOL that the substitution is really valid.

Initializing the *match* sets is easily implemented, and we will briefly comment on the method for the application inductive step, as illustrated in Figure 2. We assume there are compatible substitutions S taking F to f and S' taking A to a. We have the term $app([F],[A])$[8] in class C_i and term $app([f],[a])$ in class C_j, and want to construct the application match. The two theorems we need are of the form $t = [app([F],[A])]$ and $t' = [app([f],[a])]$, for any t and t' such that the substitution $S \cup S'$ takes t to t'. If we choose $t = app(F, A)$ and $t' = app(f, a)$, then the required two theorems can be created like this:

$$E_t = \texttt{TRANS}\ (\texttt{MK\_COMB}\ (E_F, E_A))\ E_i$$
$$E'_t = \texttt{TRANS}\ (\texttt{MK\_COMB}\ (E_f, E_a))\ E_j$$

Note that we are using HOL theorems at every stage, which may result in a bottleneck in the logical kernel. Boulton [3] provides a lazy-theorem approach to avoid such problems if they arise.

[8] Recall that $[X]$ is the notation for the class representative of the term X.

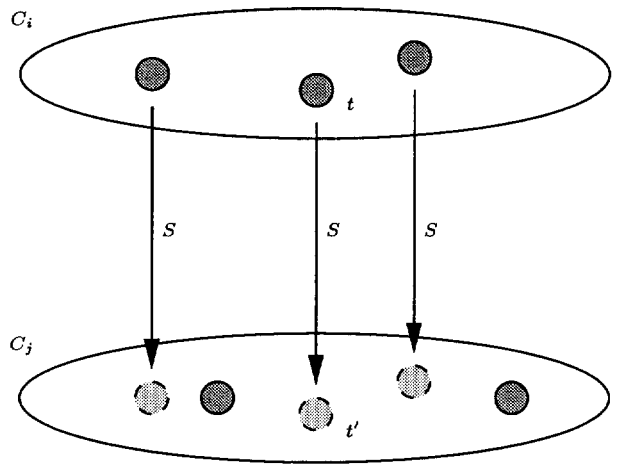

Fig. 1. One Step of the Percolation Algorithm.

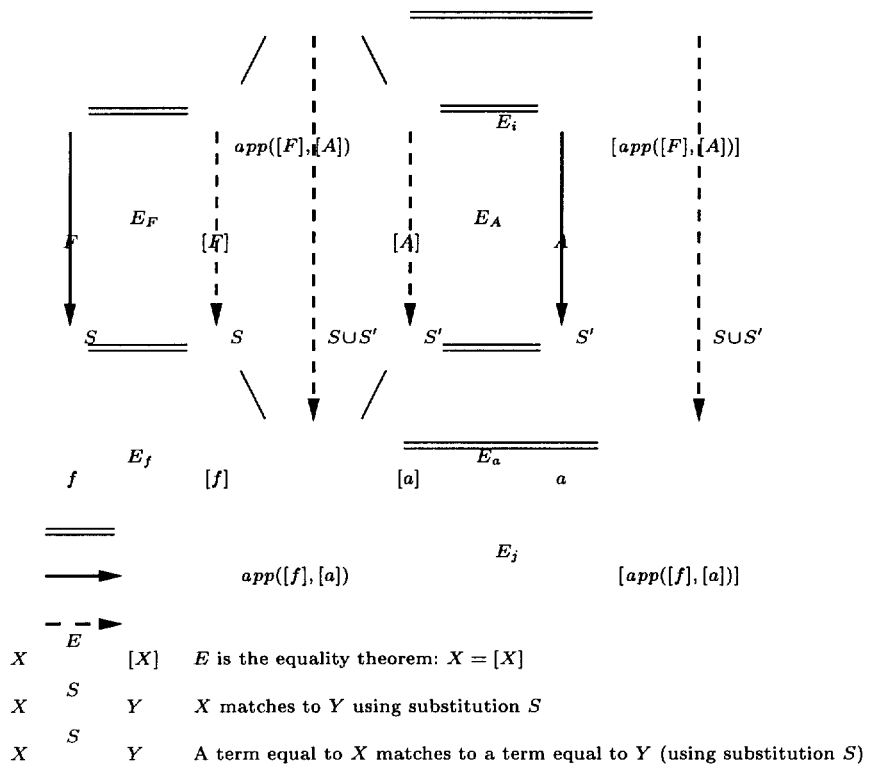

Fig. 2. Matching through an application.

Lightweight Probability Theory for Verification
Category B Paper

Joe Hurd*

Computer Laboratory
University of Cambridge
joe.hurd@cl.cam.ac.uk

Abstract. There are many algorithms that make use of probabilistic choice, but a lack of tools available to specify and verify their operation. The primary contribution of this paper is a lightweight modelling of such algorithms in higher-order logic, together with some key properties that enable verification. The theory is applied to a uniform random number generator and some basic properties are established. As a secondary contribution, all the theory developed has been mechanized in the hol98 theorem-prover.

1 Introduction

The Miller-Rabin primality test takes as input an odd integer n, and returns either the result $\top$ or the result $\bot$. If n actually is prime, then it is guaranteed to return $\top$. If n is composite, then it will return the result $\bot$ with probability at least one half.

There are many such algorithms with a probabilistic specification, and more that make use of probabilistic choice in their operation. Examples of algorithms with probabilistic specifications are certain capacity allocation algorithms on multimedia networks and soft real-time scheduling algorithms, where the quality of service is stated as "*the probability that things will go wrong (the combined demands of the clients will be more than the physical capacity of the network/the deadline will be missed) is less than* 2^{-s}", where s is some appropriately large number [6]. Examples of algorithms with deterministic specifications that make use of probabilistic choice abound in randomized versions of basic algorithms, where the purpose of the probabilistic choice is to stop there being a deterministic class of 'bad' inputs. For instance, the randomized version of quicksort chooses pivots at random, to stop there being a fixed class $C(n)$ of input permutations that result in $O(n^2)$ performance. Instead for any given run of the algorithm there is a probability $|C(n)|/n!$ that it will take $O(n^2)$.

We would like to be able to manipulate and reason about probabilistic algorithms in higher-order logic, but to date there are no tools to support this. The reason for choosing higher-order logic is that it is an expressive logic to state and prove properties of algorithms, and it has been implemented in various theorem-provers containing many pre-established theories which we can use to formalize our results.

The primary contribution of this paper is an approach for modelling probabilistic algorithms in higher-order logic, allowing us to specify and verify their operation. In order to write specifications we need to make precise the notion of probability, and we do this by formalizing some mathematical measure theory in higher-order logic. We also demonstrate how to apply this framework, by stating a probabilistic specification of the Miller-Rabin algorithm and proving some properties of a uniform random number generator.

The secondary contribution is the mechanization of this theory in the higher-order logic theorem-prover hol98[1], and in the conclusion we briefly comment on this experience.

The structure of the paper is as follows: Section 2 sets out the fundamental model of probabilistic algorithms; Section 3 builds up the necessary measure theory to define event probability; Section 4

* Supported by an EPSRC studentship

[1] http://www.ftp.cl.cam.ac.uk/ftp/hvg/hol98/

covers independence which turns out to be critically important; and Section 5 applies the theory to a uniform random number generator. Finally in Section 6 we draw some conclusions from the work, and in Sections 7 and 8 we look at future prospects and related work.

1.1 Notation

We use some types and definitions from higher-order logic, and it is worthwhile to gather their explanations into one place. Let $\mathbb{B} = \{\top, \bot\}$ be the type of booleans, $\mathbb{N} = \{0, 1, 2, \ldots\}$ the type of natural numbers and $\mathbb{R}$ the type of real numbers. $\mathbb{B}^*$ is the type of boolean lists, $\mathbb{B}^{**}$ is the type of lists of boolean lists, and $\mathbb{B}^\infty = \mathbb{N} \to \mathbb{B}$ is the type of infinite sequences of booleans. For $s : \mathbb{B}^\infty$ we write s_i to mean the ith element of s, instead of the technically correct but odd-looking $s(i)$. We also use set operations throughout the paper, but it should be kept in mind that all sets in higher-order logic are typed, modelled by the type $\alpha \to \mathbb{B}$.

We also use some primitive functions that should be explained. We assume basic arithmetic operations on the number types, including suc (successor) on the naturals and sup and inf on the reals. The functions fst and snd have their usual meanings on pairs, as do hd, tl, cons, length, map and append on lists. The higher-order function curry takes a function defined on pairs and converts it to curried form, uncurry does the opposite, and $\circ$ is function composition.

We define sequence analogues of the list functions hd, tl, take and drop, named shd : $\mathbb{B}^\infty \to \mathbb{B}$, stl : $\mathbb{B}^\infty \to \mathbb{B}^\infty$, stake : $\mathbb{N} \to \mathbb{B}^\infty \to \mathbb{B}^*$ and sdrop : $\mathbb{N} \to \mathbb{B}^\infty \to \mathbb{B}^\infty$. Here are the logical definitions:

$$\begin{aligned}
\mathsf{shd}\ s &= s_0 \\
\mathsf{stl}\ s &= s \circ \mathsf{suc} \\
\mathsf{stake}\ 0\ s = [\,] \quad &\wedge \quad \mathsf{stake}\ (\mathsf{suc}\ n)\ s = \mathsf{cons}\ (\mathsf{shd}\ s)\ (\mathsf{stake}\ n\ (\mathsf{stl}\ s)) \\
\mathsf{sdrop}\ 0\ s = s \quad &\wedge \quad \mathsf{sdrop}\ (\mathsf{suc}\ n)\ s = \mathsf{sdrop}\ n\ (\mathsf{stl}\ s)
\end{aligned}$$

Finally, let $\mathsf{sdest}\ s = (\mathsf{shd}\ s, \mathsf{stl}\ s)$.

2 Modelling Probabilistic Algorithms

We model a random number generator with an infinite sequence of 'randomly chosen' booleans. There are no global variables in higher-order logic, so functions that require random numbers must provide an extra argument to pass in the sequence, and must pass back a subsequence of 'unused random booleans' in addition to the normal results.

As an example, here is a (top-level) implementation of the Miller-Rabin algorithm in ML:

```
fun MR n = let val a = uniform (n - 1)
           in MR_det (a + 1) n end;
```

`uniform` is a uniform random number generator, and `MR_det` performs the deterministic Miller-Rabin test.

Definition 1. *Miller-Rabin is modelled in higher-order logic as follows:*

$$\begin{aligned}
\forall n\, s.\ \mathsf{MR}\ n\ s = \mathsf{let}\ (a, s') = \mathsf{uniform}\ (n-1)\ s \\
\mathsf{in}\ (\mathsf{MR_det}\ (a+1)\ n,\ s')
\end{aligned}$$

Since MR : $\mathbb{N} \to \mathbb{B}^\infty \to \mathbb{B} \times \mathbb{B}^\infty$ and uniform : $\mathbb{N} \to \mathbb{B}^\infty \to \mathbb{N} \times \mathbb{B}^\infty$ rely on random numbers, they take an additional boolean sequence argument (s), and return a sequence of unused booleans. MR does not refer to any elements of the sequence in its body, so it can safely pass back the same sequence that uniform passed back. Note that since MR_det : $\mathbb{N} \to \mathbb{N} \to \mathbb{B}$ is deterministic, its definition has not changed at all.

A random bit generator $\mathbb{B}^\infty \to \mathbb{N} \times \mathbb{B}^\infty$ provides a further example:

$$\lambda s.\ (\mathsf{if\ shd}\ s\ \mathsf{then\ 1\ else\ 0},\ \mathsf{stl}\ s)$$

This takes a sequence and returns a bit together with a sequence of unused booleans, which in this case is the tail of the sequence.

This model of a random number generator is not a new one; it is a monadic state transformer used in pure functional languages such as Haskell[2] to pass around state [8, 13]. Translating into the language of monads gives our theory some guidance:

Definition 2. *Let the higher-order logic type* α monad *denote the type* $\mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$. *Define the two functions* unit : $\alpha \to \alpha$ monad *and* bind : α monad $\to (\alpha \to \beta$ monad$) \to \beta$ monad *as follows:*

$$\mathsf{unit}\ x = \lambda s.\ (x, s)$$
$$\mathsf{bind}\ f\ g = \mathsf{uncurry}\ g \circ f$$

All the monad laws hold for this definition, and the notation allows us to write functions without explicitly mentioning the sequence that is passed around. For example, the random bit generator above is equal to:

$$\mathsf{bind\ sdest}\ (\lambda b.\ \mathsf{if}\ b\ \mathsf{then\ unit\ 1\ else\ unit\ 0})$$

As well as simplifying the definition of probabilistic algorithms, there are further advantages to using monadic notation: these will be explained in Section 4.

Now we have a basis to talk about probability. Suppose we have a probabilistic algorithm and a specification; this splits $\mathbb{B}^\infty$ into a set E of 'good' sequences that result in the algorithm meeting its specification, and a set $\mathbb{B}^\infty - E$ of 'bad' sequences that do not[3]. The precise meaning of "*the probability that the algorithm meets the specification*" is $\mathbb{P}(E)$, the probability of the set E.

Definition 3. *Here is the specification of the Miller-Rabin algorithm:*[4]

$$\begin{aligned}\forall n.\ \mathsf{odd}\ n\ \Rightarrow (\mathsf{prime}\ n\ &\Rightarrow\ (\forall s.\ \mathsf{MR}\ n\ s = \top))\\ \wedge\ (\neg\mathsf{prime}\ n\ &\Rightarrow\ \mathbb{P}(\{s : \mathsf{MR}\ n\ s = \bot\}) \geq 1/2)\end{aligned}$$

We now have two formalization tasks ahead of us: enough measure theory to precisely define our probability measure $\mathbb{P} : \mathcal{P}(\mathbb{B}^\infty) \to \mathbb{R}$, and some key results of probability theory to assist us in the verification of probabilistic algorithms. These are accomplished in the next two sections.

3 Lightweight Measure Theory

In this section we take a mathematical measure theory based on sets [14], and apply it to our problem of defining in higher-order logic a probability measure $\mathbb{P}$ on subsets of $\mathbb{B}^\infty$.

3.1 Theoretical Definitions

Suppose we have a measure function $\mu : \mathcal{P}(\mathbb{B}^\infty) \to \mathbb{R}$ from sets of sequences to the reals. Here are two desirable properties of μ:

[2] `http://www.haskell.org/`

[3] The good and bad sets are obviously disjoint, and since in higher-order logic all functions are total these two sets together make up the whole of $\mathbb{B}^\infty$

[4] Note that the prime n case is **not** the same as $\mathbb{P}(\{s : \mathsf{MR}\ n\ s = \top\}) = 1$, saying the property is true for every sequence is much stronger than this.

- **Positivity**: $\forall E.\ 0 \leq \mu(E) \leq 1$ with $\mu(\mathbb{B}^\infty) = 1$.
- **Additivity**: $\forall EE'.\ E \cap E' = \emptyset \Rightarrow \mu(E \cup E') = \mu(E) + \mu(E')$.

Unfortunately a celebrated result of Banach and Tarski showed that if the Axiom of Choice is assumed,[5] then there exist sets that are non-measurable, so we have to be more cautious than simply asserting that such a μ exists. Instead we carve out a subset of $\mathcal{P}(\mathbb{B}^\infty)$, a measurable space, upon which a particular μ does exist with the desired properties. This motivates the next few definitions.

Definition 4. *Consider the function* $\mathsf{em}_1 : \mathbb{B}^* \to \mathcal{P}(\mathbb{B}^\infty)$ *from lists of booleans to subsets of* $\mathbb{B}^\infty$*, where*

$$\mathsf{em}_1([b_0, \ldots, b_{n-1}]) = \{s : s_0 = b_0 \wedge \cdots \wedge s_{n-1} = b_{n-1}\}$$

So $\mathsf{em}_1(l)$ *is the set of all sequences that have initial segment* l*.*

Now we define our **embedding function** $\mathsf{em} : \mathbb{B}^{**} \to \mathcal{P}(\mathbb{B}^\infty)$ *by*

$$\mathsf{em}([l_0, \ldots, l_{n-1}]) = \mathsf{em}_1(l_0) \cup \cdots \cup \mathsf{em}_1(l_{n-1})$$

Definition 5. *A set* $E \subseteq \mathbb{B}^\infty$ *is* **measurable** *if there exists an* $l : \mathbb{B}^{**}$ *such that* $\mathsf{em}(l) = E$*. We define the* **measurable space** $\mathcal{A} \subset \mathcal{P}(\mathbb{B}^\infty)$ *to be the set of all measurable sets.*

Proposition 1. *It can be shown that* $\mathcal{A}$ *is an* **algebra**, *i.e., that it contains the empty set and is closed under finitely many intersection, union and complement operations on its elements.*

Definition 6. *We next define a measure* $m : \mathbb{B}^{**} \to \mathbb{R}$ *on the representation type:*

$$m([l_0, \ldots, l_{n-1}]) = \sum_{0 \leq i < n} 2^{-(\mathsf{length}\ l_i)}$$

We may now define a **measure function** $\mu : \mathcal{A} \to \mathbb{R}$ *on the algebra, by reference to the underlying* $\mathbb{B}^{**}$ *type:*

$$\mu(A) = \mathsf{inf}\{m(l) : \mathsf{em}(l) = A\}$$

Notice we have to be rather careful in this definition, because for a given $A \in \mathcal{A}$ there may be many $l : \mathbb{B}^{**}$ with $\mathsf{em}(l) = A$. As an example, both $[[\top]]$ and $[[\top, \top], [\top, \bot], [\top, \top]]$ embed to the set of all sequences beginning with $\top$. The definition respects this property, and since for every $l : \mathbb{B}^{**}$ we have that $m(l) \geq 0$ the infimum is always well-defined.[6]

Theorem 1. μ *satisfies the positivity and additivity conditions.*

In itself Definition 6 is not complete since all functions in higher-order logic must be total, so we must make a decision about what to do with the non-measurable sets. Here is the final definition of the probability function $\mathbb{P} : \mathcal{P}(\mathbb{B}^\infty) \to \mathbb{R}$:

Definition 7.

$$\mathbb{P}(E) = \mathsf{sup}\{\mu(A) : A \in \mathcal{A} \wedge A \subseteq E\}$$

$\mathbb{P}$ does not always satisfy the additivity condition (which was of course inevitable from the Banach-Tarski result), but it does at least satisfy **monotonicity**: $E \subseteq E' \Rightarrow \mathbb{P}(E) \leq \mathbb{P}(E')$; this follows from the definition of $\mathbb{P}$ and the additivity property of μ.

[5] and it certainly is in the higher-order logic we use, in common with most of mathematics.

[6] This follows immediately from the fact that we are working in the real numbers, though it will later turn out that for every $A \in \mathcal{A}$ there is always an $l : \mathbb{B}^{**}$ with $\mathsf{em}(l) = A$ and $m(l) = \mu(A)$.

3.2 Canonical Forms

The previous subsection contained all the theory we need in order to precisely define what we mean by probability. However, the definitions as they stand are completely unworkable. We cannot even easily calculate $\mathbb{P}(\mathbb{B}^\infty)$, the probability of the whole space. Why not? Because although $\mathsf{em}([[\,]]) = \mathbb{B}^\infty$ and $m([[\,]]) = 2^0 = 1$, there may also be an $l : \mathbb{B}^{**}$ with $\mathsf{em}(l) = \mathbb{B}^\infty$ and $m(l) < 1$.

To resolve this, we introduce a (computable) canonicalization function

$$c : \mathbb{B}^{**} \to \mathbb{B}^{**}$$

having two important properties:

- $\forall l\, l'.\ c(l) = c(l') \iff \mathsf{em}(l) = \mathsf{em}(l')$, i.e., $c(l)$ is a representative of the equivalence class containing l.
- $\forall l.\ m(c(l)) \leq m(l)$, i.e., the representative chosen has minimal measure.

Now to find the probability of a set $A \in \mathcal{A}$, we first find any $l : \mathbb{B}^{**}$ with $\mathsf{em}(l) = A$, and then we can evaluate $c(l)$ in the logic to give $\mathbb{P}(A) = m(c(l))$.

Although it would be tedious to go through the proofs of these theorems, it may provide some insight to explain what c is doing. Firstly, it sorts l according to an order defined on $\mathbb{B}^*$. Next it throws away any x in l where there is a y in l that is a prefix of x. This is valid because $\mathsf{em}(x) \subseteq \mathsf{em}(y)$, as can be seen from Definition 4. Finally, it merges elements with their twins. $x : \mathbb{B}^*$ and $y : \mathbb{B}^*$ are **twins** with **parent** $z : \mathbb{B}^*$ if $x = \mathsf{append}\ z\ [\top]$ and $y = \mathsf{append}\ z\ [\bot]$. If x and y are twins with parent z and x and y are both in l, then we can throw away x and replace y with z. The order function is carefully chosen to keep the list in order after a move like this, and after all three phases the canonicalization is complete.

With this technology, we can prove

Proposition 2.

$$\forall n.\ \mathbb{P}(\{s : s_n = \top\}) = 1/2$$

which says that for every boolean in the sequence, there is a probability of $1/2$ that it is $\top$, which fits well with our intuitive picture of a random number generator.

The canonicalization algorithm is not just useful to calculate probabilities, it also gives us an induction principle on elements of the algebra $\mathcal{A}$. We could just use list induction on the underlying $\mathbb{B}^{**}$ type, but this does not fit well with the natural structure of our sequence space $\mathbb{N} \to \mathbb{B}$ where we would like to use the sequence head and tail functions to give us a strong induction hypothesis. Here is the induction principle for lists in canonical form:

Theorem 2.

$$\begin{aligned}\forall Q.\ & Q([\,]) \wedge Q([[\,]]) \wedge (\forall l l'.\ (c(l) = l) \wedge (c(l') = l') \wedge Q(l) \wedge Q(l') \\ & \quad \Rightarrow Q(\mathsf{append}\ (\mathsf{map}\ (\mathsf{cons}\ \top)\ l)\ (\mathsf{map}\ (\mathsf{cons}\ \bot)\ l'))) \\ & \Rightarrow \forall l.\ (c(l) = l) \Rightarrow Q(l)\end{aligned}$$

The base cases are the lists $[\,]$ (embedding to the empty set) and $[[\,]]$ (embedding to the universe $\mathbb{B}^\infty$); the step case builds a set in canonical form from two others. To prove instances of the step case very often all we need to do is a case split on the sequence head followed by some rewriting with l or l' as appropriate.

3.3 Limitations

Our definition of probability is a **finite measure**; defined on fewer sets than definitions of probability that appear in most mathematical textbooks. Having defined a measure function on an algebra, they then

extend it to a measure on the smallest enclosing σ-algebra (using Caratheodory's extension theorem). A **σ-algebra** is an algebra that is also closed under countably many union operations on the elements.

We considered this unnecessary, because all the events that arise in the verification of probabilistic algorithms are members of $\mathcal{A}$, and so they already have a meaningful probability. However, it does lead to some anomalies in the underlying theory. If we start with $\mathbb{B}^\infty$, and for every list $l : \mathbb{B}^*$ we remove the sequence with initial segment l and continuing $\top\top\cdots$, then the set we are left with has probability zero (with our definition of probability).

Interestingly, there is a comment in Williams [14] (referring to the σ-algebra definition of measure) that makes an analogous point: "*although not all sets are measurable, it is always the case for probability theory that enough sets are measurable*".

4 Independence

Definition 8. *Two measurable sets E and E' are* **independent** *if*

$$\mathbb{P}(E \cap E') = \mathbb{P}(E)\mathbb{P}(E')$$

We write this as $E \amalg E'$.

Intuitively this says that knowing whether or not $s \in E$ gives you no information at all about whether or not $s \in E'$.

It is easy to underestimate the importance of this definition. Here are two reasons why it deserves attention:

1. Independence provides sanity checks on our definitions. It is not enough for us to prove that every boolean in the sequence has probability $1/2$ of being $\top$. This is also true in the case where every element of the sequence is always equal to every other element, and this certainly does not fit our intuitive picture of 'an infinite sequence of random booleans'. We also need to show that every element is independent of all the other elements, and only then will this establish that our sequence of booleans is genuinely Independent Identically Distributed (IID) [2].
2. Independence is essential for ease of verification. Many sets that naturally arise in a verification proof are of the form $E \cap E'$, and it would be tedious in the extreme to evaluate probabilities of this kind directly, instead of just multiplying the probabilities of E and E'.

Proposition 3. *For every number $n : \mathbb{N}$, function $r : \mathbb{B}^* \to \alpha$, predicate $p : \alpha \to \mathbb{B}$ and measurable set E:*

$$\{s : p\ (r\ (\mathsf{stake}\ n\ s))\} \amalg \{s : \mathsf{sdrop}\ n\ s \in E\}$$

Intuitively, this is obvious. Consider a probabilistic algorithm that reads n bits from a random number generator to compute a result (of type α). Knowledge of the result (from the predicate p) cannot give you any information about future bits from the random number generator, and conversely knowing the future bits of the random number generator gives no information about the result. It is this picture that allows us to generalise the result.

Given $l : \mathbb{B}^{**}$, say that l is a **cover set** if both $\mathsf{em}(l) = \mathbb{B}^\infty$ and for every two x and y in l we have that $\mathsf{em}_1(x) \cap \mathsf{em}_1(y) = \emptyset$. Observe that for every sequence s there is a unique $x \in l$ with $s \in \mathsf{em}_1(x)$. Define the **cover function** cov_l for l to be the function that takes s and returns the unique $x \in l$.

Proposition 4. *For every cover set l, function $r : \mathbb{B}^* \to \alpha$, predicate $p : \alpha \to \mathbb{B}$ and measurable set E:*

$$\{s : p\ (r\ (\mathsf{cov}_l\ s))\} \amalg \{s : \mathsf{sdrop}\ (\mathsf{length}\ (\mathsf{cov}_l\ s))\ s \in E\}$$

The intuitive picture is not much different from the previous one; again suppose a probabilistic algorithm is reading bits from a random number generator to compute a result. If at any point the list of bits read so far is a member of l, then immediately return. Again the result gives no information about future bits from the random number generator, and vice versa.

Definition 9. *Say a function* $f : \alpha$ monad *is* **split independent** *if there exist a cover set* l *and a function* $r : \mathbb{B}^* \to \alpha$ *such that*

$$\forall s.\ f\ s = (r\ (\mathsf{cov}_l\ s), \mathsf{sdrop}\ (\mathsf{length}\ (\mathsf{cov}_l\ s))\ s)$$

And now the following theorem is a trivial consequence of Definition 9 and Proposition 4:

Theorem 3. *If* $f : \alpha$ monad *is split independent then for every predicate* $p : \alpha \to \mathbb{B}$ *and measurable set* E*:*

$$\{s : p\ (\mathsf{fst}\ (f\ s))\}\ \amalg\ \{s : \mathsf{snd}\ (f\ s) \in E\}$$

This says that if a function is split independent, then the result and returned sequence will always be independent.[7] Split independence precisely formalizes the felicity condition that a function should not misuse the random number generator by 'reading ahead', and the next theorem shows how such functions compose:

Theorem 4. *1. For every* x, unit x *is split independent.*
2. sdest *is split independent.*
3. If f *is split independent and* $g(x)$ *is split independent for every* x*, then* bind f g *is split independent.*

This says that any function that we define using the unit, sdest and bind operations on sequences is guaranteed to be split independent, and thus the result and returned sequence will always be independent.

In Definition 1 we defined MR, a higher-order implementation of the Miller-Rabin algorithm. We can write this definition as

$$\forall n.\ \mathsf{MR}\ n = \mathsf{bind}\ (\mathsf{uniform}\ (n-1))\ (\lambda a.\ \mathsf{unit}\ (\mathsf{MR\_det}\ (a+1)\ n))$$

and now using Theorem 4 it is trivial to see that if uniform is split independent for every n then so will be MR.

5 Uniform Random Numbers

We start with the following definition of $\mathsf{unif} : \mathbb{N} \to \mathbb{N}$ monad.

Definition 10.

$$\begin{aligned}\mathsf{unif}\ n\ s = {} & \text{if } n = 0 \text{ then } (0, s)\\ & \text{else let } (m, s') = \mathsf{unif}\ (n \text{ div } 2)\ s\\ & \quad \text{in (if } \mathsf{shd}(s') \text{ then } 2m+1 \text{ else } 2m,\ \mathsf{stl(s')})\end{aligned}$$

Proposition 5. *For every* n *we have that* unif n *is split independent, and has the following probabilistic specification:*

$$\begin{aligned}&\forall n\, m.\ (n = m = 0)\ \vee\ 2^{m-1} \le n < 2^m\\ &\quad \Rightarrow\ \forall k.\ \mathbb{P}(\{s : \mathsf{fst}\ (\mathsf{unif}\ n\ s) = k\}) = (\mathit{if}\ k < 2^m\ \mathit{then}\ 2^{-m}\ \mathit{else}\ 0)\end{aligned}$$

This says that unif returns a number that has a uniform distribution on its range, but that its range can be up to twice as large as n.

We would like to use unif in the definition of a new function uniform that takes an argument n and returns a uniformly distributed number in the range $0, \ldots, n-1$.[8] The naive idea of calling unif n, and

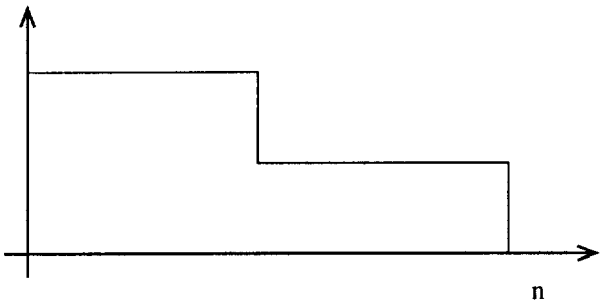

Fig. 1. The distribution of a naive attempt at uniform.

subtracting n from the result r if $r \geq n$ does not work: it produces the distribution as shown in Figure 1,[9] whereas the ideal distribution is completely flat.

Unfortunately, it is impossible to get a terminating uniform distribution on any n that is not a power of 2, because a simple argument shows that all events derived from initial segments of the sequence will have a probability that is rational and can be expressed with a power of 2 denominator, and all terminating algorithms can only read a finite number of booleans from the sequence. So in particular there is no terminating algorithm on sequences that returns an element of $\{0, 1, 2\}$, with probability $1/3$ each.

We compromise with the following definition:

Definition 11.

$$\begin{aligned}\mathsf{uniform}\ t\ n\ s = {} & \text{if } t = 0 \vee n = 0 \text{ then } (0, s) \\ & \text{else let } (m, s') = \mathsf{unif}\ (n-1)\ s \\ & \quad \text{in if } m < n \text{ then } (m, s') \text{ else } \mathsf{uniform}\ (t-1)\ n\ s\end{aligned}$$

This uniform function repeatedly evaluates unif, stopping when either unif returns a number in the correct range or when the 'cut-off' parameter t is zero. Its distribution is shown in Figure 2.

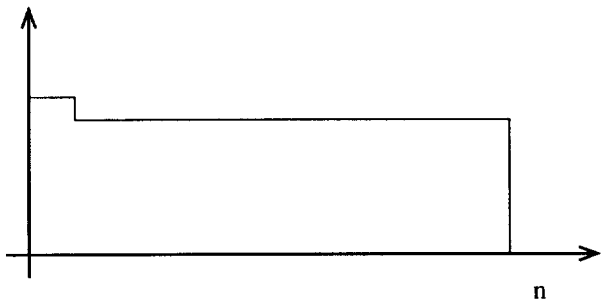

Fig. 2. The achieved distribution of uniform.

[7] Is the converse true? No: the function $\lambda s.(s_0 = s_1, \mathsf{stl}\ s)$ is a counter-example.

[8] For $n > 0$ of course, uniform should probably be undefined for $n = 0$.

[9] The numbers close to zero have twice the probability of numbers close to $n - 1$, because two results of unif n map to the smaller numbers, but only one to the larger numbers.

Theorem 5. *For every* $n : \mathbb{N}$, uniform n *terminates, is split independent, and has the following specification:*

$$\forall t\, n\, k.\ k < n \ \Rightarrow \left|\mathbb{P}(\{s : \mathsf{fst}\ (\mathsf{uniform}\ t\ n\ s) = k\}) - 1/n\right| < 2^{-t}$$

6 Conclusions

We have introduced a way to reason about probabilistic algorithms in a higher-order logic, described some of the technical difficulties that arise and their resolutions, and finally applied the theory to a simple example.

We used the `hol98` theorem-prover to do this work, creating a purely definitional theory to ensure soundness (in common with most other `hol98` theories). Particularly important for this work were the theories of real numbers [5] and predicate sets, although of course we found invaluable the basic theories of arithmetic, lists, well-founded recursion and the like. This paper represents approximately a month of work, and the resulting script files are about 5000 lines long. Automatic tools considerably sped up the whole development, particularly useful were the simplifier, first-order prover and function definition package. However, there is no substitute for picking the right definitions and coming up with general proof principles such as the canonical form induction theorem. These things had the most dramatic effect on the amount of labour required, and in addition resulted in a theory that was cleaner and more coherent than it would have otherwise been. The process of formalization gave insight.

There was only one area where the probability theory concepts and the corresponding models in higher-order logic did not fit well together, and that was termination. Consider the following probabilistic algorithm: "read bits from the sequence, and terminate when you first hit a $\bot$". This is undefinable as a computable function in higher-order logic, because there is one sequence, namely $\top\top\cdots$, upon which the algorithm does not terminate at all. But the probability that the algorithm will terminate is 1, because $\mathbb{P}(\mathbb{B}^\infty - \{\top\top\cdots\}) = 1$. If termination with probability 1 is an important enough concept, then it may be possible to create a new defining principle to allow it.

7 Future Work

Now that we have set up the basic model, there are many directions in which to take this further. In the short term, it would make sense to further develop the probability theory, to make the theory cleaner, easier to use and create some better automatic tools. One idea would be to take a Markov Chain (such as a random walk on the integers), recast it as a probabilistic algorithm, and try and establish the well-known properties. Once the theory is mature and stable, we can then look to verify some real programs with probabilistic guarantees, such as the allocation or scheduling algorithms mentioned in the introduction.

In another direction, there are computer algebra packages (such as Mathematica[10]) that offer basic statistical analysis functions, as well as many specialist statistical packages (such as SPSS[11]), but so far there is no theorem-proving system able to verify their results. This interaction has benefitted both theorem-provers and computer algebra systems in other domains; perhaps we could apply the technology to probability.

Finally, there are many application domains in which it is natural to use continuous distributions in the model. The obvious approach is to model the probability density functions directly with higher-order logic functions $\mathbb{R} \to \mathbb{R}$. Curiously, the current approach as it stands fits in very nicely if we use only infinite precision real arithmetic algorithms on the continuous distributions. All requests to the random number generator will be of the form "*produce a random number between 0 and 1 with precision n*", which directly transforms to "*produce n random bits*". Any algorithm of this form can be specified and verified using the techniques presented in this paper.

[10] `http://www.wolfram.com/`

[11] `http://www.spss.com/`

8 Related Work

Sometimes we can replace the probabilistic choice in a program with a stronger requirement, and prove the specification with this. An idea along these lines is to substitute demonic non-determinism for probabilitistic choice, which requires that every possible sequence of choices will satisfy the specification. Another version exists within the UNITY framework, which requires that every fair sequence of choices will satisfy the specification, where a sequence is fair if every choice point eventually chooses infinitely many of each choice. Demonic non-determinism is just universal quantification in higher-order logic, and UNITY has been also been integrated and mechanized [12, 10]. However, neither of these two approaches can deal with specifications that refer to probabilities strictly between 0 and 1 (such as Miller-Rabin), and even when no such references exist their requirements might still be too strong to give a proof.

Kozen [7] produced some early work in the verification of programs containing genuine probabilistic choice, and this was extended by Feldman and Harel [3]. The essential concepts of their system are the same as ours; programs operate on a sequence of random variables; the analysis is measure-theoretic, and cylinders (our measurable sets) play an important role; a computation level (where programs operate on concrete values) is distinguished from an analysis level (where the probability reasoning takes place). However, the logical embedding is very different. Their language is a two-level extension of first-order logic, with an axiomatised proof system, whereas our theory fits cleanly into higher-order logic. In summary, their model is more general (their sequence of random variables can have arbitrary distributions and they reason about partial functions), and ours is simpler, embedded in a more expressive logic, and integrated into an existing theorem-prover.

Members of the Oxford Programming Research Group have recently generalised Dijkstra weakest precondition semantics to probabilistic analyses [11, 9], and have used this to generate some paper proofs of various examples. We could easily have chosen this method instead of the one presented to model probabilistic programs; we felt however that the method chosen would involve less overall effort and would again integrate more cleanly with the existing theories.

Finally, theoretical results of combining probability and various (first-order) logics have been well-researched; papers of Halpern and Abadi [4, 1] give some decidability and complexity results.

Acknowledgements

The papers written by the Oxford Programming Research Group were instrumental in my initial understanding of probabilistic choice. I had many fruitful discussions with Judita Preiss, Konrad Slind, Michael Norrish and Anuj Dawar while engaging on this research. Finally my supervisor, Mike Gordon, greatly improved the paper by reviewing the first draft.

References

1. Martin Abadi and Joseph Y. Halpern. Decidability and expressiveness for first-order logics of probability. *Information and Computation*, 1994.
2. Morris DeGroot. *Probability and Statistics.* Addison-Wesley, 2nd edition, 1989.
3. V. A. Feldman and D. Harel. A probabilistic dynamic logic. *Journal of Computer and System Sciences*, 28(2):193–215, 1984.
4. Joseph Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 1990.
5. J. Harrison. *Theorem Proving with the Real Numbers (Distinguished dissertations).* Springer-Verlag, 1998.
6. F.P. Kelly. Notes on effective bandwidths. In F.P. Kelly, S. Zachary, and I.B. Ziedins, editors, *Stochastic Networks: Theory and Applications*, number 4 in Royal Statistical Society Lecture Notes Series, pages 141–168. Oxford University Press, 1996.
7. D. Kozen. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science*, pages 101–114, Long Beach, Ca., USA, October 1979. IEEE Computer Society Press.
8. John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'94), Orlando*, pages 24–35, June 1994.

9. Carroll Morgan. Proof rules for probabilistic loops. In *Proceedings of the BCS-FACS 7th Refinement Workshop*, 1996.
10. Lawrence C. Paulson. Mechanizing UNITY in Isabelle. *ACM Transactions on Computational Logic*, 2000. In press.
11. Karen Seidel, Carroll Morgan, and Annabelle McIver. An introduction to probabilistic predicate transformers. Technical Report TR-6-96, Oxford Programming Research Group Technical Report, 1996.
12. T. E. J. Vos. *UNITY in Diversity: A Stratified Approach to the Verification of Distributed Algorithms.* PhD thesis, Utrecht University, 2000.
13. Philip Wadler. The essence of functional programming. In *19th Symposium on Principles of Programming Languages.* ACM Press, January 1992.
14. David Williams. *Probability with Martingales.* Cambridge University Press, 1991.

A Executing Probabilistic Algorithms

Suppose we have proved that the Miller-Rabin algorithm MR (Definition 1) meets its specification (Definition 3). Then if a particular (odd) n is composite we have an algorithm for proving it in higher-order logic: evaluate $\mathsf{MR}\ n$ on different sequences until we find an s with $\neg\mathsf{MR}\ n\ s$.[12] This is one situation where we would like to execute our probabilistic algorithms in higher-order logic; another arises when we have defined a probabilistic algorithm and would like to debug it.

This is possible by defining a pseudo-random number generator in the logic, and for this we just need an initial seed $d : \alpha$ and an iteration function $i : \alpha \to \mathbb{B} \times \alpha$. For example, we could choose $d = 0$ and $i = \lambda n.\ (\mathsf{even}\ n, An + B \bmod (2N + 1))$.

We now make use of the following proposition, which creates a sequence from an iteration function.

Proposition 6.

$$\forall f.\ \exists g.\ (\forall x.\ \mathsf{shd}\ (g\ x) = \mathsf{fst}\ (f\ x))\ \wedge\ (\forall x.\ \mathsf{stl}\ (g\ x) = g\ (\mathsf{snd}\ (f\ x)))$$

Specializing this proposition to i gives us a g, and we can now pass in the sequence $g(d)$ to our probabilistic algorithm. During evaluation calls will be made to shd and stl on sequences of the form $g(x)$, and at this point we don't do this by rewriting with the definitions of shd and stl, but rather we use the above proposition which merely requires the evaluation of $i(x)$. In this way it closely simulates a real random number generator by carrying around its state, and avoids the horrible situation where to calculate the 100th bit of the sequence it must calculate the 99th, the 98th, etc., all the way back to the beginning.

[12] On a practical note, for implementation purposes we may not want to execute costly algorithms in the logic unless we know that they will succeed in the proof. Perhaps we could execute the algorithm in ML first until we find the right sequence for the proof, and then execute the algorithm in the logic just once.

A PCI Formalization and Refinement

Michael Jones, Ganesh Gopalakrishnan
50 S. Central Campus Dr. Rm. 3190
Salt Lake City, UT 84112-9205
mjones,ganesh@cs.utah.edu

University of Utah School of Computing

Abstract. We present our formalization of the PCI protocol using inference rules written in the PVS theorem prover. We created an abstractin of PCI, called PCI′, and have shown that PCI is a refinement of PCI′. The refinement proof was done by induction on PCI traces—using each inference rule as an inductive subgoal. The producer/consumer property for the PCI′ protocol was then exhaustively verified at the bus/bridge level using the Murϕ model checker in just under two minutes. No violations were found in the reduced model, implying that the PCI model also satisfies the producer/consumer property. The use of inference rules to describe the operational semantics of PCI and PCI′ facilitated the trace-inclusion refinement proof and provided a convenient model for use in both the theorem prover and model checker.

1 Introduction

The refinement proof discussed here arises in the context of a recently completed case study involving the PCI 2.1 local bus protocol [PCI95]. The goal was to show that the protocol, extended with local master IDs, and implemented over arbitrary acyclic networks satisfies the producer/consumer transaction ordering property at the bus/bridge level. The PCI 2.1 protocol is a popular I/O interconnect standard widely used by many PC and peripheral manufacturers and will appear in SOC applications as a legacy interconnect.

The PCI 2.1 specification document requires the PCI protocol to satisfy a "producer/consumer" transaction ordering property. The published PCI 2.1 protocol violates the producer/consumer property due to a phenomenon called "completion stealing" as pointed out by Corella. Corella [Cor96] proposed the inclusion of local master IDs to prevent completion stealing and bring the protocol into compliance with the producer/consumer property.

While protocols over branching networks are increasingly important in DSM or MP applications, little reasoning support exists for reasoning about them. The majority of recent verification work in unbounded, or parameterized, processes has focused on linear [KMM+97,CJLW99], rather than branching, topologies. Protocols over branching networks present unique challenges to model checking, theorem proving and hybrid techniques.

Our first attempts to solve this verification problem [MHG98,MHJG00] relied on theorem proving, and later an informal combination of theorem proving and model checking, applied *directly* to the PCI protocol. We found that the direct theorem proving approach was too difficult due to the complexity of identifying and proving auxiliary invariants (as will be discussed later). We also found that model checking applied directly to limited finite instances of the protocol quickly became mired in capacity issues (as is discussed in [MHJG00]). Our current solution *reduces* the PCI 2.1 protocol to a an abstract protocol, called PCI′, which can be exhaustively verified using the Murϕ *[ID96] model checker. The refinement proof relates the abstract model to the concrete* model and was carried out using the PVS [ORS92] theorem prover.

In this paper, we present our higher-order logic formalization of PCI and PCI′along with a sketch of the refinement proof. The next section presents the specific verification problem in more detail and outlines our overall solution. Section 3 discusses the definitional theories of both protocols and the refinement proof. Section 4 briefly discusses the use of the Murϕ model checker to verify the reduced model. After presenting the details of the refinement proof and how the proof was carried out, we close with a discussion of how to eliminate the need for such a refinement proof.

2 The Problem and a Solution

We give an overview of the PCI protocol and the requirements on the topology of a PCI network. We then describe the PC transaction ordering property specified in the PCI 2.1 specification. An acyclic PCI network is an acyclic hypergraph of agents and bridges connected by busses such that there exists a unique path between any two agents. Agents are connected to one bus while bridges are connected to two busses. Agents and bridges each contain two queues: one in each direction. The other queue in a bridge or agent is referred to as the "opposite queue". The opposite queue of the opposite of q is q.

A PCI network supports two types of transactions: posted and delayed. Posted transactions are unacknowledged transactions that can not be deleted and can not be passed. Delayed transactions are acknowledged transactions that can be dropped before being committed and can be passed at any time by other delayed or posted transactions. Committed delayed transactions are delayed transactions that have been attempted at least once, but not necessarily latched, on a local bus. Delayed transactions leave a trail of committed copies of themselves in every bridge through which they pass. The response to a delayed transactions is called its completion. Completions travel back from target to source following the trail of delayed transaction copies. Completions can be deleted and passed—except for delayed write completions, which can not be passed. A posted transaction is considered complete on a bus as soon as issued, while a delayed transaction is considered complete after its completion has returned to the local bus. The transaction reordering and deletion rules are intended to prevent deadlock while preserving the transaction ordering required by the PC property.

The PCI specification requires that PCI bus/bridge networks have the PC transaction ordering property. For the purposes of PCI, the PC property is stated as follows. If the following preconditions are satisfied:

(1) An agent, the producer (*Prod*), issues two write transactions: W_{Data} (a posted or delayed write transaction to the address *Data*) followed by W_{Flag} (a posted or delayed write transaction to the address *Flag*),

(2) An agent, the consumer *Cons* issues two Delayed Read Request transactions R_{Flag} followed by R_{Data},

(3) W_{Flag} is committed on the originating bus after the completion of W_{Data} on the originating bus,

(4) R_{Data} is committed on the originating bus after the completion of R_{Flag} on the originating bus,

(5) R_{Flag} is completed on the destination bus after W_{Flag} completes on the destination bus, and

Then, assuming no other agents write to the data address, the value returned by *Cons* R_{data} is the value written by the producer's W_{data}. The central problem in this case study then is to show that all execution traces of all PCI networks satisfy the PC property.

Initially, we applied theorem proving directly to the PCI protocol. The standard approach to showing an invariant over an infinite state protocol using a theorem prover is to break the protocol into *steps* and show that each step preserves the invariant. We represent the steps using *inference rules*. The precondition of a rule describes the conditions under which the protocol step is enabled; the postcondition of a rule describes the effects of a protocol step. For each inference rule r_i we show that if state s satisfies the invariant *inv*, then the all possible states created by r_i applied to s also satisfies *inv*:

$$\forall s, r_i.inv(s) \Rightarrow inv(r_i(s))$$

The problem is that $inv(s)$ might not constrain s enough so that we can show *inv* on $r_i(s)$. This requires identifying and proving another set of invariants, which we call *auxiliary invariants*, so that, for some set of auxiliary invariants aux_i, we have the following:

$$\forall s, r_i.aux_0(s) \wedge aux_1(s) \ldots aux_i(s) \wedge inv(s) \Rightarrow inv(r_i(s))$$

In practice, identifying and proving each of the auxiliary invariants can be difficult and time consuming.

For the PCI case study, we identified a set of invariants which we believed could be used to show the transaction ordering property invariant. We created a hypothetical proof tree showing the relationships between the auxiliary invariants and the primary invariant that filled a 2 meter by 1 meter white board. After approximately 3 months of proof effort by an experienced PVS user, only a few of these invariants had been proven. One of those invariants read "if a transaction exists in a bridge, then there exists a previous state in which that transaction was created" The proof of this invariant required approximately one and a half weeks of effort.

Rather than continue our brute-force PVS proof of the PC property, we chose to define an abstraction that would reduce the original unbounded problem to one suitable for model checking. The abstraction of PCI eliminates all sources of unboundedness in the original problem resulting in the abstracted PCI′ protocol. The abstraction has four components:

- Reducing arbitrary instances of the PC property over PCI networks to one of four[1] reduced networks. The reduction is done by keeping only *significant paths* in the abstract network. Significant paths are the paths between PCI devices required by an instance of the PC property. The four network classes created by this reduction were identified by a proof in PVS [MHJG00].
- Coalescing bridges in a significant path. Bridges are coalesced by concatenating their contents. The number of bridges in a significant path is unbounded, coalescing bridges maps all paths of n bridges to the same abstract path.
- Ignoring all transaction except *significant transactions*. Significant transactions, like significant paths, are the transactions required by an instance of the PC property.
- Ignoring all but the newest committed copy of a significant transaction. Recall that delayed transactions leave committed copies as they travel from bridge to bridge. Since a path may contain n bridges, an abstract path might contain up to n copies of a significant committed delayed transaction. Since n is unbounded, the number of states in an abstracted PCI network would also be unbounded.

While the preceding abstraction results in a small model with under 6,000 states, the abstraction is sufficiently convoluted that one should not be easily convinced that the properties of PCI′ apply to the PCI model. We formally relate PCI′ to the concrete PCI model by showing that the concrete model is a trace inclusion refinement of PCI′. This relationship was established using a PVS proof.

3 Refinement Proof

Trace inclusion refinement requires showing that for every concrete PCI trace, there exists an abstract PCI′ trace such that each state in the abstract trace is equal to the abstraction of the corresponding concrete state [AL91]. Trace inclusion refinement weakly preserves safety properties. In the context of PCI, for each concrete trace, σ, we need to show that there exists an abstract trace, σ_A in PCI′, such that the following relationship holds:

$$\begin{aligned}\forall \sigma \in \text{PCI}.\ \{\sigma = (A_0, C_0), (A_1, C_1), (A_2, C_2) \ldots (A_n, C_n) \\ \Rightarrow \exists \sigma_A \in \text{PCI}'.\sigma_A = A_0, A_1, A_2 \ldots A_n\}\end{aligned}$$

where each next state is created by applying a PCI transition to a particular transaction in a particular network. Each abstract state A_i in σ is created by applying the abstraction function to concrete state C_i. The refinement relationship will be shown by induction on the length of σ. The abstraction function includes the network symmetry reduction.

The proof is outlined in Figure 1. The states in the concrete trace are shown on the top of the figure while the states in the abstract trace are shown at the bottom. The abstract and concrete traces are related by the abstraction function, α. The basis case is shown on the left. It requires us to show that the abstraction of the initial concrete state is equal to the abstract initial state. The inductive hypothesis assumes that for any trace of length n, there exists some abstract trace, also of length n (PCI′ includes a "no-op" transition to allow stuttering), such that the abstract and concrete states are related by the abstraction mapping. The inductive step requires us to show that

[1] Previously, we claimed we could reduce the problem to three networks. This incorrect claim is due to a misinterpretation of the theorem proven about PCI networks

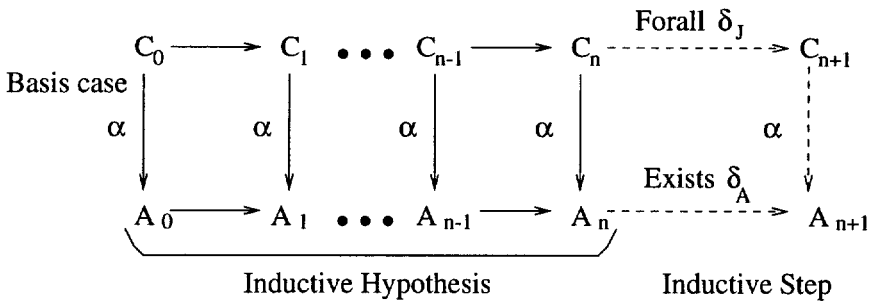

Fig. 1. Outline of proof that PCI is a refinement of PCI′.

for every application of every concrete rule, denoted δ_J, there exists an application of an abstract rule, δ_A, such that the abstraction of the next concrete state equals the next abstract state.

The refinement proof was carried out in the PVS theorem prover using formalizations of the PCI and PCI′ models and the abstraction. The inductive step of the refinement property is stated in PVS[2] as:

1

```
refine_p_compl : THEOREM
FORALL (R: concrete.rule, cPos:concrete.index,c:concrete.state) :
R`pre(cPos,c) AND
(not (osig (trans_at (cPos,c))))
IMPLIES
(EXISTS (a : nat) : aRules(j)`pre(index_abs(cPos,c),state_abs(c)) AND
                    aRules(j)`action(index_abs(cPos,c),state_abs(c)) =
                    state_abs(R`action(cPos,c)))
```

The theorem reads that "forall indices, *cPos*, in all PCI states, *c*, if the transaction at *cPos* satisfies the preconditions to a concrete rule *R*, then there exists a PCI′ rule *a* such that the abstraction of *cPos* in state *c* satisfies the preconditions of rule *a* and the PCI′ state created by the action of rule *a* applied to the abstraction of *cPos* in state *c* is equal to the abstraction of the PCI state created by the action of *R* applied to *cPos* in state *c*." The refinement theorem was proven for each of the rules representing the PCI execution steps.

Both PCI and PCI′ protocols were formalized using sets of inference rules to define their operational semantics. Inference rules are represent as records with three fields: a "pre" field for modeling the precondition as a predicate, a "post" field which maps states to next states and a "name" field which was included to aid readability. We begin by giving the formalization of the PCI p-latch and discussing the various effects PCI p-latch might have on the abstract trace. We then give the PCI′ rule needed to model the effects of PCI p-latch on the abstract trace.

[2] All PVS excerpts slightly modified to enhance readability

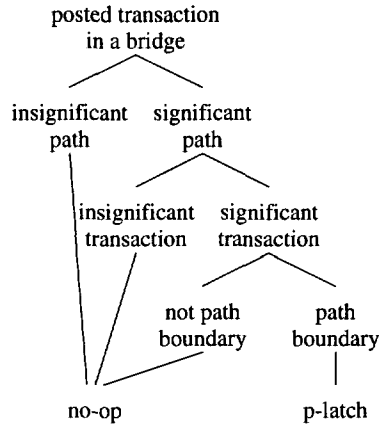

Fig. 2. Case analysis in p-latch refinement proof.

2

```
(# name := "p latch",
   pre := lambda(cPos:concrete.index,c:concrete.state):
     (head_of_queue (cPos)) AND
     (not (final_queue (cPos,c))) AND
     (posted(cPos,c)),
   action := lambda(cPos:concrete.index, c:concrete.state):
     let nextq = next (cPos,c),
         t = trans_at (cPos,c),
         cPos2 = (side(cPos),nextq,c'net(side(cPos),nextq)'length)
     in
     insert (cPos2,t,(delete (cPos,c)))
#)
```

If a posted transaction has reached the head of a queue in a bridge, the p-latch rule moves the transaction to the tail of the next queue. The "pre" field for p-latch checks that the transaction:

- has reached the head of its queue,
- is not in its final queue and
- is a posted transaction.

If the precondition is satisfied, then the function in the "post" field creates a new reachable state in which the posted transaction at "cPos" has been deleted and inserted at the tail of the next queue (labeled "cPos2" in the definition).

Rule p-latch is parameterized by a location, or index, and a state. For a concrete state c, we assume c satisfies the preconditions to p-latch and perform a case analysis on the concrete index, $cPos$. The case analysis on $cPos$ partitions the effects of p-latch on the abstraction of the next state. The structure of the case analysis is outlined in figure 2.

The first case split is on whether or not $cPos$ lies on a significant path. If $cPos$ does not lie on a significant path, then the transaction at $cPos$ is not a significant transaction and is discarded by the abstraction. In this case, PCI rule p-latch has no effect on the abstraction of the next state and the PCI′ no-op rule models the effects of p-latch. In t he next case split, we assume $cPos$ lies on a significant path and split on whether or not the transaction at $cPos$ is a significant transaction. If the transaction at $cPos$ is not a significant transaction, then it is also discarded by the abstraction; and p-latch does not affect on the next abstract state so we use the PCI′ no-op rule here as well.

If the transaction at $cPos$ is a significant transaction, then the next case split is on whether or not the bridge containing $cPos$ lies on a path boundary. If the transaction does not lie on a path

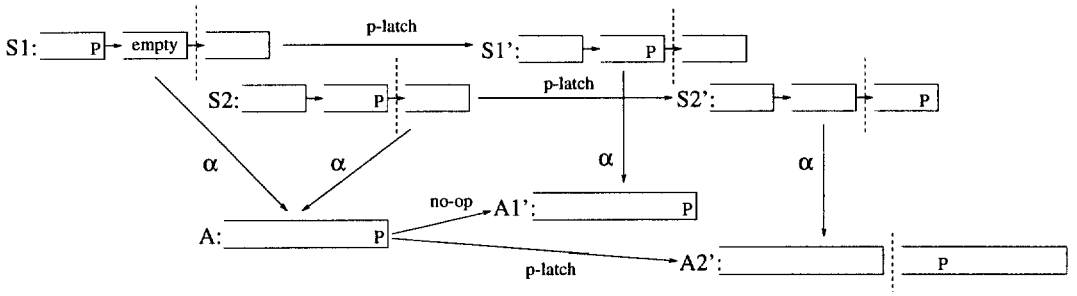

Fig. 3. Non-determinism in p-latch PCI′ transition.

boundary, then moving the transaction from the head of one bridge to the tail of the next has no visible effect in the abstraction because bridge contents are coalesced. The PCI′ no-op rule applies here as well. But if the transaction is at the head of a bridge on a path boundary, then moving the transaction to the next bridge moves the transaction to the next path in the next abstract state. This potential effect on the abstract trace is mirrored by the PCI′ p-latch rule, given below:

3

```
(# name := "p latch next",
   pre := lambda (aPos : abstract.index, a :abstract.state) :
(head_of_path (aPos)) AND
(not (final_path (aPos,a))) AND
(posted(aPos,a)),
   action := lambda (aPos : abstract.index , a : abstract.state) :
let t = trans_at (aPos,a),
    next_aPos = (side(aPos),path(aPos)+1,
    a'net(side(aPos),path(aPos)+1)'length)
in
if (not (final_path (aPos,a)))
then
insert (next_aPos,t,(delete(aPos,a)))
else a
endif
#)
```

The PCI′ p-latch rule is similar to the PCI p-latch rule. The precondition checks that a posted transaction has reached the head of a queue. The postcondition deletes the transaction out of the current queue and inserts it into the next queue.

However, the PCI′ no-op rule may also apply to states which satisfy the preconditions to the PCI′ p-latch rule. This non-determinism is created by the information lost when bridges are coalesced to form paths. For example, consider an abstract state a in which a posted transaction p appears at the head of a path. If we take the pre-image of a under the abstraction, then any state in which a posted transaction appears as the final significant transaction in a set of bridges may appear at the head of a path in a. This is illustrated in figure 3. Two concrete state fragments, $S1$ and $S2$, appear at the top of the figure and an abstract state fragment, A, appears at the botton. States $S1$ and $S2$ both contain a single significant posted transaction in one of three queues. A path boundary is indicated by the dashed line between queues 2 and 3. In $S1$, queue 2 is empty. Under the abstraction, $S1$ and $S2$ map to the same abstract state fragment, A.

Next we apply a p-latch transition to states $S1$ and $S2$ to create the next states $S1'$ and $S2'$. In $S1'$, the posted transaction now appears in queue 2, but in $S2'$ the posted transaction has crossed

the path boundary. Despite the fact that $S1$ and $S2$ map to the same abstract state, $S1'$ and $S2'$ map to different abstract states, labeled $A1'$ and $A2'$. In the abstract trace, the transition from $S1$ to $S2'$ is mirrored by a PCI′ no-op transition; while the transition from $S2$ to $S2'$ is mirrored by a PCI′ p-latch transition.
In this case, the non-determinism is easily handled by the no-op rule in PCI′. The no-op rule has a precondition of "true" and is thus always enabled. However, the preconditions of p-latch in PCI′ allow more behaviors than the PCI version of p-latch. For example, the posted transaction in $S1$ from figure 3 may move to the next path in $A1'$. This is clearly not possible in a single execution step from $S1'$ because the transaction must first pass through bridge 2 before moving to the next path. The additional behaviors allowed by PCI′ are acceptable for a trace inclusion refinement, but may lead to a false negatives.
The refinement proof for the PCI p-latch rule is completed using the PCI′ rules p-latch and no-op with a case analysis similar to that in figure 2. The completed PVS proof of refinement for p-latch uses approximately 100 proof commands.
Similar case by cases analyses were completed for the remaining 9 PCI transition rules. However, non-determinism appears in more complex forms in the refinement proofs for other PCI rules. In the case of the delayed transaction completion rule, non-determinism created by information lost in the abstraction can not be modeled by no-op transitions, as is the case for p-latch. In the case of delayed completions, the non-determinism requires the creation of additional rules in the PCI′ model. These rules also allow more behaviors in PCI′ than PCI.
The refinement proof was developed in one month's time by a single experienced PVS user and requires about 1000 PVS proof commands (determined by taking the number of lines in the proof file and dividing by two to account for pretty-printing of right parentheses). The PCI′ and PCI protocols were modeled using definitional theories of about 500 (determined using `wc -l`)lines each. The abstraction was modeled using an axiomatic theory containing about 700 lines and 51 axioms. We chose to axiomitize the effects of the abstraction, rather than defining the abstraction and reasoning about the effects, to save time in the development of the theory. While most axioms describe properties of the abstraction, a few are PCI invariants needed for the refinement proof. All PVS theory and proof files are available online at [Jon00].

4 Model checking PCI′

The same inference rules used to define the operational semantics of PCI′ in PVS were used to create a model of PCI′ for the Murϕ model checker. Both the PVS and the Murϕ model were created manually. The rule-based notation used in the input language of the Murϕ model checker leads to a relatively trivial manual translation between Murϕ and PVS models. For comparison to the PVS definition of p-latch, we give the Murϕ definition of the p-latch rule below.

4

```
Rule "p latch"
  (posted(trans)) & !(final_path (trans,i))
==>
  begin
    enqueue (network[route(trans,i)], trans);
    delete_trans (path, trans); -- delete from old path.
  end;
```

While the PVS version includes three clauses in the precondition, the Murϕ version includes only two. This is because this Murϕ rule is part of a rule-set which is defined only for the first entry in a path. Hence there is no need to check that the transaction is at the head of a path. We claim that the postconditions of both rules are semantically equivalent.
Using the Murϕ model of PCI′, we can exhaustively check the producer/consumer property for all execution traces in all abstract network classes. The time and number of states required to check each reduced network is given in table 4. No violations were found in any of the networks.

Table 1. Model checking results for PC property on PCI′.

Network	CPU Time	States
A	35.35 sec.	1614
B	18.68 sec.	914
C	12.56 sec.	648
D	51.2 sec.	2690
Total	117.61 sec.	5866

Since all traces in all configurations of the abstract protocol satisfy the PC property and the PCI protocol is a refinement of the abstract protocol, we conclude that all traces of all networks in the PCI protocol also satisfy the PC property.

5 Remarks

This case study combining theorem proving and model checking demonstrates the use of a manually derived abstraction to reduce a difficult infinite state problem to a finite state problem amenable to model checking. The reduction depends on a mechanically checked proof of the refinement relation. While the proof was completed in one month, the amount of effort and expertise required (not to mention the 51 axioms describing the abstraction) to complete the proof suggest that eliminating the need for a refinement proof is the next logical step in our ongoing research.

Based on our experience thus far, we believe that the effort required to verify a difficult property on a complex system can be expended in either the abstraction or the verification. In our previous work on the unreduced PCI protocol, we expended very little effort on the abstraction and a great deal of effort in the (unfinished) verification. In the work reported here, we have expended very little effort on the verification but a great deal of effort on the abstraction. In our current work, we are aiming for a better balance between the effort required to do the verificaiton and abstraction. Our goal is to simplify the abstraction process by eliminating the need for a refinement proof. We anticipate that this will result in abstractions which are easier to derrive, but harder to verify.

We are currently working to eliminate the need for a refinement proof by using a version predicate abstraction [GS97] for parameterized processes and a structural abstraction over acyclic networks. There are still several obstacles to overcome in this predicate abstraction scheme, but the basic idea is simple. The predicate abstraction technique uses parameterized predicates to represent the states of some or all nodes in a path using existential quantification. The user provides a list of predicates, $\varphi_1^i \ldots \varphi_n^i$, paramerterized by a node i. Two abstract state variables, $\hat{\vartheta}_j^p \hat{\varphi}_j^p$, are then created for each predicate j and each abstract network path p. The $\hat{\vartheta}_j^p$ variable indicates that a node exists in path p that satisfies or does not satisfy predicate ϑ_j depending on the value of $\hat{\vartheta}_j^p$. For example, if $\hat{\vartheta}_j^p \neg \hat{\varphi}_j^p$ appears in an abstract state, then a node exists in path p that does not satisfy predicate φ_j in the concretization of that abstract state. A structural abstraction, similar to the one used for PCI, based on topologically non-isomorphic spanning trees over n-terminal nodes is used to divide the unbounded configuration space of acyclic networks into a finite number of abstract network classes. We then plan to check the resulting abstract system over each network class using state exploration.

The primary obstacle is the use of quantification in the concretization of the abstract state variables. This translates to quantification in the proof obligations in the abstract transition relation. A similar problem involving quantification for predicate abstraction on parameterized proccesses was encountered by Das [DDP99]. However, Das' work was done in an extended model checker and we plan to use a theorem prover. The use of a theorem prover will allow the use of decision procedures, or even interactive proof as a last resort, to deal with the quantification.

References

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[CJLW99] Edmund Clarke, Somesh Jha, Yuan Lu, and Dong Wang. Abstract BDDs: A technique for using abstraction in model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, CHARME '99*, volume 1703 of *Lecture Notes in Computer Science*, Bad Herrenalb, Germany, September 1999. Springer-Verlag.

[Cor96] Francisco Corella. Proposal to fix ordering problem in PCI 2.1, 1996. Accessed April 2000. www.pcisig.com/reflector/thrd8.html#00706.

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

[Gru97] Orna Grumburg, editor. volume 1254 of *Lecture Notes in Computer Science*, Haifa, Israel, June 1997. Springer-Verlag.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In Grumburg [Gru97].

[ID96] C. Norris Ip and David L. Dill. Verifying systems with repplicated components in Murϕ. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 147–158, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[Jon00] Michael Jones. PVS theory files for PCI refinement proof. www.cs.utah.edu/~mjones/pci-pvs.html, 2000.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A Pnueli, and E. Shahar. symbolic model checking with rich assertional languages. In Grumburg [Gru97].

[MHG98] Abdel Mokkedem, Ravi Hosabettu, and Ganesh Gopalakrishnan. Formalization and proof of a solution to the PCI 2.1 bus transaction ordering problem. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design, FMCAD '98*, volume 1522 of *Lecture Notes in Computer Science*. Springer-Verlag, November 1998.

[MHJG00] Abdel Mokkedem, Ravi Hosabettu, Michael D. Jones, and Ganesh Gopalakrishnan. Formalization and proof of a solution to the PCI 2.1 bus transaction ordering problem. *Formal Methods in Systems Design*, 16(1):93–119, January 2000.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1992.

[PCI95] PCISIG. PCI Special Interest Group–PCI Local Bus Specification, Revision 2.1, June 1995.

A Substructural Logic for Formal Verification

Sara Kalvala and Mike Squire

Department of Computer Science, University of Warwick
Coventry CV4 7AL, UK email: {sk,msquire}@dcs.warwick.ac.uk

Abstract. We present a mechanisation of a resource-sensitive logic that allows us to capture the notion of state transitions being logical operations that *consume* resources while generating new ones. These so-called substructural logics have been around for a while; our innovation is in embedding Higher Order Logic within a substructural logic and implementing this system within Isabelle, thus allowing us to exploit this logical system for verification of computational systems. We illustrate our approach with the verification of a run of a simple program.

1 Introduction

Computations involve change: to values assumed by variables, to bits stored in particular memory locations, or more generally to a global *state*. Formal verification of computational systems must employ an adequate mathematical model for systems and changes to systems. Most current verification methodologies do not do this in a direct way, and the complexity of representing change is partly responsible for the perceived difficulty in applying theorem provers in verification. While formal verification is a mature area of research, with a growing industrial intake, it is still limited by the practical difficulty of modelling large representational systems in low-level mathematical notations.

We propose to address this limitation in theorem proving technology by exploiting the development of substructural logics such as linear logic [5], which provide proof systems that naturally address some of the problematic issues associated with the use of more traditional logics in reasoning about dynamic systems. The aim of the project is to examine variant substructural logics and use them for building tools for specification and verification of dynamic systems, both directly and as meta logics. Our emphasis will be on case studies and concrete applications. We work with Isabelle [14], a well-known proof tool, instead of creating a completely new proof system. Isabelle provides a quick route from a formal logic to a proof environment that can be applied directly in verification tasks.

While several toy examples, such as in the "blocks world" or concerning the purchase of cigarettes and chocolates, have been used to demonstrate the applicability of linear logic to capture the concept of resources in the real world, linear and other substructural logics have not yet been successfully used to model temporal properties, which are necessary for verification of computing systems

(software and hardware). Traditionally, such systems are specified using variants of temporal logics, which capture many of the notions of behaviour over a possibly infinite computation; however, they are not ideal for describing and reasoning about *each* step of the computation.

Linear logic can be seen to model single computation steps directly, as it is often claimed that linear logic captures the notion of state, and a single step in a computation is a change of state. If a computation step *consumes* certain properties of state and generates new properties, it can be represented by a linear formula

$$P_1(s) \multimap P_2(s)$$

where s is the representation of state and P_1 and P_2 are predicates characterising each state. Programs consist of *actions*, which can either be single commands represented as above or structuring constructs.

2 Substructural logics

A common starting point for describing formal logics is Gentzen's *Sequent Calculus* [3]. A sequent, written $P \vdash C$, is *valid* if the consequences C can be proved from the premises P. A logic, or a deductive system, is a set of schematic rules that state the conclusion of each rule is valid if all the hypotheses are valid.

The rules of traditional logic systems can be divided into two groups: structural rules, which manipulate the structure of a sequent (by duplicating an existing premise, discarding a premise, or even changing the order of premises) and rules that define the meaning of connectives of the logic, when they appear within a formula on either the left or right side of a sequent. A typical collection of structural rules is shown in Figure 1. The *exchange* rule states that the order in which premises are stated does not matter, and the *weakening* and *contraction* rules state that the same premise can be discarded or used once or more as necessary for a proof. Often, especially in logics which are not substructural, the structural rules are hidden within the rules of the connectives and thus do not appear explicitly in the deductive system. Most deductive systems of this form could however be reformulated to separate the structural rules from those defining the connectives.

Typically, in what we call traditional logics, the structural rules are always accepted and the logics differ in the rules defining connectives. In substructural logics [3], one or more of the structural rules are restricted. If they were all simply removed, the resulting logic would be rendered too weak to be useful, so expressive power is restored by creating new connectives. Furthermore, modifying the structural rules usually implies that connectives such as the conjunction and the disjunction can have different interpretations, and therefore one usually considers two variations on these connections: the *multiplicatives* and the *additives*.

Substructural logics have been of interest to logicians for a variety of reasons. One of the earlier substructural logics is the Lambek Calculus [10], developed for representing sentences in natural languages. Relevance logic rejects *weakening*

$$\frac{}{A \vdash A}\ Identity \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\ Exchange \qquad \frac{\Gamma \vdash B \qquad B, \Delta \vdash C}{\Gamma, \Delta \vdash C}\ Cut$$

$$\frac{\Gamma \vdash A}{\Gamma, I \vdash A}\,(I_{\mathcal{L}}) \qquad \frac{}{\vdash I}\,(I_{\mathcal{R}})$$

$$\frac{}{\Gamma, 0 \vdash A}\,(0_{\mathcal{L}}) \qquad \frac{}{\Gamma \vdash 1}\,(1_{\mathcal{R}})$$

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\,(\otimes_{\mathcal{L}}) \qquad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\,(\otimes_{\mathcal{R}})$$

$$\frac{\Gamma \vdash A \qquad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C}\,(\multimap_{\mathcal{L}}) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\,(\multimap_{\mathcal{R}})$$

$$\frac{\Gamma, A \vdash C}{\Gamma, A \& B \vdash C}\,(\&_{\mathcal{L}}) \quad \frac{\Gamma, B \vdash C}{\Gamma, A \& B \vdash C}\,(\&_{\mathcal{L}}) \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \& B}\,(\&_{\mathcal{R}})$$

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}\,(\oplus_{\mathcal{L}}) \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\,(\oplus_{\mathcal{R}}) \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}\,(\oplus_{\mathcal{R}})$$

$$\frac{\Gamma, [t/x]A \vdash B}{\Gamma, \forall x.A \vdash B}\,(\forall_{\mathcal{L}}) \qquad \frac{\Gamma \vdash [a/x]A}{\Gamma \vdash \forall x.A}\,(\forall *_{\mathcal{R}})$$

$$\frac{\Gamma, [t/x]A \vdash B}{\Gamma, \exists x.A \vdash B}\,(\exists *_{\mathcal{L}}) \qquad \frac{\Gamma \vdash [t/x]A}{\Gamma \vdash \exists x.A}\,(\exists_{\mathcal{R}})$$

$$\frac{\Gamma \vdash B}{\Gamma, !A \vdash B}\ Weakening \qquad \frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B}\ Contraction$$

$$\frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B}\ Dereliction \qquad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A}\ Promotion$$

($*$ is the usual side condition: x is not free in the context Γ)

Fig. 1. The rules of Intuitionistic Linear Logic

with the intuition that all premises of a sequent must be relevant to entail the consequence relation [3].

Linear logic is a particular substructural logic with restricted weakening and contraction [5]. The deductive system naturally leads to an interpretation of premises as 'resources' that are consumed and can normally be used only once. Expressive power is added by introducing *exponentials*, which provide a mechanism for controlled use of weakening and contraction. The implication connective in linear logic (also called linear implication) also reflects consumption of the premise of a deduction.

Linear logic has been the target of much interest in the research community, particularly with regards to theoretical issues such as its semantics and time complexity. A lot of the development in the proof theory of linear logic builds on the concept of *proof nets*, graphical links that are added to existing proofs to illustrate the creation and consumption of resources.

The notion that some valid formulae are consumed in the proof process and are therefore transient suggests that some substructural logics may be natural formalisms for expressing dynamic properties, which require the invalidation of old properties and the validation of new properties over the computation. In fact, they have already been proposed and used as a metalanguage to describe models of computation, with varying degrees of success [6, 2, 15, 1].

3 A substructural logic for verification

From a logical view, most of the computations we might wish to examine can be represented by an initial state, *init*, being mapped to *final* by zero or more computation steps involving *prog*. As explained above, the added value linear logic brings by formulating the computation as $init, prog \vdash final$ is the natural consumption of resources. Thus, in the process of performing a proof, out-of-date information is consumed in favour of new results, helping keep the description of state consistent.

It is of course true that a pure substructural logic in itself is not sufficient to represent non-trivial computations. Real programs have typed variables of with associated values. Additionally it may be necessary to represent induction and define numerous distinct data types to properly translate a program. A logic for the verification of programs should therefore be able to represent these concepts. Here we propose to incorporate the power of HOL in a substructural logic by embedding HOL formulae as its atomic propositions.

3.1 Embedding HOL in (many-sorted) intuitionistic linear logic

The power of HOL within a linear formulation is possible by means of an embedding function that maps objects in HOL to propositions in intuitionistic linear logic. This has the type $prop_{HOL} \rightarrow prop_{LL}$ and gives an almost complete separation between the HOL and linear worlds. This logic, which we call ILL[HOL], is propositional intuitionistic linear logic whose atomic propositions

are parameterised by HOL formulae. Provided this embedding is done with care linear sequents can express $x = 1$ as a resource in a computation. Quantification must however be dealt with differently. To preserve the linearity of a quantified statement a linear quantifier is required, but, as in the case of assignment, quantification must be over HOL objects. Universal and existential quantification shall be represented by $\forall_{Lin}$ and $\exists_{Lin}$ respectively to indicate they are "one use only" linear quantifiers. Introduction of these quantifiers means that, in terms of semantics, we no longer have pure intuitionistic linear logic, but a many-sorted variant. This is because the objects quantified over exist in HOL.

Additionally a rule to deal with the case where something is trivially true in HOL is required. The statement $True : prop_{HOL}$ is meaningless in the linear world so the embedding function should map it to a sensible equivalent in linear logic. It is important to observe that a truth statement in HOL should not lead to resource consumption. The solution is to consider something that is a tautology in HOL to be a tautology in linear logic and consider *True* in HOL to be equivalent to I in linear logic. The rule for this is:[1]

$$\frac{\vdash_{HOL} True : prop_{HOL}}{\vdash_{LL} I : prop_{LL}}\ hol\_true$$

In practice, this rule is often required when dealing with the clause of a conditional which in general should not consume resources.

It is also necessary to properly define how to represent $x = 1$, where x represents not a value but a *location* in memory. Although $x = 1$ is a visually pleasing way of asserting a statement about the state of the system, in practice it introduces inconsistencies into our proofs because of the way substitution is dealt with in HOL. HOL represents facts in a very denotational way whereas the linear world is dealt with in an operational manner. Thus the assertion that $x = 1$ will have the effect of instantiating x with 1 in every place it occurs, even in subformulas which are intended to assert some "new" statement, $x = a$, after $x = 1$ is consumed. This is illustrated in the statement below where substitution transforms

$$x = 1, \forall_{Lin} n.(C \otimes x = n) \multimap x = n + 1$$

into

$$x = 1, \forall_{Lin} n.(C \otimes 1 = n) \multimap 1 = n + 1$$

Clearly this is inconsistent since on one side of the implication n must be instantiated with 1 and on the other, 0. In our original implementation we created a function $\mathcal{V}$ of type $loc \rightarrow nat$ and continued to use equality. Thus $\mathcal{V}\ x = 1$ was the statement that 1 was in location x. However this had similar problems and ηecessitated a messy substitution rule:

$$\frac{\Gamma, \mathcal{V}\ x = n, P(n) \multimap Q \vdash G}{\Gamma, \mathcal{V}\ x = n, P(\mathcal{V}\ x) \multimap Q \vdash G}\ hol\_subst$$

[1] In the rest of the paper we will omit the annotation for $\vdash$ as we will be using $\vdash_{LL}$.

Here only the left side of the implication is substituted and the right is deferred till after application of the implication rule. It turns out that predicates and not equality are the most effective way to assert statements about the state of a system. Thus, as a resource, $\mathcal{V}\ x\ n$ will be used to assert that location x has value n. For the moment $\mathcal{V}$ will be considered as a function of type $loc \rightarrow nat \rightarrow prop_{HOL}$, though in general we need not restrict it to natural numbers (we can think of $\mathcal{V}$ being of type $loc \rightarrow a' \rightarrow prop_{HOL}$ and having a specialisation of this function for the natural numbers). Thus x is a location, type *loc*, and n is of type *nat*. Using this, the inconsistent example above is reformulated as:

$$\mathcal{V}\ x\ 1, \forall_{Lin} n.(C \otimes \mathcal{V}\ x\ n) \multimap \mathcal{V}\ x\ (n+1)$$

This predicative notion of state makes our implementation more robust by both guaranteeing that x is necessarily the same on the left as on the right of a linear implication and avoiding the inconsistencies introduced by using equality.

Correctness and generality of embedding The embedding of HOL in linear logic we now have keeps the two logics quite separate apart from three 'bridges':

1. the mapping of tautologies from HOL to multiplicative unit in linear logic;
2. mapping of quantifications from one logic to the other;
3. using unification to instantiate formulas of the form $\forall_{Lin} a.\mathcal{V}\ x\ a \multimap \mathcal{V}\ x\ (a+1)$ where there exists a resource of the form $\mathcal{V}\ x\ n$ in the context.

The tautology mapping is straightforward. Quantification is introduced by considering ILL to be a many sorted variant. In its current form, the usage of unification is also straightforward, however further work is required to generalise it such that it can be applied to subformulas of larger HOL terms.

Apart from these three cases, we have linear and higher order worlds imported directly. Of course, we may need to establish more links when we want to integrate proof strategies for these two logics.

3.2 Representation of programs in ILL[HOL]

A translation of common programming constructs such as assignment, choice, conditionals and iteration into linear logic is necessary if it is to be used as a basis for verification.

The price paid for using a substructural logic like linear logic is that all truth values (often referred to as resources) must be accounted for. Particular care must be taken not to throw away assertions when processing the guard of a conditional if their value is required elsewhere. Conversely a careless program might reach a state in which two distinct assertions about the value of a variable exist.

We suggest, however, that if the translation is adhered to correctly, these events should not occur in practice.

- Assignment in the computational world is similar to linear implication in that to assign a value to a variable, or location, the old value must be forgotten. Linear logics' notion of resource consumption is an obvious way to "forget" about assertions in this way. The general form for `x := f(x)` is $\forall_{Lin} n.\mathcal{V}\ x\ n \multimap \mathcal{V}\ x\ (f(n))$. Universal quantification over n serves to generate a resource on the left of the implication which matches an assertion of the form $\mathcal{V}\ x\ v$ and consumes it. This was alluded to in in Section 3.1.
- Non-deterministic choice is represented by additive conjunction on the left and additive disjunction on the right of a sequent (Figure 1).
- If-then-else translates in much the same way as in classical logic. However, care must be taken to duplicate consumed resources still required on the right side of the implication. This often occurs in practice since instantiation of variables whose values are asserted as resources is often required to process conditions. For example `if x > 0 then A` will translate as $\forall_{Lin} n.\mathcal{V}\ x\ n \otimes n > 0 \multimap \mathcal{V}\ x\ n \otimes A$ with $\mathcal{V}\ x\ n$ duplicated on the left and right of the linear implication.
- $!(A \multimap B)$ encodes iteration. A should be strong enough to fail when the loop is finished in addition to containing any resources requiring consumption or to instantiate B.

4 A GCD algorithm

The GCD example below exemplifies the use of HOL embedded in ILL. Here, as explained above, x and y are of type *loc* and the statement $\mathcal{V}\ x\ n$ asserts that the value at location x is n, or more simply, that the value of the variable x is n. The sequent is merely a implementation of a naive difference algorithm in ILL[HOL]. It is important to note that we are not proving this to be a correct implementation of a GCD algorithm. However assuming we made no mistakes and it is indeed a correct implementation, a proof will exist for the case when a, b and *gcd* are mutually correct (i.e. *gcd* is the greatest common divisor of a and b) and there will be no proof where this is not the case.

$$\begin{array}{c} \mathcal{V}\ x\ a \otimes \mathcal{V}\ y\ b, \\ !(\forall_{Lin} v_1\ v_2\ a\ b.\ \begin{pmatrix} \mathcal{V}\ v_1\ a \otimes \mathcal{V}\ v_2\ b \\ \otimes\ a \leq b \otimes 0 < a \end{pmatrix} \multimap \begin{pmatrix} \mathcal{V}\ v_1\ a \\ \otimes\ \mathcal{V}\ v_2\ (b-a) \end{pmatrix}) \\ \vdash\ (\mathcal{V}\ x\ 0 \otimes \mathcal{V}\ y\ gcd)\ \oplus\ (\mathcal{V}\ x\ gcd \otimes \mathcal{V}\ y\ 0) \end{array}$$

Let us see what this means: the initial state for the computation is specified by assigning the values a and b to x and y, respectively. The guard of the loop ensures it is only enabled if both values are greater than 0. It is also important to note that we quantify over both values and locations in the loop. The algorithm could be equivalently encoded as a choice of two symmetric actions using additive conjunction, one for when the value at x was less than or equal to that at y and another for the reverse. The implementation offered here is more compact and does not require x and y to be stated explicitly in the program by the use of universal quantification. Finally, the desired final state of the computation is

reached when either x or y becomes 0, in which case the other variable contains the calculated *gcd*.

The description of the program illustrates the some of the translation schemes proposed in Section 3.2: an initial state, a program loop, and a conclusion that serves as a final state.

5 Implementation in Isabelle

5.1 Combining ILL and HOL in Isabelle

The first step in implementation was to create a modified version of `Sequents` based on Curried functions (rather than the default of bracketed predicates). This is called `CSequents` and is necessary to allow `Sequents` (and thus `ILL`) to be combined with the existing `HOL` implementation. This marriage is achieved using an auxiliary theory called `HOLSequents` which combines `CSequents` with `HOL` and defines the embedding function as described below and in Section 3.1. `HOLSequents` is the base theory for `HOL_PILL`. `HOL_PILL` stands for HOL and Propositional Intuitionistic Linear Logic and is a very slightly modified version of the existing `ILL` theory is Isabelle [8]. Finally, `ILLHOL` (which corresponds to ILL[HOL]) inherits `HOL_PILL` and defines the linear quantifiers $\forall_{Lin}$ and $\exists_{Lin}$.

The most important aspects of the implementation of ILL[HOL] are:

- The embedding function called `HolProp` : $bool \rightarrow lino$ which maps HOL objects into the linear world. Currently parsing still necessitates the use of special syntax to encapsulate HOL formulas. Ideally the parser would consider all HOL constructs to bind more tightly than ILL connectives.
- The linear quantifiers $\forall_{Lin}$ and $\exists_{Lin}$. These are based on the quantifiers in `LK` (another theory inherited from `Sequents`) but their implementation adapted to be "one use only" linear quantifiers.
- A theory of locations which implements $\mathcal{V} : loc \rightarrow nat \rightarrow prop_{HOL}$. This actually inherits `ILLHOL` and is therefore not strictly part of the `ILLHOL` theory. This is because we consider the theory of locations to be of a slightly higher level than the embedding of HOL in ILL.

Figure 2 summarises the inheritance structure of the implementation. (Theory numtrans contains some auxiliary pretty-printing for numbers.) Figure 3 shows the two extra axioms necessary for the embedding.

5.2 Implementing the GCD algorithm

The theory of GCD (Figure 4) inherits directly from ILLHOL (as well as the theory of locations). This defines the locations x and y as constants as well as the algorithm as presented in Section 4. The algorithm is split into manageable chunks but essentially follows the $init, prog \vdash final$ model described previously. The values of a, b and *gcd* are also defined here.

This example shows the surface syntax of the embedding. The multiplicative conjunction, linear implication, additive conjunction and additive disjunction

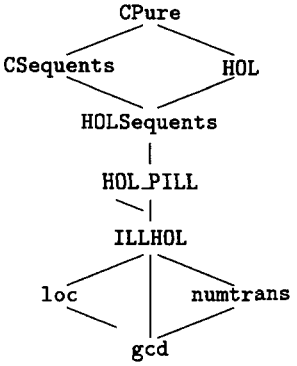

Fig. 2. ILLHOL's inheritance graph

```
(* quantifiers *)
  lallr  "(!!x.$H |- P(x)) ==> $H |- LALL x. P(x)"
  lalll  "$H, P(x), $G |- X ==> $H, LALL x. P(x), $G |- X"

  lexr   "$H |- P(x) ==> $H |- LEX x. P(x)"
  lexl   "(!!x.$H, P(x), $G |- X) ==> $H, LEX x. P(x), $G |- X"

(* linear tautology *)
  "L1"  == "[= True =]"
```

Fig. 3. The axioms required to embed HOL in (many-sorted) ILL

are represented as `><`, `-o`, `&&`, and `||` respectively. The embedding of HOL terms as ILL atoms is done using the `[= ... =]` notation. (While Isabelle's parsing strategy can support doing away with this, we thought it would be useful for now to show clearly the separation between higher-order and linear objects.)

5.3 Proofs of the computation

Having implemented the algorithm in ILLHOL, we proved some runs of this algorithm on particular inputs. Note that we *haven't* proved the correctness of the algorithm! This would involve using a recursive definition of GCD and proving—by induction—that the algorithm works in all cases. We hope to prove this general correctness of the algorithm as a test case when we develop the logic further. For now, we have a prover that works to show that *given* a certain pair of numbers, the computation stops when one of the locations has the correct result.

The basis of the implementation is `gcd_tac` which is summarised in Figure 5. This succeeds if *gcd* is indeed the greatest common divisor of a and b

```
gcd = ILLHOL + loc + numtrans +

consts
  x, y          :: loc

constdefs
  init  :: lino
  "init == [= V x a =] >< [= V y b =]"

  prog  :: lino
  "prog == ! (LALL v1 v2 (a::nat) (b::nat).
              (([= V v1 a =] >< [= V v2 b =] >< [= a <= b =] >< [= 0 < a =])
                                  -o ([= V v1 a =] >< [= V v2 (b - a) =])))"

  f1    :: lino
  "f1 == ([= V x 0 =] >< [= V y (gcd::nat) =])"

  f2    :: lino
  "f2 == ([= V x (gcd::nat) =] >< [= V y 0 =])"

  final :: lino
  "final == f1 || f2"

  a   :: nat  "a   == 99"
  b   :: nat  "b   == 69"
  gcd :: nat  "gcd == 3"
end
```

Fig. 4. The Isabelle GCD theory file

and fails otherwise. gcd_tac works by successively performing stage_tac followed by match_tac. stage_tac contracts the program loop and performs one iteration of the GCD loop (with appropriate back-tracking if the incorrect instantiation of locations is chosen). If stage_tac fails it means one of the values is 0. match_tac tries to match each side of the conclusion to the current state of the context by weakening on the left to remove the program and checking via identity. If match_tac fail then either one or more applications of stage_tac are still required or *gcd* is not the GCD of a and b. Note the guard of the conditionals in the loop prevent the program from entering an infinite loop. The result is that if *gcd* is not the gcd of a and b then both match_tac and stage_tac will fail and thus gcd_tac will fail.

```
val stage_tac =
  EVERY [tensl_tac 1, copy_tac 1, choosel_tac 1, clean_up_tac];

val match_tac = EVERY [weaken_tac 1, chooser_tac 1, rtac identity 1];

fun gcd_tac state =
  (DETERM stage_tac THEN (match_tac ORELSE gcd_tac)) state;
```

Fig. 5. GCD's Isabelle tacticals

Although gcd_tac automates the entire process, it can of course be performed interactively (Figure 6). An interactive proof done in this way shows the state changes involved in the computation with each application of stage tac.

6 Open questions and future work

This paper describes a rather simplistic use of linear logic operators for describing algorithms. There are still many issues to be addressed - such as how to represent concurrency, or data scoping, or non-determinism, or how to perform correctness proofs, or which variant of linear/substructural logic is ideal for this kind of use, from the semantical point of view. It is hoped that this presentation will fuel further discussion and result in collaboration between logic developers and the applications community keen on exploring new avenues to facilitate formal verification.

6.1 Proving temporal properties of computations

Up to now we have only considered how to develop a sufficiently expressive logic into which to translate programs. A logic for verification should not be restricted to reasoning about successive states in a computation but over, for example, the whole computation or even over all computations. In the context of the GCD

```
> goalw gcd.thy gcd_defs "init, prog |- final";

Level 0 (1 subgoal)
init, prog |- final
 1. [= V x 99 =] >< [= V y 69 =],
     ! (LALL v1 v2 a b.
           ([= V v1 a =] >< [= V v2 b =] >< [= a <= b =] >< [= 0 < a =]) -o
           ([= V v1 a =] >< [= V v2 (b - a) =]))
     |- ([= V x 0 =] >< [= V y 3 =]) || [= V x 3 =] >< [= V y 0 =]

val it = [] : thm list
> by stage_tac;

Level 1 (1 subgoal)
init, prog |- final
 1. [= V y 69 =] >< [= V x 30 =],
     ! (LALL v1 v2 a b.
           ([= V v1 a =] >< [= V v2 b =] >< [= a <= b =] >< [= 0 < a =]) -o
           ([= V v1 a =] >< [= V v2 (b - a) =]))
     |- ([= V x 0 =] >< [= V y 3 =]) || [= V x 3 =] >< [= V y 0 =]

val it = () : unit
> by stage_tac;

(* snip *)

> by stage_tac;

Level 9 (1 subgoal)
init, prog |- final
 1. [= V x 3 =] >< [= V y 0 =],
     ! (LALL v1 v2 a b.
           ([= V v1 a =] >< [= V v2 b =] >< [= a <= b =] >< [= 0 < a =]) -o
          ~([= V v1 a =] >< [= V v2 (b - a) =]))
     |- ([= V x 0 =] >< [= V y 3 =]) || [= V x 3 =] >< [= V y 0 =]

val it = () : unit
> by match tac;

Level 10
init, prog |- final
No subgoals!

val it = () : unit
```

Fig. 6. Sample (interactive) Isabelle session with the GCD algorithm

example in Section 4 this might mean having a specification of the algorithm and proving the program implements it. Some classes of proof that may be required are well-known to the temporal logic community:

Possibility: that *eventually* some situation will arise. In our GCD example we are able to prove the property of the final state, but we do not have facilities to state or prove that that final state *will* occur.

Impossibility: temporal logics allow us to state that something will never occur, which is necessary for modelling safety conditions. While proofs in our GCD example fail if given wrong inputs, it is not clear why the proof has failed.

Invariance: a property that is true of every intermediate proof state, modelled by the temporal 'always', which on the surface is quite similar to the linear exponential, but with a very different proof system.

Presently it is unclear how we can represent these properties within the ILL[HOL] theory. The solution might be to add a temporal logic to the implementation. A prototype for linking a temporal logic to a logic of states or *actions* on states is Lamport's TLA [11, 7], where certain proof rules link together properties that are proved at the action level with a general temporal logic. These proof rules link together fairness properties of sub-processes to allow the derivation of fairness properties of the whole process. Unfortunately, the form of some of these rules makes it difficult to use them adequately in a semi-automated and robust proof strategy. For us, more strongly than simply the ability to implement these in the logic is the consideration that the logic should not become too unwieldy to perform proofs. We expect that because linear logic gives us a stronger basis that the typical calculi of state transitions such as given in TLA and UNITY, we may be able to use a simpler and more direct temporal logic.

There are some attempts at adding temporal logic to linear logic [9, 15] but it is unclear how they maintain the identity of each of those components. We hope to build an integration between ILL[HOL] and temporal logic that gives the same level of separation between the temporal and ILL[HOL] worlds that we achieved in ILL[HOL] with the ILL and HOL worlds.

6.2 Choice of substructural logic

We have so far exploited a fairly standard version of intuitionistic linear logic. However, there are several options worth considering:

Classical Linear Logic We have chosen to remain in the intuitionistic fragment of linear logic, not because of a conviction that this is strictly necessary, but to keep our options open pending further work. In terms of representing computations, there is no obvious reason for having multiple conclusions or a classical negation. And as we are keen on developing a *general* method of embedding HOL into a substructural logic, it seemed best to concentrate on ILL.

Non-commutative Linear Logic In our first attempt, we used non-commutative linear logic for two reasons: it seemed to us that the ordering of assertions in the context could imply a linear progression from one state to the other, such that the first assertion would represent the initial state and the last assertion the final state. This would correspond quite closely with the sequentiality implicit in most programs. Another advantage would have been the tightening of proof search mechanisms, as the exchange rule is often a source of difficulty with automated proof search in linear logic. However, the semantics of non-commutative linear logic is more complicated, and trying to attach the notion of time sequence to the context was not straightforward. Retore and Ruet have both done work on capturing computations using non-commutative substructural logics in a semantic sense [16, 17] but we found the syntax of the resulting embedding counter-intuitive for verification purposes. In the end we decided to use the commutative version and describe the flow of control with a *guard*; but the case has not yet been made that non-commutativity is not a desired feature.

Bunched Implications A substructural logic that attempts to re-formulate the fundamental concepts is O'Hearn and Pym's Logic of Bunched Implications [13]. A *bunch* is the structure given to the antecedents of a sequent, which is a tree rather than a sequence. An important feature of Girard's linear logic is the relation between the linear implication with the traditional intuitionistic implication. Girard proposes the mapping of $A \to B$ to $!A \multimap B$ and this certainly makes it possible to map proofs in traditional intuitionistic logic into intuitionistic linear logic. O'Hearn and Pym, however, propose the coexistence of both linear and intuitionistic implications, and do away with the exponential. As linear implications and exponentials make up a substantial part of our formulation of state changes, iterations, and if-then-else control flow, it would be very interesting to see how the same concepts can be expressed in BI. We could also see how a bunched structure for a sequent could help us maintain state-sensitive and persistent statements in an easier way.

6.3 Implementation matters

Work on the implementation continues apace. A translation from a more-user friendly notation to the linear notation would be useful, as linear logic is characterised by an often bewildering array of operators.

The full power of proof mechanisms in Isabelle—namely unification and proof search—has not yet been harnessed. We currently perform only rewriting on the HOL terms individually, but do not really use any of the proof strategies that involve instantiation of logical variables and exploiting backtracking over a set of rules, strategies that feature strongly in any serious Isabelle proof. A generalised solution for substitution will also be required to allow HOL terms separated by linear connectives to share information in a denotation manner.

Proof search in linear logic is itself a subject of extensive study and experimentation. There are results concerning search complexity for various fragments

[12, 4, 18]. Our interest lies in developing a general strategy for proofs in ILL that explores existing proof algorithms for the embedded HOL logic. What we envisage as a key success is a LL-prover (modelled around say `fast_tac`) that calls an appropriate HOL tactic (such as the powerful `blast_tac`) as part of its *subgoaler* mechanism. A further goal would be to develop a general theory (and implementation!) for prover embeddings that follows the idea of logic embeddings.

7 Conclusions

While interactive theorem provers have been used as analysis tools for computational systems for a while, there is still a difficulty in reasoning about state changes, which need an explicit notion of time or state to provide a handle on non-monotonicity of the information involved. A controlled form of deduction such as in linear logic might be useful, and in fact linear logic has always been proposed as a model for concurrent systems, and many researches have looked at the semantic aspects of embedding formalisms such as π-Calculus, Petri Nets, etc in linear logic. Practical analysis of problems (i.e. verification) using these correspondences is the obvious next step. However, this would involve building a complete formalisation of every domain of interest, something that can take years, as has been illustrated by the various provers for Higher-Order Logics in existence.

What we propose and have been developing is a combination of higher order and linear logics, so that all the theories and tools for verification already developed can be exploited to handle the most intensive part of the verification effort, such as numerical operation correctness, case analysis, induction over datatypes and so on. By associating these results with *linear* operations from a substructural logic, we make sure these results do not 'overstay their welcome' when there is a change in state, which would result in inconsistencies. The typical solution is to consider each variable as a function from time or state to value. By using linearity, we get rid of this extra clause in the descriptions. The resulting assertions may look a bit more complicated, and further work will be needed to develop the right surface syntax, but the proofs themselves will hopefully be easier to find and faster to execute.

Acknowledgements Valeria de Paiva from Xerox Parc and Maria Emilia Maietti and Eike Ritter from the University of Birmingham have helped us through the semantic and logical aspects of the embedding. This work is funded through EPSRC Grant no. GR/L90538.

References

1. Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

2. Carolyn Brown, Doug Gurr, and Valeria de Paiva. A linear specification language for Petri Nets. Technical Report DAIMI PB ff 363, Aarhus University, 1991.
3. Kosta Dosen. A historical introduction to substructural logics. In Peter Schroeder-Heister and Kosta Dosen, editors, *Substructural Logics*, volume 2 of *Studies in Logic and Computation*. Clarendon, 1993.
4. D. Galmiche and G. Perrier. Foundations of proof search strategies design in linear logic. In *Symposium on Logical Foundations of Computer Science*, volume 813 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
5. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–101, 1987.
6. Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 1994. To Appear.
7. Sara Kalvala. A formulation of TLA in Isabelle. In E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, number 971 in Lecture Notes in Computer Science. Springer-Verlag, 1995.
8. Sara Kalvala and Valeria de Paiva. Linear logic in Isabelle. In Lawrence C. Paulson, editor, *Proceedings of the First Isabelle Users Workshop*, number 379 in Technical Report. University of Cambridge Computer Laboratory, 1995.
9. Max Kanovitch and Takayasu Ito. Temporal linear logic specifications for concurrent processes. In *Logic in Computer Science*. IEEE, 1997.
10. Joachim Lambek. From categorial grammar to bilinear logic. In Peter Schroeder-Heister and Kosta Dosen, editors, *Substructural Logics*, volume 2 of *Studies in Logic and Computation*. Clarendon, 1993.
11. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), 1994.
12. P.D. Lincoln and N. Shankar. Proof search in first-order Linear Logic and other cut-free sequent calculii. In *Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1994.
13. Peter O'Hearn and David Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999.
14. Lawrence Paulson. *Isabelle: A generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
15. Uday S. Reddy. A linear logic model of state, 1993.
16. Christian Retoré. Pomset logic: a non-commutative extension of classical linear logic. In *TLCA '97*, volume 1210 of *LNCS*. Springer-Verlag, 1997.
17. Paul Ruet. Non-commutative linear logic with mobilities. In *Bulletin of Symbolic Logic*, volume 3-2, 1997.
18. Tanel Tammet. Proof strategies in Linear Logic. Technical Report 70, Department of Computer Science, Chalmers University of Technology, 1993.

Formalization of Isabelle Meta Logic in NuPRL

Pavel Naumov*

Pennsylvania State University
Middletown, PA 17057

Abstract. NuPRL and Isabelle are two general purpose theorem provers. Both of them are based on a version of Constructive Higher Order Type Theory. In an earlier work the author has proposed an informal semantics of Isabelle Meta Logic in an extension of NuPRL Type Theory. Based on this semantics an automated converter that translates Isabelle theorem statements into NuPRL has been developed.
This work presents a formalization of the above semantics in NuPRL. It starts with a deep embedding of Isabelle type and term syntax into NuPRL Constructive Type Theory. Next, two internal NuPRL functions are defined. One of them maps Isabelle types into NuPRL types and the other maps Isabelle terms into elements of appropriate NuPRL types. These two functions provide an interpretation of Isabelle in NuPRL. Finally, interpretations of all Isabelle Meta Logic rules are proven as theorems in some classical extension of NuPRL Type Theory.
This formalization is aimed to provide a more secure foundation for the interaction between two systems.

1 Introduction

This work studies connection between two different proof development environments: Isabelle [14] and NuPRL [2]. Previous works in this area were directed towards creating effective semi-automated procedures for translating mathematical results from one system into another. D. Howe in [5] and [4] defined a shallow embedding of HOL [3] into a classical extension of NuPRL Type Theory, proved its soundness using set-theoretical semantics of NuPRL, and wrote a converter from HOL into NuPRL based on this embedding.

Following Howe's approach, in [11] we defined an embedding of Isabelle Meta Logic into NuPRL and wrote a converter that automates such translation. An important novelty of [12] was in the way the soundness of the embedding is justified. Instead of relying on semantical arguments, it is done in a purely syntactical proof that shows how the translation of any instance of an Isabelle inference rule can be decomposed into several NuPRL inference rule applications. A similar result for Howe's embedding of HOL into NuPRL was independently obtained by J. Meseguer and M.-O. Stehr [8].

* This work was done at the Computer Science Department of Cornell University and it was supported by DARPA grant F30602-98-2-0198

The syntactical soundness proof opens the door to two new directions in the research. First, it becomes feasible to convert formal proofs, not just statements of the theorems, from one system into another. Such proof translation mechanism will actually eliminate the need for justification of conversion soundness, since all translated results will have proofs in the target system. The second opportunity is that now the justification argument can be formalized. Although such formalization does not completely eliminate the need for one system to trust the soundness of another, it provides a more secure foundation for interaction between theorem provers.

In this work we have explored the second opportunity. We have defined a deep embedding of Isabelle syntax into NuPRL Constructive Type Theory, formalized the translation as an internal NuPRL function and proved translations of Isabelle Meta Logic inference rules as theorems in a classical extension of NuPRL.

In addition to providing a secure foundation for the translator, this work also shows that NuPRL, as Type Theory *and* a proof development environment, is mature enough to reflect not just one particular mathematical theory, but an entire formal system.

The paper is structured as follows. Section 2 describes NuPRL formal notations, used throughout the manuscript. Section 3 deals with generic mathematical facts that were added to standard Nuprl 4.2 library in order to accomplish the formalization of Isabelle. Sections 4 and 5 formalize Isabelle type and term syntax correspondingly. Together they define a deep embedding of Isabelle Meta Logic into NuPRL Type Theory. Section 6 defines the interpretation of Isabelle Meta Logic in NuPRL as an internal NuPRL function. The final Section 7 shows that this function maps Isabelle meta inference rules into NuPRL propositions, derivable in a classical extension of NuPRL Type Theory.

The presentation of the material closely follows to corresponding formal NuPRL theories. In particular, all key theorems are reproduced the way they are formalized in NuPRL. For space considerations, proofs are omitted and some long formal definition are presented informally. Complete NuPRL theories in HTML format are available from the author's Web page[1], technical report [10] contains more detailed version of this paper.

2 NuPRL Formal Notations

One of the features that makes NuPRL stand out among the other formal proof development environments is its advanced graphical user interface. Terms are entered into NuPRL not by typing in an ASCII text, but by filling fields in a structural term editor. The same term editor is normally used to display formal theories. Hyperlink mechanism, provided by the editor, allows the reader to inspect the abstraction definition just by clicking on any instance of this abstraction in any term. The editor also relies on an extensive use of abstraction *display forms* to achieve better readability. For example, the default display form

[1] http://www.cs.cornell.edu/home/pavel

for the addition operator $add(x, y)$ is $x + y$ and the one for universal quantifier $all(T, x.P)$ is $\forall x : T.P$. Display forms can be set to hide some of operator parameters if their values are assumed to be obvious. For instance, equality operator $equal(T, x, y)$ which states that elements x and y of type T are equal, is normally displayed as $x = y$. Parameters missing from the operator display form are known among NuPRL users as *hidden* parameters.

Because of mentioned above features, conversion of NuPRL proofs into a paper-based form is not trivial. The standard NuPRL printing function, which converts NuPRL formal theories into LaTeX files, uses abstraction display forms, and, as a result, loses all hidden parameters. Therefore, NuPRL theory printouts do not provide enough information to reconstruct formal definitions and proofs in their original form. Instead, they should be considered as "semi-formal" descriptions of original theories.

In this work we will be incorporating fragments of these formal library printouts in the text. In most cases we will benefit from this approach since it allows to present formal results in a form close to the original. Nevertheless, in some cases these fragments are ambiguous and will need additional comments. In any case, it will be important to keep in mind that these printout fragments *are not* the real formalization. The other facts about NuPRL library structure, important for understanding of the formalization, are

1. Each abstraction, theorem, or display form is normally stored in its own library object. Any object printout contains four fields that show object kind (theorem (`T`), abstraction (`A`), display form (`D`), etc).
2. In most cases there are three objects in the library that correspond to a definition: a display form, an abstraction, and a well-formness lemma. To save space, here we usually reproduce only abstraction object from each definition. In some rare cases well-formness lemmas also will be given.
   Majority of recursive functions are defined in NuPRL using Y-combinator. Since such definitions are hard to read, appropriate abstraction object is hidden in the library and a more intuitive recursive "definition" is displayed instead.

3 Auxiliary Mathematical Facts

Before presenting Isabelle syntax and semantics formalization, we need to state some general mathematical facts and notations that will be used in this work, but which are not parts of standard NuPRL 4.2 library. These facts can be divided into three categories: NuPRL library enhancement, extension of NuPRL type theory by parametrized recursive types, and a classical extension of NuPRL Constructive Type Theory. Below all three categories are discussed in details.

3.1 Library Enhancement

List Equality For any decidable type T, type $T\ List$ is also decidable. In other words, if there is a *boolean* equality relation E on type T, there is one on type $T\ List$. We denote it by as $=_l$. This notation hides parameter E.

Records While defining terms recursively, we need to deal with bindings of bounded variables in partially dis-assembled terms. Such binding is basically a mapping from variable names into type names. Since Isabelle bound variables are named by de Bruijn [1] indices, binding is a function from an initial segment of natural numbers into a type of Isabelle type names. Such functions are commonly called *records*. In type theories which support dependent function types, record is a more general notion than a list, because different fields can have different types. We will be using this feature later.

There are three basic operations on records that are used in this work: *update*, *shift*, and *tail*. Update function `f[n → a]` extends a record f of length n by a new element a, added in the end of the record. This function is similar to appending a single-element list to the end of a given list. Shift function `[s >> f]` extends record f by a new element s, added in the beginning of the record. New element gets number 0 and all other element numbers are incremented by 1. This function is similar to *cons* on lists. Tail function `(f|n)` is similar to the *n-th tail* on lists. It removes first n elements of record f and re-enumerates the rest accordingly.

3.2 Parametrized Recursive Types

Recursive (inductive) type is a widely studied type constructor. It allows for any monotonic function $b : \mathbb{U} \to \mathbb{U}$ to define a new type $rec(X.B(X))$ that can be informally viewed as the minimal solution of the type equation $X = B(X)$. This equation naturally leads us to a more general type constructor that gives the minimal solution to a system of type equations: $X_i = b_i(X_1, \ldots, X_n)$, $1 \leq i \leq n$. Another way to express the same system of equations is to think about variables $X_1, \ldots$ as about a function from index type I into the type universe: $X = \lambda i.b(i, X)$, where $X : I \to \mathbb{U}$, $b : I \to (I \to \mathbb{U}) \to \mathbb{U}$. This approach allows for infinite systems of type equations if type I is infinite. If X_0 is the minimal solution of the above equations, we will denote the application of X_0 to an element i_0 of type I by $parec(X, i.b(X, i)@i_0)$.

Inference rules and semantics for such types have been introduced in [7] and [6]. These rules were added to NuPRL as an extension of Constructive Type Theory by the author in [9].

3.3 Classical Extension of NuPRL Type Theory

Isabelle syntax formalization will be done entirely in Constructive Type Theory. A classical extension is used only to provide a semantics of Isabelle Meta Logic in NuPRL. In fact, since Meta Logic is itself intuitionistic, non-constructivity probably can be avoided.

The main reason for using a classical extension of NuPRL is that we want results, brought over from Isabelle theories, to be useful in NuPRL proofs. Thus, ideally, Isabelle type of meta-propositions o should be mapped into NuPRL type of propositions $\mathbb{P}$. Unfortunately, this mapping is hard to implement since NuPRL type $\mathbb{P}$ has infinitely many elements and Isabelle type o is assumed to

have only two elements: true and false. It means that, for example, Isabelle meta rule

$$\frac{\begin{array}{cc}[\phi] & [\psi] \\ \psi & \phi\end{array}}{\phi \equiv_{prop} \psi}$$

would not be valid when Isabelle propositions are interpreted as NuPRL propositions and Isabelle equality $\equiv$ is translated as NuPRL equality. A possible way around this problem would be to interpret Isabelle type o as NuPRL quotient type $Q = \mathbb{P}//(x, y.x \Leftrightarrow y)$. Type Q is a factorization of type $\mathbb{P}$ by the equivalence relation "if and only if". From author's experience, dealing with quotient propositions in NuPRL is not an easy task and conversion of theorems stated as quotient propositions into standard NuPRL propositions is not trivial either. But the most importantly, using type Q does not solve our problems. Unlike Isabelle Meta Logic, NuPRL propositions do not belong all to the same type. They are split into a chain of propositional types of different levels:

$$\mathbb{P}_1 \subseteq \mathbb{P}_2 \subseteq \ldots \subseteq \mathbb{P}_n \subseteq \ldots$$

such that equality of two propositions of level n is actually an element of type $\mathbb{P}_{n+1}$. It means that type Q in NuPRL is also a sequence of types $Q_n = \mathbb{P}_n//(x, y.x \Leftrightarrow y)$. If we choose some Q_{n_0} to be the interpretation of Isabelle type o, then equality on elements of o would need to be translated as equality of Q_{n_0} elements which in NuPRL belong to a higher-level type Q_{n_0+1}.

Therefore, there probably is no reasonably simple adjustment to NuPRL type $\mathbb{P}_i$ that can be used as an interpretation of Isabelle type o.

All mentioned above problems can be avoided by interpreting Isabelle type o as NuPRL boolean type $\mathbb{B}$. Type $\mathbb{B}$ has only two elements and there is a boolean equality $=_b$ on type $\mathbb{B}$ such that for any elements x and y of type $\mathbb{B}$, $x =_b y$ is also an element of type $\mathbb{B}$.

On the other hand, type $\mathbb{B}$ brings problems of its own. Not every NuPRL type has a boolean equality relation. In fact, only decidable types can have boolean equality in NuPRL because boolean equality, just as any other NuPRL function, would be assumed to be computable. We will extend NuPRL Constructive Type Theory by a boolean equality predicate for any type. Such an extension is non-constructive, or, in other words, classical. Boolean equality is a three-place predicate that takes any type T and two elements x and y of type T as arguments. It returns boolean true if and only if x and y are equal as elements of the type T. To make formulas more readable, NuPRL display mechanism normally hides type argument of boolean equality and shows the boolean equality just as $x =_b y$.

There are two properties of boolean equality that we assume. They normally would be stated as inference rules in NuPRL, but we state them as proof-less theorems in case if somebody would want to use some other primitive abstraction instead of boolean equality. These two theorems are

```
*T bequal_wf      ∀T:U. ∀x,y:T.  (x =b y) ∈ ℬ
*T assert_bequal  ∀T:U. ∀x,y:T.  ↑(x =b y) ⟺ x = y
```

In the last formula, $\uparrow$ stands for NuPRL "assert" operator that converts booleans to propositions. It is important to remember that both boolean equality $=_b$ and propositional equality $=$ have hidden type parameter T.

We also need to add boolean universal quantifier to our theory. It is very similar to standard NuPRL propositional universal quantifier with the exception that it works on boolean terms. Boolean universal quantifier has two arguments: a type term T and a boolean term B with one bound variable x. It will be displayed as $\forall_b x : T.B$. Two assumptions about boolean universal quantifier are also stated as theorems:

```
*T ball_wf       ∀T:U. ∀b:T → 𝔹.  ∀ᵇx:T. b[x] ∈ 𝔹
*T assert_ball   ∀T:U. ∀b:T → 𝔹.  ↑∀ᵇx:T. b[x] ⟺ (∀x:T. ↑b[x])
```

Note that the proposed classical NuPRL extension is not minimal. Boolean universal quantifier can be expressed via propositional universal quantifier and boolean equality as $\forall_b x : T.b[x] \equiv ((\forall x : T. \uparrow b[x]) =_b true)$.

Alternatively, more compact single-argument operator $\downarrow$ that converts propositions to booleans can be used as a primitive notion. Boolean equality can be obviously defined via this operator and propositional equality.

Consistency of a NuPRL classical extension can be shown using D. Howe [5] set-theoretical semantics for NuPRL.

4 Isabelle Type Term Syntax

Isabelle types are elements of Isabelle classes. Each type can belong to a finite number of classes. The list of classes, to which any given type belongs is called a *sort* of this type. Our formalization of Isabelle syntax includes classes and sorts, but they will be ignored later, during interpretation definition. Informally, all classes will be mapped into a NuPRL universal type $\mathbb{U}_i$ of an arbitrary level i.

4.1 Classes and Sorts

Isabelle defines SML type `class` as type `string`, SML type `sort` as type `class list`, and SML type `indexname` as a Cartesian product of `string` and `int`. We formalize these abstractions in NuPRL in almost identical form

```
*A class          Class ==  Atom
*A sort           Sort ==  Class List
*A indexname      Indexname ==  Atom × ℤ
```

where `Atom` is NuPRL type of tokens. All three types defined above are decidable. Boolean equality $=_c$ on classes is inherited from type `Atom`. Boolean equality `=s` on sorts can be defined using boolean list equality, discussed in Section 3.1, and boolean equality on classes. Finally, boolean equality `=ixn` on type `indexname` is defined via boolean equalities on tokens and integers.

4.2 Type Term

Isabelle defines SML type of Isabelle type terms, called `typ`, as

```
datatype typ = Type  of string * typ list
             | TFree of string * sort
             | TVar  of indexname * sort
```

There are two adjustments that we make to this definition in order to simplify reasoning about this type in NuPRL. First, two different kinds of free variables (`TFree` and `TVar`) will be combined into one kind TypVarName Second, among different Isabelle type constructors, functional type constructor `Type(''fun'',[S,T])` plays a special role in the definition of Isabelle type "term". It is convenient to separate this type constructor into a kind of its own.

Therefore, our type term formalization is based on the following, slightly modified, version of SML `typ` datatype

```
datatype typ = Type of string * typ list
             | TFun of typ * typ
             | TVar of TypVarName
```

Obvious homomorphism maps original `typ` type into its modified version.

Using inductive types, the above definition can be written in NuPRL as

```
*A typ_var_name TypVarName ==  Atom × Sort + Indexname × Sort
*A typ          Typ ==  rec(T.Atom × T List + T × T + TypVarName)
*A type         Type(a;ts) ==  inl <a, ts>
*A t_fun        (t ⇒ s) ==  inr (inl <t, s> )
*A t_var        TVar(q) ==  inr inr q
```

Many functions on type `Typ`[2] will be defined using the case split operator. Among such functions are boolean equality `=n` on type `VarName` and boolean equality `=tp` on type `Typ`. These definitions are straightforward and they are omitted here.

5 Term Syntax

Isabelle defines SML type `term` as

```
datatype term = Const of string * typ
              | Free  of string * typ
              | Var   of indexname * typ
              | Bound of int
              | Abs   of string * typ * term
              | op $  of term * term
```

This type includes well-formed terms as well as non-well-formed terms. There are two conditions which an element of type `term` should satisfy in order to be well-formed:

- De Bruijn index i in any subterm $Bound(i)$ should be non-negative and less than the number of abstractions above this subterm in the term tree.
- For each occurrence of application operator $t_1\$t_2$, the type of term t_1 should have the form $T_1 \Rightarrow T_2$ where T_1 is the type of term t_2. Function "type of" is a recursively defined partial function.

[2] We start the world `Typ` with the capital letter when it refers to NuPRL formalization of SML datatype `typ`.

Only well-formed terms are used in Isabelle proofs. Before any term is used in Isabelle, its well-formness is checked via certification process. Well-formed terms are naturally split into groups of terms of the same type.

It is possible to encode this term definition into NuPRL directly. The main disadvantage of this approach is its complexity. All terms will have the same NuPRL type, but different Isabelle types. Hence, Isabelle term types will be defined without using already existing NuPRL type mechanism. More attractive seems to be the idea that a group of Isabelle terms that have the same Isabelle type, should constitute a separate NuPRL type. In this case already existing in NuPRL tactics and theorems can be used to deal with Isabelle types. This approach can be implemented using NuPRL parametrized recursive type constructor.

The key idea is to recursively define a parametrized family of NuPRL types $Term(tp)$, where parameter tp ranges over NuPRL type Typ. Type $Term(tp)$ represents well-formed terms of type tp. It is defined as a disjoint union of several types, corresponding to different Isabelle term constructors.

Constant Terms Constant terms of Isabelle type tp are pairs, whose first elements are arbitrary tokens and the second element is tp:

```
*A const_term   ConstTerm(tp) ==  Atom × {x:Typ| x = tp}
```

Variable Terms Following the definition of type Typ, constructors `Free` and `Var` will be combined into one entity

```
*A var_term     VarTerm(tp) ==  VarName × {x:Typ| x = tp}
```

where

```
*A var_name     VarName ==  Atom + Indexname
```

Note that `VarName` is a decidable type. Boolean equality `=v` on this type can be defined through boolean equality on types `Atom` and `Indexname`.

Bound Variable Term Since we want to define type $Term(tp)$ recursively, any "partially dis-assembled" term should also be considered to be an element of type $Term(tp)$. In particular, $Bound(i)$ needs to be an element of type $Term(tp)$ for some element tp of type Typ. At the same time, Isabelle type of subterm $Bound(i)$ can be determined only from the abstraction operator that binds index i. In a "partially dis-assembled" terms an appropriate abstraction operator may not exist. Hence, we can talk about Isabelle type of subterm $Bound(i)$ only with respect to some kind of environment, that stores types from binding abstractions. Such environment will be called *binding*. Binding is specified by its length n and a function $bd : \mathbb{N}_n \to Typ$ that maps de Bruijn indices into Isabelle types.

Formally, binding is a parameter of type $Term(tp)$, which, therefore, should be written as $Term(tp, n, bd)$. A bound variable term of type tp is an integer k such that $k < n$ and $bd(n - 1 - k) = tp$[3]

```
*A bound_term   BoundTerm(tp;n;bd) ==  {k:ℕn| bd (n - 1 - k) = tp}
```

[3] We assume here that binding stores type information in the "reverse" order. This unusual assumption greatly simplifies the formalization below.

Abstraction Term In SML any Isabelle abstraction term is composed of variable name a (used only for display purposes), type of this variable s, and a term t. Isabelle type of term t is not a part of the abstraction syntax, but it can be re-constructed using SML typ_of function. In NuPRL, it will be convenient to add the type of the term t to the abstraction syntax

```
*A abs_term     AbsTerm(tp;n;bd;tm) ==
                  Atom × s:Typ × q:{q:Typ| tp = (s ⇒ q)}  ×
                        tm <q, n + 1, bd[n → s]>
```

The fourth argument of AbsTerm is a function that maps a triple $\langle tp, n, bd \rangle$ into the type $Term(tp, n, bd)$. Later this argument will be bound by the parametrized recursive type constructor.

Application Term Similarly to the abstraction term case, we add one extra type parameter to application term syntax, namely, the Isabelle type of the argument

```
*A op_term      OpTerm(tp;n;bd;tm) ==
                    s:Typ × tm <(s ⇒ tp), n, bd> × tm <s, n, bd>
```

Finalizing Term Definition Any Isabelle term is either a constant, or a free variable, or a bound term, or an abstraction, or an application. Hence, we would want to define type $Term(tp, n, bd)$ to be the minimal solution of the following type equation:

$$Term(tp, n, bd) = ConstTerm(tp) + VarTerm(tp) + BoundTerm(tp, n, bd) +$$

$$+ AbsTerm(tp, n, bd, \lambda tp, \lambda n, \lambda bd.Term(tp, n, bd)) +$$

$$+ OpTerm(tp, n, bd, \lambda tp, \lambda n, \lambda bd.Term(tp, n, bd))$$

This recursive type has three parameters: tp, n, and bd. Since our parametrized recursive type theory (see Section 3.2) permits only one parameter, these three parameters should be combined into one

```
*A proto_term  ProtoTerm(p) ==
                 parec(tm,p. let <tp,z> = p in let <n,bd> = z
                             in
                             ConstTerm(tp) + VarTerm(tp) +
                                BoundTerm(tp;n;bd) +
                                AbsTerm(tp;n;bd;tm) +
                                OpTerm(tp;n;bd;tm) @ p)
*A term         Term(t;n;bd) ==  ProtoTerm(<t, n, bd>)
*A const        Const(a;t)   ==  inl <a, t>
*A vari         Vari(a;t)    ==  inr (inl <a, t> )
*A bound        Bound(k)     ==  inr inr (inl k )
*A abs          λa:s. tm:q   ==  inr inr inr (inl <a, s, q, tm> )
*A op           (tm1 o tm2)  ==  inr inr inr inr <s, tm1, tm2>
```

Intermediate notion of *proto term* will be extensively used later in proofs by induction on type $Term(tp, n, bd)$. Since induction rule for parametrized recursive types assumes existence of only one parameter in recursive types, it is hard to do inductive proofs directly over type $Term(tp, n, bd)$. Instead, we normally prove an auxiliary theorem carrying induction over type $ProtoTerm(p)$ and derive from it the main result, stated in terms of type $Term(tp, n, bd)$.

Among all types $Term(tp, n, bd)$, special role play those with $n = 0$ since elements of such types correspond to actual Isabelle terms. Display form for type $term(tp, 0, bd)$ is just $Term(tp)$.

5.1 Operations on Terms

Such operations and predicates on Isabelle terms as "substitution" and "occurs free" will be used later to state Isabelle Meta Logic inference rules. Definitions of these operators are based on *term case split* constructor.

Binding If tm is a term with a free variable vn of type tp, then, before this variable can be bound by an abstraction, it should be converted into a bound variable. If $tm \in Term(tp', n, bd)$, then such conversion can be done by the function

```
*M bind_ml   bind(tm;vn;tp;n;bd) ==r case tm
 of Const(a,t) -> tm
  | Var(v,t) -> if (v =v vn) ∧b (t =tp tp) then Bound(n) else tm fi
  | Bound(j) -> tm
  | λa:s. m:q -> λa:s. bind(m;vn;tp;n + 1;bd[n → s]):q
  | (f o m | s) -> (bind(f;vn;tp;n;bd) o bind(m;vn;tp;n;bd))
```

This function substitutes an appropriate de Bruijn index $Bound(i)$ for every occurrence of variable $Var(vn, tp)$ in term tm. The resulting term has one extra element in the binding

```
*T bind_wf_tm  ∀t:Typ. ∀n:ℕ. ∀bd:ℕn → Typ. ∀tm:Term(t;n;bd).
               ∀vn:VarName. ∀tp:Typ. ∀k:ℕ. k = n + 1 ⇒
               bind(tm;vn;tp;n;bd) ∈ Term(t;k;[tp >> bd])
```

where `[tp>>bd]` is the operator *shift* on records that was discussed in Section 3.1.

Substitution The operator

```
*M subs_ml   tm[vn,tp→t]  ==r case tm
   of Const(a,s) -> tm
    | Var(w,s) -> if (w =v vn) ∧b (s =tp tp) then t else tm fi
    | Bound(j) -> tm
    | λa:s. m:q -> λa:s. m[vn,tp→t]:q
    | (f o m | s) -> (f[vn,tp→t] o m[vn,tp→t])
```

substitutes term t for every occurrence of variable $Var(vn, tp)$ in the term tm. In order to avoid de Bruijn indices collision, term t in the above definition should have nil binding

```
*T subs_wf    ∀t:Typ. ∀n:ℕ. ∀bd:ℕn → Typ. ∀tm:Term(t;n;bd).
              ∀vn:VarName. ∀tp:Typ. ∀b:ℕ0 → Typ. ∀m:Term(tp).
                 tm[vn,tp→m] ∈ Term(t;n;bd)
```

Free Occurrence Some logical rules require a variable not to occur free in a term. We prefer to define boolean predicate "variable $Var(vn, tp)$ *does* occur in term tm":

```
*M free_ml   Free(vn;tp;tm) ==r case tm
    of Const(a,tp') -> ff
     | Var(vn',tp') -> (vn' =v vn) ∧b (tp' =tp tp)
     | Bound(j) -> ff
     | λa:s. tm':q -> Free(vn;tp;tm')
     | (f o m | s) -> Free(vn;tp;f) ∨b Free(vn;tp;m)
```

6 Interpretation

6.1 Type Interpretation

In order to define Isabelle type term interpretation in NuPRL, one needs to select evaluation of type variables and type constructors. After that an interpretation can be extrapolated on all type terms. The basic evaluation of constructors and variables will be called *type evaluation*

```
*A t_eval   TEval ==  (Atom → 𝕌 List → 𝕌) × (TypVarName → 𝕌)
```

For any given type evaluation ev, an interpretation of a type term t is defined by recursion:

```
*M t_interp_ml ρ(t) ==r case t
  of Type(a,ts) -> ev.1 a map(λx.ρ(x);ts)
   | (p ⇒ q) -> ρ(p) → ρ(q)
   | Var(n) -> ev.2 n
```

Type interpretation $\rho(t)$ has two arguments: type term t and type evaluation ev. The last one is not normally displayed.

6.2 Term Evaluation

Interpretation of an Isabelle term will be defined with respect to a type evaluation tev and term evaluation ev, where term evaluation assigns values to atomic terms – constants and variables. Term evaluation is called just *evaluation*

```
*A eval   Eval(tev) ==  (Atom → t:Typ → ρ(t)) ×
                        (VarName → t:Typ → ρ(t))
```

Application of an evaluation ev to constants and variables is called *constant evaluation* and *variable evaluation* correspondingly:

```
*A const_eval   ConstEval(a;t|e) ==  e.1 a t
*A var_eval     VarEval(i;t|e)   ==  e.2 i t
```

Later, verifying Isabelle Meta Logic rules, we will be using operator

```
*A eval_update ev[vn,tp → val] ==
<ev.1, λw,s.if (w =v vn) ∧b (s =tp tp) then val else ev.2 w s fi >
```

that changes evaluation function ev at the point $Var(vn, tp)$.

6.3 Binding Evaluation

If an Isabelle term *tm* belongs to a NuPRL type $Term(tp, n, bd)$ and $n > 0$, then *tm* can have occurrences of de Bruijn indices that are not bound by any abstraction. Interpretation of such dis-assembled terms can be defined only if we assign first some specific values to unbound indices. We call this assignment a *binding value*. Binding value of elements of type $Term(tp, n, bd)$ has to map each integer index i, such that $i < n$, into an element of NuPRL type $\rho(bd(i))$:

```
*A bind_value    BindVal(n;bd) ==  i:ℕn → ρ(bd i)
```

Binding values are essentially records that we have discussed in Section 3.1. Introduced there operators *update*, *shift*, and *tail* can be applied to binding values as well. Although it is more convenient to define duplicates of these operators to deal specifically with binding values. For example,

```
*A bdv_update    (bdv:bd)[n => v:t] ==  bdv[n → v]
```

Note that extra parameters *bd* and *t* do not appear on the right hand side of the definition. These are "dummy" parameters incorporated into `bdv_update` to assist NuPRL type guessing procedure. They can be ignored from the logical point of view. Normally we would make such parameters hidden, but in this particular case they are sometimes helpful for proof understanding.

Operators `bdv_shift` and `bdv_tail` also have dummy parameters for type guessing, but in our formalization they are hidden:

```
*A bdv_shift      [v >>> bdv] ==  [v >> bdv]
*A bdv_tail       (bdv|k) ==  (bdv|k)
```

We also define `bdv_apply` operator as a duplicate of the standard NuPRL application. This definition is also needed only in order to assist type guessing procedure.

```
*A bdv_apply      [bdv](i) ==  bdv i
```

6.4 Finalizing Interpretation Definition

For any binding value *bdv*, and an evaluation *ev*, we define an interpretation of a term *t* recursively as

```
*M interp_ml   β(t|bdv,ev)
   ==r case t
        of Const(a,tp) -> ConstEval(a;tp|ev)
         | Var(i,tp) -> VarEval(i;tp|ev)
         | Bound(j) -> [bdv](n - 1 - j)
         | λa:s. tm:q -> λz.β(tm|(bdv:bd)[n => z:s],ev)
         | (f o tm | s) -> β(f|bdv,ev) β(tm|bdv,ev)
```

If term *t* has Isabelle type *tp*, then its interpretation belongs to NuPRL type $\rho(tp)$

```
*T interp_wf ∀tev:TEval. ∀ev:Eval(tev). ∀tp:Typ. ∀n:ℕ.
              ∀bd:ℕn → Typ. ∀t:Term(tp;n;bd). ∀bdv:BindVal(n;bd).
                 β(t|bdv,ev) ∈ ρ(tp)
```

7 Isabelle Meta Logic

In the previous sections we have formalized Isabelle syntax and defined its interpretation in NuPRL. In this section we state Isabelle meta rules and show that they are translated into valid NuPRL statements.

7.1 Meta Logic Syntax

Propositions Isabelle Meta Logic declares a type constant o, which stands for Isabelle type of all propositions. In NuPRL we represent this declaration by the following definition:

```
*A prop_typ   0 ==  Type("prop";[])
```

We restrict the class of possible type evaluations to such evaluations that map constant o into NuPRL boolean type $\mathbb{B}$. Technically, this restriction is put by adding condition

$$(pr_1(tev))("prop",[]) = \mathbb{B}$$

as a hypothesis to all theorems that we are proving about type o. We call the above condition *propositional signature*

```
*A prop_sign  PropSign ==  tev.1 "prop" [] = 𝔹
```

Propositional signature has type evaluation *tev* and universe level i as hidden parameters.

Implication Isabelle declares $\Longrightarrow$ as a constant of type $o \Rightarrow o \Rightarrow o$. Accordingly, in NuPRL we define Isabelle implication as

```
*A imp_const  ==> ==  Const("==>";(0 ⇒ (0 ⇒ 0)))
```

We will assume that Isabelle implication is interpreted as NuPRL boolean implication:

```
*A imp_sign   ImpSign ==
     ConstEval("==>";(0 ⇒ (0 ⇒ 0))|ev) = (λx,y.x ⇒b y)
```

Each time when constant $\Longrightarrow$ is used in Isabelle meta rules, it is applied to a pair of arguments. As a result, the following notation is handy:

```
*A imp          (p ==> q) ==  ((==> o p) o q)
```

Universal Quantifier Isabelle declares universal quantifier $\wedge$ as a constant of the type $(\alpha \Rightarrow o) \Rightarrow o$. Although at the first glance it seems that the same constant $\wedge$ belongs to the type $(\alpha \Rightarrow o) \Rightarrow o$ for any type α of class *term*, this is not true. Any instance of constant $\wedge$ in any Isabelle term has form $Const("all", (\alpha \Rightarrow o) \Rightarrow o)$ for a particular type term α. Hence, $\wedge$ is actually a family of constants, parametrized by α.

```
*A all_const   ∩('a) ==  Const("all";(('a ⇒ 0) ⇒ 0))
*A all_sign    AllSign ==
                ∀'a:Typ. ConstEval("all";(('a ⇒ 0) ⇒ 0)|ev) =
                              (λb.∀bx:ρ('a). b x)
*A iall         ∩(a:s. p) ==  (∩(s) o λa:s. p:0)
```

We added letter i in the name "iall" in order to avoid name collision with standard NuPRL universal quantifier.

Equality Isabelle declares equality predicate as a constant of the type $\alpha \Rightarrow \alpha \Rightarrow o$. Just like the universal quantifier, this constant actually is a family of constants, parametrized by α:

```
*A eq_const    eq_const('a) ==  Const("==";('a ⇒ ('a ⇒ 0)))
*A eq_sign     EqSign ==
   ∀'a:Typ. ConstEval("==";('a ⇒ ('a ⇒ 0))|ev) = (λx,y.(x =b y))
*A eq            (x ≡ y) ==  ((eq_const(tp) o x) o y)
```

Operator *eq* hides parameter *tp* to make formulas more readable.

7.2 Meta Logic Rules

Isabelle Meta Logic rules stated in [13] are reproduced on Figure 1. The following

---

1. $\dfrac{\overset{[\phi]}{\psi}}{\phi \Rightarrow \psi}$ 2. $\dfrac{\phi \Rightarrow \psi \quad \phi}{\psi}$ 3. $\dfrac{\phi}{\wedge_\sigma x.\phi}$

4. $\dfrac{\wedge_\sigma x.\phi}{\phi[b/x]}$ 5. $\dfrac{}{a \equiv_\sigma a}$ 6. $\dfrac{a \equiv_\sigma b}{b \equiv_\sigma a}$

7. $\dfrac{a \equiv_\sigma b \quad b \equiv_\sigma c}{a \equiv_\sigma c}$ 8. $\dfrac{}{(\lambda x.a) \equiv_\sigma (\lambda y.a[y/x])}$ 9. $\dfrac{}{((\lambda x.a)b) \equiv_\sigma a[b/x]}$

10. $\dfrac{f(x) \equiv_\sigma g(x)}{f \equiv_{\rho \to \sigma} g}$ 11. $\dfrac{a \equiv_\sigma b}{(\lambda x.a) \equiv_{\rho \to \sigma} (\lambda x.b)}$ 12. $\dfrac{f \equiv_{\rho \to \sigma} g \quad a \equiv_\rho b}{f(a) \equiv_\sigma g(b)}$

13. $\dfrac{\overset{[\phi]}{\psi} \quad \overset{[\psi]}{\phi}}{\phi \equiv_{prop} \psi}$ 14. $\dfrac{\phi \equiv_{prop} \psi \quad \phi}{\psi}$

Fig. 1. Isabelle Meta Logic rules

theorem verify these rules in the classical extension of NuPRL. Complete formal proofs are available from the author's Web page.

```
*T rule_1
   ∀tev:TEval. ∀n:ℕ. ∀bd:ℕn → Typ. ∀bdv:BindVal(n;bd).
    ∀p,q:Term(0;n;bd). ∀ev:Eval(tev). PropSign ⇒ ImpSign ⇒
     (↑β(p|bdv,ev) ⇒ ↑β(q|bdv,ev)) ⇒
      ↑β((p ==> q)|bdv,ev)
*T rule_2
   ∀tev:TEval. ∀n:ℕ. ∀bd:ℕn → Typ. ∀bdv:BindVal(n;bd).
    ∀p,q:Term(0;n;bd). ∀ev:Eval(tev). PropSign ⇒ ImpSign ⇒
     ↑β((p ==> q)|bdv,ev) ⇒ ↑β(p|bdv,ev) ⇒ ↑β(q|bdv,ev)
*T rule_3
   ∀tev:TEval. ∀bd:ℕ0 → Typ. ∀bdv:BindVal(0;bd). ∀p:Term(0).
    ∀a:Atom. ∀s:Typ. ∀vn:VarName. PropSign ⇒
     (∀ev:Eval(tev). AllSign ⇒ ↑β(p|bdv,ev)) ⇒
      (∀ev:Eval(tev). AllSign ⇒ ↑β(∩(a:s. bind(p;vn;s))|bdv,ev))
```

```
*T rule_4
   ∀tev:TEval. ∀ev:Eval(tev). ∀bd:ℕ0 → Typ. ∀bdv:BindVal(0;bd).
    ∀p:Term(0). ∀a:Atom. ∀s:Typ. ∀vn:VarName. ∀m:Term(s).
     PropSign ⇒ AllSign ⇒ ↑β(∩(a:s. bind(p;vn;s))|bdv,ev)
      ⇒ ↑β(p[vn,s→m]|bdv,ev)
*T rule_5
   ∀tev:TEval. ∀ev:Eval(tev). ∀n:ℕ. ∀bd:ℕn → Typ.
    ∀bdv:BindVal(n;bd). ∀tp:Typ. ∀a:Term(tp;n;bd). PropSign ⇒
     EqSign ⇒ ↑β((a ≡ a)|bdv,ev)
*T rule_6
   ∀tev:TEval. ∀ev:Eval(tev). ∀n:ℕ. ∀bd:ℕn → Typ.
    ∀bdv:BindVal(n;bd). ∀tp:Typ. ∀a,b:Term(tp;n;bd). PropSign ⇒
     EqSign ⇒ ↑β((a ≡ b)|bdv,ev) ⇒ ↑β((b ≡ a)|bdv,ev)
*T rule_7
   ∀tev:TEval. ∀ev:Eval(tev). ∀n:ℕ. ∀bd:ℕn → Typ.
    ∀bdv:BindVal(n;bd). ∀tp:Typ. ∀a,b,c:Term(tp;n;bd). PropSign ⇒
     EqSign ⇒ ↑β((a ≡ b)|bdv,ev) ⇒
      ↑β((b ≡ c)|bdv,ev) ⇒ ↑β((a ≡ c)|bdv,ev)
*T rule_8
   ∀tev:TEval. ∀ev:Eval(tev). ∀n:ℕ. ∀bd:ℕn → Typ.
    ∀a,b:Atom. ∀s,q:Typ. ∀tm:Term(q;n + 1;bd[n → s]).
     ∀bdv:BindVal(n;bd). PropSign ⇒ EqSign ⇒
      ↑β((λa:s. tm:q ≡ λb:s. tm:q)|bdv,ev)
*T rule_9
   ∀tev:TEval. ∀ev:Eval(tev). ∀bd:ℕ0 → Typ. ∀bdv:BindVal(0;bd).
    ∀a:Atom. ∀s,q:Typ. ∀vn:VarName. ∀t:Term(s). ∀m:Term(q).
     PropSign ⇒ EqSign ⇒
      ↑β(((λa:q. bind(t;vn;q):s o m) ≡ t[vn,q→m])|bdv,ev)
*T rule_10
   ∀tev:TEval. ∀n:ℕ. ∀bd:ℕn → Typ. ∀bdv:BindVal(n;bd).
    ∀s,q:Typ. ∀f,g:Term((s ⇒ q);n;bd). ∀x:VarName. PropSign ⇒
     ↑¬ᵦFree(x;s;f) ⇒ ↑¬ᵦFree(x;s;g) ⇒
      (∀ev:Eval(tev). EqSign ⇒
          ↑β(((f o Vari(x;s)) ≡ (g o Vari(x;s)))|bdv,ev))
       ⇒ (∀ev:Eval(tev). EqSign ⇒ ↑β((f ≡ g)|bdv,ev))
*T rule_11
   ∀tev:TEval. ∀bd:ℕ0 → Typ. ∀bdv:BindVal(0;bd). ∀tp:Typ.
    ∀a:Atom. ∀s:Typ. ∀vn:VarName. ∀p,q:Term(tp). PropSign ⇒
     (∀ev:Eval(tev). EqSign ⇒ ↑β((p ≡ q)|bdv,ev)) ⇒
      (∀ev:Eval(tev) EqSign ⇒
      ↑β((λa:s. bind(p;vn;s):tp ≡ λa:s. bind(q;vn;s):tp)|bdv,ev))
*T rule_12
   ∀tev:TEval. ∀ev:Eval(tev). ∀tp:Typ. ∀n:ℕ.
    ∀bd:ℕn → Typ. ∀bdv:BindVal(n;bd). ∀s:Typ.
     ∀f,g:Term((s ⇒ tp);n;bd). ∀a,b:Term(s;n;bd). PropSign ⇒
      EqSign ⇒ ↑β((f ≡ g)|bdv,ev) ⇒ ↑β((a ≡ b)|bdv,ev) ⇒
       ↑β(((f o a) ≡ (g o b))|bdv,ev)
*T rule_13
   ∀tev:TEval. ∀n:ℕ. ∀bd:ℕn → Typ. ∀bdv:BindVal(n;bd).
    ∀p,q:Term(0;n;bd). ∀ev:Eval(tev). PropSign ⇒ EqSign ⇒
```

```
     (↑β(p|bdv,ev) ⇒ ↑β(q|bdv,ev)) ⇒
      (↑β(q|bdv,ev) ⇒ ↑β(p|bdv,ev)) ⇒ ↑β((p ≡ q)|bdv,ev)
*T rule_14
   ∀tev:TEval. ∀n:ℕ. ∀bd:ℕn → Typ. ∀bdv:BindVal(n;bd).
    ∀p,q:Term(0;n;bd). ∀ev:Eval(tev). PropSign ⇒ EqSign ⇒
     ↑β((p ≡ q)|bdv,ev) ⇒ ↑β(p|bdv,ev) ⇒ ↑β(q|bdv,ev)
```

8 Acknowledgements

Author would like to thank Robert L. Constable for numerous discussions and support of this project and Doug J. Howe for the time and effort of explaining technical details of his work.

References

1. N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to Church-Rosser theorem. *Indag. Math.*, 34:381–392, 1972.
2. R.L. Constable et al. *Implementing Mathematics with Nuprl Proof Development System.* Prentice Hall, 1986.
3. M.J.C. Gordon and T.F. Melham. *Introduction to HOL – A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
4. D.J. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 267–282, Berlin, 1996. Springer-Verlag.
5. D.J Howe. Semantics foundation for embedding HOL in Nuprl. In M. Wirsing and A. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 85–101, Berlin, 1996. Springer-Verlag.
6. N.P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):159–173, 1991.
7. P.F. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, 1988.
8. J. Meseguer and M.-O. Stehr. The HOL-NuPRL Connection from the Viewpoint of General Logic. Working paper, June 1999.
9. P. Naumov. *Formalizing Reference Types in NuPRL.* PhD thesis, Cornell University, August 1998.
10. P. Naumov. Formalization of Isabelle Meta Logic in NuPRL. Technical Report TR99-1769, Cornell University, Computer Science Department, Ithaca, NY, September 1999.
11. P. Naumov. Importing Isabelle Formal Mathematics into NuPRL. In *Supplemental proceedings of The 12th International Conference on Theorem Proving in Higher Order Logics*, Nice, France, September 1999.
12. P. Naumov. Importing Isabelle Formal Mathematics into NuPRL. Technical Report TR99-1734, Cornell University, Computer Science Department, Ithaca, NY, February 1999.
13. L.C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
14. L.C. Paulson. *Isabelle – A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1994.

Axiomatic Semantics for Java^{light} in Isabelle/HOL

David von Oheimb*

Technische Universität München
http://www.in.tum.de/~oheimb/

Abstract. We introduce a Hoare-style calculus for a nearly full subset of sequential Java, which we call Java^{light}. In particular, we present solutions to challenging features like exception handling, static initialization of classes and dynamic binding of methods.
This axiomatic semantics has been proved sound and complete w.r.t. pour operational semantics of Java^{light}, described in earlier papers. To our knowledge, our Hoare logic is the first one for an object-oriented language that has been proved complete. The proofs also give new insights into the role of type-safety. All the formalization and proofs have been done with the theorem prover Isabelle/HOL.

1 Introduction

Since languages like Java are widely used in safety-critical applications, verification of object-oriented programs has grown more and more important. A first step towards verification seems to be developing a suitable axiomatic semantics (a.k.a. "Hoare logic") for such languages.

Recently several proposals for Hoare logics for object-oriented languages, e.g. [dB99,PHM99,HJ00], have been given. Typically they deal with some small core language and are partially proved sound on paper (except for [HJ00], which has been machine-checked). None of them has been proved complete. Our new logic, in part inspired by [PHM99], has the following special merits.

- Apart from static overloading and dynamic binding of methods as well as references to dynamically allocated objects, it also covers full exception handling, static fields and methods, and static initialization of classes. Thus our sequential sublanguage Java^{light} is almost the same as Java Card[Sun99].
- Instead of modeling expressions with side-effects as assignments to intermediate variables, it handles them first-class. Thus programs to be verified do not need to undergo an artificial structural transformation.
- It is both sound – w.r.t. a mature formalization of the operational semantics of Java – and complete. This means that programs using even non-trivial features like mutual recursion and dynamic binding can be proved correct.
- It has been both defined and verified within the interactive theorem proving system Isabelle/HOL [Pau94]. This guarantees a rigorous and unambiguous formalization and reliable proofs.

2 Some basics of the Java^{light} formalization

Our axiomatic semantics inherits all features concerning type declarations and the program state from our operational semantics of Java^{light}. See [ON99] for a more detailed description.

Here we just recall that a program Γ (which serves as the context for most judgments) consists of a list of class and interface declarations and that the execution state is defined as

datatype st = st ($globs$) ($locals$)
types $state$ = $xcpt\ option \times st$

* Research funded by the DFG Project BALI, http://isabelle.in.tum.de/Bali/

where *globs* and *locals* map class references to objects (including class objects) and variable names to values, respectively, and *xcpt* references an exception object. Using the projection operators on tuples, we define e.g. normal $\sigma \equiv$ fst σ = None, which expresses that in state σ there is no pending exception, and write snd σ to refer to the state without the information on exceptions, typically denoted by s.

A term of Java^{light} is either an expression, a statement, a variable, or an expression list, and has a corresponding result. For uniformity, even a statement has a (dummy) result, called Unit. The result of a variable is an *lval*, which is a value (for read access) and a state update function (for write access).

types *terms* = (*expr* + *stmt*) + *var* + *expr list*
types *vals* = *val* + *lval* + *val list*
types *lval* = *val* × (*val* → *state* → *state*)

There are many other auxiliary type and function definitions which we cannot define here for lack of space. The complete Isabelle sources, including an example, may be obtained from `http://isabelle.in.tum.de/Bali/src/Bali4/`.

3 The axiomatic semantics

3.1 Assertions

In our axiomatic semantics we shallow-embed assertions in the meta logic HOL, i.e. define them as predicates on (basically) the state, making the dependence on the state explicit and simplifying their handling within Isabelle. This general approach is extended in two ways.

- We let the assertions depend also on so-called *auxiliary variables* (denoted by the meta variable Z of any type α), which are required to relate variable contents between pre- and postconditions, as discussed in [Sch97].
- We extend the state by a stack (implemented as a list and denoted by Y) of result values of type res, which are used to transfer results between Hoare triples. In an operational semantics, these nameless values can be referred to via meta variables, but in an axiomatic semantics, such a simple technique is impossible since all values in a triple are logically bound to that scope (by universal quantification).

As a result, we define the type of assertions (with parameter α) as

types α *assn* = *res list* × *state* → α → *bool*
datatype *res* = Res (*vals*) | Xcpt (*xcpt option*) | Lcls (*locals*) | DynT (*tname*)

We write e.g. Val v as an abbreviation for Res (In1 v), injecting a value v into *res*. Names like Val and DynT are used not only as constructors, but also as (destructor) patterns. For example, λVal v:Y. $f\ v\ Y$ is a function on the result stack that expects a value v as the top element and passes it to f together with the rest of the stack, referred to by Y.

In order to keep the Hoare rules short and thus more readable, we define several assertion (predicate) transformers.

- $\lambda s.\ P\ s \equiv \lambda(Y,\sigma).\ P$ (snd σ) (Y,σ) allows P to peek at the state directly.
- $P\ \wedge.\ p \equiv \lambda(Y,\sigma)\ Z.\ P\ (Y,\sigma)\ Z \wedge p\ \sigma$ means that not only P holds but also p (applied to the program state only). The assertion Normal $P \equiv P\ \wedge.$ normal is a simple application stating that P holds and no exception has occurred.
- $P{\leftarrow}{:}f \equiv \lambda(Y,\sigma).\ P\ (Y,f\ \sigma)$ means that P holds for the state transformed by f.
- $P\ ;.\ f \equiv \lambda(Y,\sigma')\ Z.\ \exists\sigma.\ P\ (Y,\sigma)\ Z \wedge \sigma' = f\ \sigma$ means that P holds for some state σ and the current state is then derived from σ by the state transformer f.

3.2 Hoare triples and validity

We define triples as judgments of the form $prog \vdash \{\alpha\ assn\}\ terms \succ \{\alpha\ assn\}$ with some obvious variants for the different sorts of terms, e.g.
$\Gamma \vdash \{P\}\ e \text{-}\succ \{Q\} \equiv \Gamma \vdash \{P\}\ \mathsf{In1(Inl}\ e\mathsf{)} \succ \{Q\}$ and $\{P\}\ .c.\ \{Q\} \equiv \{P\}\ \mathsf{In1(Inr}\ c\mathsf{)} \succ \{Q\}$.

Here we simplify the presentation by leaving out triples as assumptions within judgments, which are necessary to handle recursion; we have discussed this issue in detail in [Ohe99]. The validity of triples is defined as

$$\Gamma \models \{P\}\ t \succ \{Q\} \equiv \forall Y\ \sigma\ Z.\ P\ (Y,\sigma)\ Z \longrightarrow \mathsf{type\_ok}\ \Gamma\ t\ \sigma \longrightarrow \forall v\ \sigma'.\ \Gamma \vdash \sigma\ -t\succ\!\!\rightarrow\ (v,\sigma') \longrightarrow Q\ (\mathsf{res}\ t\ v\ Y,\sigma')\ Z$$

where Y stands for the result stack and Z denotes the auxiliary variables. The judgment $\mathsf{type\_ok}\ \Gamma\ t\ \sigma$ means that the term t is well-typed (if σ is a normal state) and that all values in σ conform to their static types. This additional precondition is required to ensure soundness, as discussed in §3.6. $\Gamma \vdash \sigma\ -t\succ\!\!\rightarrow\ (v,\sigma')$ is the evaluation judgment from the operational semantics meaning that from the initial state σ the term t evaluates to a value v and final state σ'. Note that we define partial correctness.

Unless t is statement, the result value v is pushed onto the result stack via
$\mathsf{res}\ t\ v\ Y \equiv \mathsf{if\ is\_stmt}\ t\ \mathsf{then}\ Y\ \mathsf{else\ Res}\ v{:}Y$.

3.3 Result value passing

We define the following abbreviations for producing and consuming results:

- $P\uparrow\mathsf{:w} \equiv \lambda(\mathsf{Y},\sigma).\ \mathsf{P\ (w:Y},\sigma)$ means that P holds where the result w is pushed.
- $\lambda w{:}.\ P\ w \equiv \lambda(w{:}Y,\sigma).\ P\ w\ (Y,\sigma)$ expects and pops a result w and asserts $P\ w$.

A typical application of the former is the rule for literal values v:

$$Lit\ \frac{}{\Gamma \vdash \{\mathsf{Normal}\ (P\uparrow\mathsf{:Val\ v})\}\ \mathsf{Lit\ v}\text{-}\succ \{\mathsf{P}\}}$$

Analogously to the well-known assignment rule, it states that for a literal expression (i.e., constant) v the postcondition P can be derived if P – with the value v inserted – holds as the precondition and the (pre-)state is normal.

The rule for array variables handles result values in a more advanced way:

$$AVar\ \frac{\Gamma \vdash \{\mathsf{Normal}\ P\}\ e_1\text{-}\succ \{Q\} \quad \Gamma \vdash \{Q\}\ e_2\text{-}\succ \{\lambda\mathsf{Val}\ i{:}.\ \mathsf{RefVar\ (avar}\ \Gamma\ i)\ R\}}{\Gamma \vdash \{\mathsf{Normal}\ P\}\ e_1[e_2]{=}\succ \{R\}}$$

where $\mathsf{RefVar}\ vf\ P \equiv \lambda(\mathsf{Val}\ a{:}Y,(x,s)).\ \mathsf{let}\ (v,x') = vf\ a\ x\ s\ \mathsf{in}\ (P\uparrow\mathsf{:Var\ v})\ (\mathsf{Y},(\mathsf{x'},\mathsf{s}))$.
Both subexpressions are evaluated in sequence, where Q as intermediate assertion typically involves the result of e_1. The final postcondition R is modified for the proof on e_2 as follows: from the result stack two values are expected and popped, namely i (the index) and a (an address) of e_2 and e_1, respectively. Out of these and the intermediate state (x,s), the auxiliary function avar computes the variable v, which is pushed as the final result, and (possibly) an exception x'.

For terms involving a condition, we define the assertion $P\uparrow\mathsf{:Bool{=}b} \equiv \lambda(\mathsf{Y},\sigma)\ \mathsf{Z}.\ \exists v.\ (P\uparrow\mathsf{:Val\ v})\ (\mathsf{Y},\sigma)\ \mathsf{Z} \wedge (\mathsf{normal}\ \sigma \longrightarrow \mathsf{the\_Bool}\ v = b)$ expressing (basically) that the result of a preceding boolean expression is b. Together with the meta-level conditional expression (if b then e_1 else e_2) depending on b and $P'\uparrow\mathsf{:Bool{=}b}$ identifying b with the result of a boolean expression e_0, we can describe both branches of conditional terms with a single triple, like in

$$Cond\ \frac{\Gamma \vdash \{\mathsf{Normal}\ P\}\ e_0\text{-}\succ \{P'\} \quad \forall b.\ \Gamma \vdash \{P'\uparrow\mathsf{:Bool{=}b}\}\ (\mathsf{if\ b\ then\ e_1\ else\ e_2})\text{-}\succ \{\mathsf{Q}\}}{\Gamma \vdash \{\mathsf{Normal}\ P\}\ e_0\ ?\ e_1\ :\ e_2\text{-}\succ \{Q\}}$$

The value b is universally quantified, such that when applying this rule, one has to prove its second antecedent for any possible value, i.e., both True and False. What is a notational convenience here (to avoid two triples, one for each case), will be essential for the *Call* rule, given below.

The rules for the standard statements appear almost as usual:

$$\textit{Skip}\ \frac{}{\Gamma\vdash\{P\}\ .\mathsf{Skip}.\ \{P\}} \qquad \textit{Loop}\ \frac{\Gamma\vdash\{P\}\ e\text{-}\!\succ\ \{P'\} \quad \Gamma\vdash\{P'\uparrow\text{:Bool=True}\}\ .\mathsf{c}.\ \{\mathsf{P}\}}{\Gamma\vdash\{P\}\ .\mathtt{while}(e)\ c.\ \{P'\uparrow\text{:Bool=False}\}}$$

Note that in all[1] rules (except Loop for obvious reasons) the postconditions of the conclusion is a variable. Thus in the typical "backward-proof" style of Hoare logic the rules are applied easily.

3.4 Dynamic binding

The great challenge of an axiomatic semantics for an object-oriented language is dynamic binding in method calls, for two reasons.

First, the code selected depends on the class D dynamically computed from the target reference expression e. The range of values for D depends on the whole program and thus cannot be fixed locally, in contrast to the two possible boolean values appearing in conditional terms described above. Standard Hoare triples cannot express such an unbound case distinction. We handle this problem with the strong technique given above, using universal quantification and the precondition $R\uparrow$:DynT D $\wedge\ldots$ with the special result value DynT D. An alternative solution is given in [PHM99], where D is referred to via This and the possible variety of D is handled in a cascadic way using two special rules.

Second, the actual value D often can be inferred statically, but in general for invocation mode "virtual", one can only know that it is a subtype of some reference type rt computed by static analysis during type-checking. The intuitive – but absolutely non-trivial – reason why the subtype relation Class $D\preceq$RefT rt holds is of course type-safety. The problem here is how to establish this relation. The rules given in [PHM99], for example, put the burden of verifying the relation on the user, which is possible, but in general not practically feasible. In contrast, our solution make the relation available to the user as a helpful assumption (see the sub-formula $\Gamma\vdash mode\to D\preceq rt$ in the rule given below), which transfers the proof burden once and for all to the soundness proof on the meta-level.

The remaining parts of the rule for method calls deals with the unproblematic issues of argument evaluation, setting up the local variables (including parameters) of the called method and restoring the previous local variables on return, for which we use the special result value Lcls.

$$\textit{Call}\ \frac{\begin{array}{r}\Gamma\vdash\{\mathsf{Normal}\ P\}\ e\text{-}\!\succ\ \{Q\}\\ \Gamma\vdash\{Q\}\ args\doteq\!\succ\ \{\lambda\mathsf{Vals}\ vs\text{:Val}\ a\text{:.}\ \lambda s:\ \mathsf{let}\ D=\mathsf{target}\ mode\ s\ a\ \tau\ \mathsf{in}\\ R\uparrow\text{:DynT D}\uparrow\text{:Lcls (locals s)}\leftarrow\text{:init\_lvars}\ \Gamma\ \mathsf{D\ (mn,pTs)\ mode\ a\ vs}\}\\ \forall D.\ \Gamma\vdash\{R\uparrow\text{:DynT D}\wedge\lambda\sigma.\ \mathsf{normal}\ \sigma\longrightarrow\Gamma\vdash\mathsf{mode}\to\mathsf{D}\preceq\mathsf{rt}\}\\ \mathsf{Body}\ D\ (mn,pTs)\text{-}\!\succ\ \{\lambda\mathsf{Val}\ v\text{:Lcls}\ l\text{:.}\ S\uparrow\text{:Val v}\leftarrow\text{:set\_lvars l}\}\end{array}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ \{rt,\tau,mode\}e.mn(\{pTs\}args)\text{-}\!\succ\ \{S\}}$$

3.5 Class initialization

The static initialization of classes is an unpleasant feature to model as its structure depends on the class hierarchy and it is not syntax-driven but rather triggered on demand. Thus at several places, e.g. field access and method calls, one has to consider potential initialization of some referenced class C, which we denote by the special statement init C. If the class in question is already initialized, there is nothing do:

$$\textit{Done}\ \frac{}{\Gamma\vdash\{\mathsf{Normal}\ (P\wedge.\ \mathsf{initd}\ C)\}\ .\mathsf{init}\ C.\ \{P\}}$$

[1] The rules not mentioned here may be found in the appendix.

Otherwise, initialization allocates a new class object, treats the superclass (if any), and finally invokes the static initializers of the class itself, whereby the current local variables have to be hidden and later restored:

$$\textit{Init}\ \frac{\begin{array}{c}\text{the (class } \Gamma\ C) = (sc,\_,\_,\ ,ini)\\ sup = \text{if } C = \texttt{Object} \text{ then Skip else init } sc\\ \Gamma\vdash\{\text{Normal } ((P\ \wedge.\ \text{Not} \circ \text{initd } C)\ ;.\ \text{supd (new\_class\_obj } \Gamma\ C))\}\ .sup.\ \{Q\uparrow;.\lambda s.\ \text{Lcls (locals s)}\}\\ \Gamma\vdash\{Q\ ;.\ \text{set\_lvars empty}\}\ .ini.\ \{\lambda \text{Lcls } l;.\ R\leftarrow;\text{set\_lvars } l\}\end{array}}{\Gamma\vdash\{\text{Normal } (P\ \wedge.\ \text{Not} \circ \text{initd } C)\}\ .\text{init } C.\ \{R\}}$$

3.6 Soundness and completeness

With the help of Isabelle/HOL, we have proved soundness and completeness:

$$\text{wf\_prog } \Gamma\ \longrightarrow\ \Gamma\models\{P\}\ t\succ\ \{Q\}\ =\ \Gamma\vdash\{P\}\ t\succ\ \{Q\}$$

where wf_prog Γ means that the program Γ is well-formed. As usual, soundness is proved by rule induction on the derivation of triples. Surprisingly, type-safety plays a crucial role here. The important fact that for method calls the subtype relation Class $D\preceq$RefT rt holds can be derived only if the state conforms to the environment. This was the reason for bringing the judgment type_ok into our definition of validity, which also gives rise to the new rule (required for the completeness proof)

$$\textit{hazard}\ \frac{}{\Gamma\vdash\{P\ \wedge.\ \text{Not} \circ \text{type\_ok } \Gamma\ t\}\ t\succ\ \{Q\}}$$

indicating that if at any time conformance was violated, anything could happen.

Completeness is proved (basically) by structural induction with the MGF approach discussed in [Ohe99]. This includes an outer auxiliary induction on the number of methods already verified, which requires well-typedness in order to ensure that for any program there is only a finite number of methods to consider. Due to class initialization, an extra induction on the number of classes already initialized is required.

4 Example

To illustrate our approach and for gaining experience how our Hoare logic behaves in practice, we use the following (artificial) example.

```
class Base {
  boolean vee;
  Base foo(Base x) {
    return x;
  }
}

class Ext extends Base{
  int vee;
  Ext foo(Base x) {
    ((Ext) x).vee = 1;
    return null;
  }
}

Base e=new Ext();
try {e.foo(null); }
catch (NullPointerException z) {throw z; }
```

This program fragment consists of two simple but complete class declarations and a block of statements that might occur in any method that has access to these declarations. All important features of Java^{light} are taken into account.

We prove that if there is enough memory to successfully allocate an instance of `Ext`, calling `foo` on this instance with a `null` argument will eventually throw a `NullPointer` exception. This is because, taking dynamic binding into account, `foo` of `Ext` is called, which attempts to assign to the field `vee` through the `null` reference. The resulting exception is caught, but immediately re-thrown.

enough_mem $\longrightarrow$ *tprg*⊢{Normal *Any*} .*tblk*. {*Any* ∧. λ*s*. *tprg*,*s*⊢catch SXcpt NP}

where *Any* = λ(*Y*,σ) *Z*. True
tprg = ([],[(*Base*,*BaseCl*),(*Ext*,*ExtCl*)]@standard_classes)
tblk = Expr(LVar *e*:=**new** *Ext*);
try Expr({ClassT *Base*,ClassT *Base*,IntVir}
Acc (LVar *e*).*foo*({[Class *Base*]}[Lit Null]))
catch((SXcpt **NullPointer**) *z*) (**throw** (Acc (LVar *z*)))

The proof is done in the typical "backward"-style and takes some 80 steps of rule application, simplification, and classical reasoning. About 30% of this deals with class initialization. The rule of consequence is applied four times, and there are four explicit instantiations of schematic assertion variables.

5 Conclusion

We have sketched a Hoare logic for a (rather extensive) subset of Java. This logic seems to be the first one for an object-oriented language that has been proved not only sound, but even complete.

Unfortunately, many of the rules given are quite complex. This is in part due to the result value handling for (sub-)expressions, for which we could not find a simpler solution. But the main point is that Java is an inherently difficult language, taking into account e.g. mutual recursion, dynamic binding, exception handling, and static initialization. As experience with our small example confirms, verifying programs heavily dealing with exceptions and class initialization is tedious, though further machine support might be a relief.

Nevertheless, using an axiomatic semantics like ours for program verifications helps concentrating on the interesting properties of a program (rather than fiddling with details of the state as with an operational semantics), and this experience carries over from procedural to object-oriented languages like Java.

Both for formalizing the Hoare rules and conducting the meta-level proofs, the support of the theorem proving system was indispensable. With some 400 lines of theories and about 1500 lines of (already rather condensed) proof scripts on a highly complex subject, otherwise there would not only be plenty of opportunity for omissions and inaccuracies, but also the sheer amount of inferences to perform by hand would be overwhelming. This is particularly true since within such a non-trivial project many iterations are performed, leading to frequent replay of the large proofs with often subtle, but possibly crucial differences.

References

[dB99] Frank de Boer. A WP-calculus for OO. In *Foundations of Software Science and Computation Structures*, volume 1578 of *LNCS*. Springer-Verlag, 1999.

[HJ00] Marieke Huisman and Bart Jacobs. Java program verfication via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering (FASE'00)*, LNCS. Springer-Verlag, 2000. to appear.

[Ohe99] David von Oheimb. Hoare logic for mutual recursion and local variables. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science (FST&TCS'99)*, volume 1738 of *LNCS*, pages 168–180. Springer-Verlag, 1999.

[ON99] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1999. `http://isabelle.in.tum.de/Bali/papers/Springer98.html`.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994. For an up-to-date description, see `http://isabelle.in.tum.de/`.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.

[Sch97] Thomas Schreiber. Auxiliary variables and recursive procedures. In *TAPSOFT'97*, volume 1214 of *LNCS*, pages 697–711. Springer-Verlag, 1997.

[Sun99] Sun Microsystems. *Java Card Specification*, 1999. `http://java.sun.com/products/javacard/`.

A The remaining rules

$$conseq\ \frac{\forall Y\ \sigma\ Z\ .\ P\ (Y,\sigma)\ Z \longrightarrow (\exists P'\ Q'.\ \Gamma\vdash\{P'\}\ t\succ\ \{Q'\} \wedge (\forall w\ \sigma'.\ (\forall Y'\ Z'.\ P'\ (Y',\sigma)\ Z' \longrightarrow Q'\ (\mathsf{res}\ t\ w\ Y',\sigma')\ Z') \longrightarrow Q\ (\mathsf{res}\ t\ w\ Y,\sigma')\ Z))}{\Gamma\vdash\{P\}\ t\succ\ \{Q\}}$$

$$Xcpt\ \frac{}{\Gamma\vdash\{(\lambda(Y,\sigma).\ P\ (\mathsf{res}\ t\ (\mathsf{arbitrary3}\ t)\ Y,\sigma)) \wedge.\ \mathsf{Not} \circ \mathsf{normal}\}\ t\succ\ \{P\}}$$

$$Super\ \frac{}{\Gamma\vdash\{\mathsf{Normal}\ (\lambda s:\ P\!\uparrow\!:\mathsf{Val}\ (\mathsf{val\_this}\ s))\}\ \mathbf{super}\!-\!\succ\ \{\mathsf{P}\}}$$

$$LVar\ \frac{}{\Gamma\vdash\{\mathsf{Normal}\ (\lambda s:\ P\!\uparrow\!:\mathsf{Var}\ (\mathsf{lvar\ vn\ s}))\}\ \mathsf{LVar\ vn}\!=\!\succ\ \{\mathsf{P}\}}$$

$$FVar\ \frac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ .\mathsf{init}\ C.\ \{Q\} \quad \Gamma\vdash\{Q\}\ e\!-\!\succ\ \{\mathsf{RefVar}\ (\mathsf{fvar}\ C\ stat\ fn)\ R\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ \{C,stat\}e.fn\!=\!\succ\ \{R\}}$$

$$Acc\ \frac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ va\!=\!\succ\ \{\lambda\mathsf{Var}\ (v,f):.\ Q\!\uparrow\!:\mathsf{Val\ v}\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ \mathsf{Acc}\ va\!-\!\succ\ \{Q\}}$$

$$Ass\ \frac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ va\!=\!\succ\ \{Q\} \quad \Gamma\vdash\{Q\}\ e\!-\!\succ\ \{\lambda\mathsf{Val}\ v\!:\!\mathsf{Var}\ (w,f):.\ R\!\uparrow\!:\mathsf{Val\ v}\leftarrow\!:\mathsf{assign\ f\ v}\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ va\!:=\!e\!-\!\succ\ \{R\}}$$

$$Nil\ \frac{}{\{\mathsf{Normal}\ P\!\uparrow\!:\mathsf{Vals}\ []\}\ []\doteq\!\succ\ \{\mathsf{P}\}}$$

$$Cons\ \frac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e\!-\!\succ\ \{Q\} \quad \Gamma\vdash\{Q\}\ es\doteq\!\succ\ \{\lambda\mathsf{Vals}\ vs\!:\!\mathsf{Val}\ v:.\ R\!\uparrow\!:\mathsf{Vals\ (v:vs)}\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e\!:\!es\doteq\!\succ\ \{R\}}$$

$$NewC\ \frac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ .\mathsf{init}\ C.\ \{\mathsf{Alloc}\ \Gamma\ (\mathsf{CInst}\ C)\ id\ Q\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ \mathbf{new}\ C\!-\!\succ\ \{Q\}}$$

where $\mathsf{Alloc}\ \Gamma\ tag\ f\ P \equiv \lambda(Y,(x,s))\ Z.\ \forall\sigma'\ a.\ \Gamma\vdash(f\ x,s)\ -\mathsf{halloc}\ tag\!\succ\!a\!\rightarrow\ \sigma'\longrightarrow\ (P\!\uparrow\!:\mathsf{Val\ (Addr\ a)})\ (\mathsf{Y},\sigma')\ \mathsf{Z}$

$$NewA\ \frac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ .\mathsf{init\_comp\_ty}\ T.\ \{Q\} \quad \Gamma\vdash\{Q\}\ e\!-\!\succ\ \{\lambda\mathsf{Val}\ i:.\ \mathsf{Alloc}\ \Gamma\ (\mathsf{Arr}\ T\ (\mathsf{the\_Intg}\ i))\ (\mathsf{check\_neg}\ i)\ R\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ \mathbf{new}\ T[e]\!-\!\succ\ \{R\}}$$

$$\textit{Cast}\ \frac{\Gamma\vdash\{\text{Normal } P\}\ e\text{-}\succ\ \{\lambda\text{Val } v\text{:. } Q\uparrow\text{:Val v}\leftarrow\text{:}\lambda(\text{x,s}).\ (\text{raise\_if } (\neg\Gamma,\text{s}\vdash\text{v fits T})\ \texttt{ClassCast}\ \text{x,s})\}}{\Gamma\vdash\{\text{Normal } P\}\ \text{Cast } T\ e\text{-}\succ\ \{Q\}}$$

$$\textit{Inst}\ \frac{\Gamma\vdash\{\text{Normal } P\}\ e\text{-}\succ\ \{\lambda\text{Val } v\text{:. } \lambda s:\ (Q\uparrow\text{:Val (Bool (v}\neq\text{Null}\wedge\Gamma,\text{s}\vdash\text{v fits RefT T}))))\}}{\Gamma\vdash\{\text{Normal } P\}\ e\ \texttt{instanceof}\ T\text{-}\succ\ \{Q\}}$$

$$\textit{Body}\ \frac{\begin{array}{c}\text{the (cmethd } \Gamma\ C\ sig) = (md,\ \_,\ \_,\ blk,\ \text{res})\\ \Gamma\vdash\{\text{Normal } P\}\ .\text{init } md.\ \{Q\}\quad \Gamma\vdash\{Q\}\ .blk.\ \{R\}\quad \Gamma\vdash\{R\}\ \text{res}\text{-}\succ\ \{S\}\end{array}}{\Gamma\vdash\{\text{Normal } P\}\ \text{Body } C\ sig\text{-}\succ\ \{S\}}$$

$$\textit{Expr}\ \frac{\Gamma\vdash\{\text{Normal } P\}\ e\text{-}\succ\ \{\lambda w\text{:. } Q\}}{\Gamma\vdash\{\text{Normal } P\}\ .\text{Expr } e.\ \{Q\}}$$

$$\textit{Comp}\ \frac{\Gamma\vdash\{\text{Normal } P\}\ .c_1.\ \{Q\}\quad \Gamma\vdash\{Q\}\ .c_2.\ \{R\}}{\Gamma\vdash\{\text{Normal } P\}\ .c_1;c_2.\ \{R\}}$$

$$\textit{If}\ \frac{\Gamma\vdash\{\text{Normal } P\}\ e\text{-}\succ\ \{P'\}\quad \forall b.\ \Gamma\vdash\{P'\uparrow\text{:Bool=b}\}\ .(\text{if b then } c_1 \text{ else } c_2).\ \{\text{Q}\}}{\Gamma\vdash\{\text{Normal } P\}\ .\texttt{if}(e)\ c_1\ \texttt{else}\ c_2.\ \{Q\}}$$

$$\textit{Throw}\ \frac{\Gamma\vdash\{\text{Normal } P\}\ e\text{-}\succ\ \{\lambda\text{Val } a\text{:. } Q\leftarrow\text{:}\lambda(x,s).\ (\text{throw } a\ x,s)\}}{\Gamma\vdash\{\text{Normal } P\}\ .\texttt{throw}\ e.\ \{Q\}}$$

$$\textit{Try}\ \frac{\begin{array}{c}\Gamma\vdash\{\text{Normal } P\}\ .c_1.\ \{Q\}\\ \Gamma\vdash\{(Q\ \wedge.\lambda\sigma.\ \Gamma,\sigma\vdash\text{catch } C)\ ;.\ \text{new\_xcpt\_var } vn\}\ .c_2.\ \{R\}\\ \Gamma\vdash\{Q\ \wedge.\lambda\sigma.\ \neg\Gamma,\sigma\vdash\text{catch } C\}\ .\text{Skip}.\ \{R\}\end{array}}{\Gamma\vdash\{\text{Normal } P\}\ .\texttt{try}\ c_1\ \texttt{catch}(C\ vn)\ c_2.\ \{R\}}$$

$$\textit{Fin}\ \frac{\begin{array}{c}\Gamma\vdash\{\text{Normal } P\}.c_1.\{\lambda(Y,(x,s)).\ (Q\uparrow\text{:Xcpt x})\ (\text{Y,(None,s)})\}\\ \Gamma\vdash\{\text{Normal } Q\}.c_2.\{\lambda\text{Xcpt } x'\text{:. } R\leftarrow\text{:}\lambda(x,s).\ (\text{xcpt\_if } (x'\neq\text{None})\ x'\ x,s)\}\end{array}}{\Gamma\vdash\{\text{Normal } P\}\ .c_1\ \texttt{finally}\ c_2.\ \{R\}}$$

A Component Retrieval System Using PVS

Makarand Patil and Perry Alexander

University of Kansas, Lawrence KS 66046, USA

Summary. Component reusability is well known concept used for using existing software components to speed up system design and development. Along with expediting the software development cycle, this also provides a useful methodology for synthesizing large component architectures by using existing components, that are already implemented and tested. Formal specification matching techniques for component retrieval use axiomatic style specifications to formally represent the semantic properties of the component. The specification matching is done using extensive theorem proving over the pre- and post- conditions. It is vital that a component retrieval methodology be both theoretically sound and practically useful. However, current techniques that emphasize theoretical soundness tend to be too expensive in practice, while practical techniques tend to sacrifice semantic rigor. The existing techniques of theorem proving to establish reusability prove to be impractical for a very large library of components. In this paper, we describe the implementation details of a specification-based component retrieval system that circumvents the above problem. The system uses a feature-based component classification scheme to effectively limit the amount of theorem proving required at query time. It also presents the automated theorem proving technique employed using PVS.

1 Introduction

Building complex systems from simpler components is a fundamental process in software engineering. Components provide abstraction, that enables the design of huge complex systems by composing them as the integration of simpler components. However, conventional design processes often ignore existing components that are already implemented and tested. Those that try to reuse existing components are mostly incapable of finding the right components that match the problem specification. The performance and reliability of this bottom-up design process can be considerably improved by using a component reuse framework. Reusing components leads to a more reliable design and at the same time, it speeds up the software development cycle. Formal methods for component retrieval rely on specification matching algorithms to establish the reuse potential of components. Specification matching is preferred over signature matching techniques that ignore the semantics of the component specification. However, one of the major problems of specification matching for component retrieval is the computationally expensive task of theorem proving over a large library of components. One obvious solution to the above problem is to devise a technique that reduces the number of proofs required at query time. A feature based component classification scheme is used to classify the components and use this classification to do a specification matching over a small number of components as opposed to the entire library [3]. The work presented here is the implementation details of the component retrieval technique [3].

Our goal is to create a fully automated software reuse framework which can be used over all engineering domains. The automated theorem proving ability gives the additional advantage that the tool does not entail the users to have any prior knowledge of theorem provers. In this paper, we describe the implementation of this retrieval mechanism built and tested over a sample library of **list** components specified in Rosetta [6]. The advantage of using rosetta - a system level design language, is that the reuse technique can be used independent of the domain of the application.

2 Background Theory

A software reuse framework consists of a component retrieval phase followed by a component adaptation phase. Figure 1 shows the block diagram of the entire reuse framework. This paper describes the

implementation of the component retrieval phase. It is important to understand the need for component classification before using the specification matching algorithm.

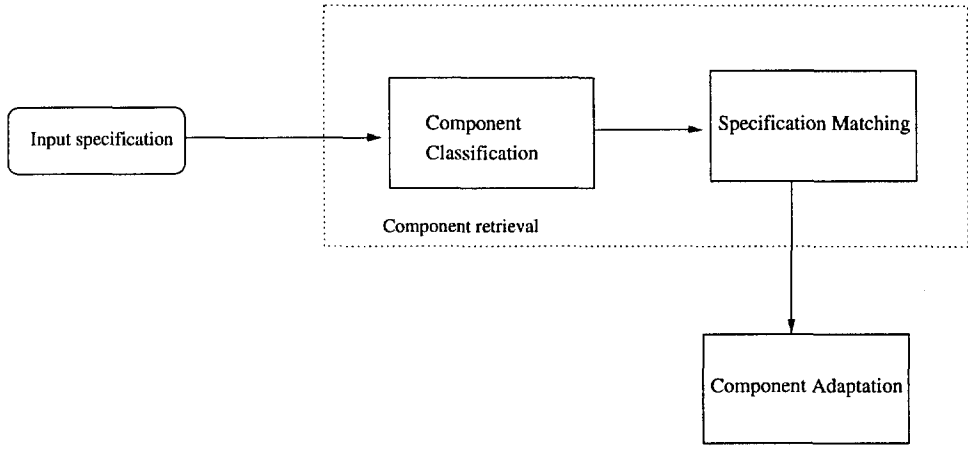

Fig. 1. A block diagram of the entire component reuse framework

The component classification phase improves the efficiency of the entire reuse framework and also make specification matching practical over a large library of components. This is because the specification matching is done over only the components retrieved by the classification mechanism, as opposed to the entire library. The essential elements that make up the retrieval system are as follows:

- A library of components specified in a formal specification language
- *Features* with which the components are classified
- An automated theorem proving methodology for component classification and specification matching

The first requirement is satisfied by using a formal specification language like Rosetta, [6], a system level design language to represent components in terms of their pre- and post conditions. Component classification and theorem proving are explained in following subsections.

2.1 Component classification

Classifying components is a key feature that improves the performance of the reuse system. It relies on the property that similar components can be grouped together and components within the same group have a much greater potential to match each other than with components from other groups. Thus, there is no need for the theorem prover to prove and evaluate specification matching over the entire library of components. Theorem proving is only restricted to *similar* components i.e. the ones with the same feature set. Given a component specification in Rosetta, the objective is to assign all the possible **features** to it [4]. The features are predefined in a *feature-base*. Figure 3 shows a set of typical features that can be assigned to **list** components. Features are logical predicates that have a certain property associated with them. Any component that satisfies this property will be assigned that feature. Mathematically this is equivalent to a set of predicate/name pairs (ϕ, f_i), such that each feature name f is unique and ϕ is the condition associated with that feature.

Definition 1. A *feature set is a collection of all the features that can be assigned to that component.* $FS(C) = \{f_i \mid Ic \wedge Oc \Rightarrow \phi\}$

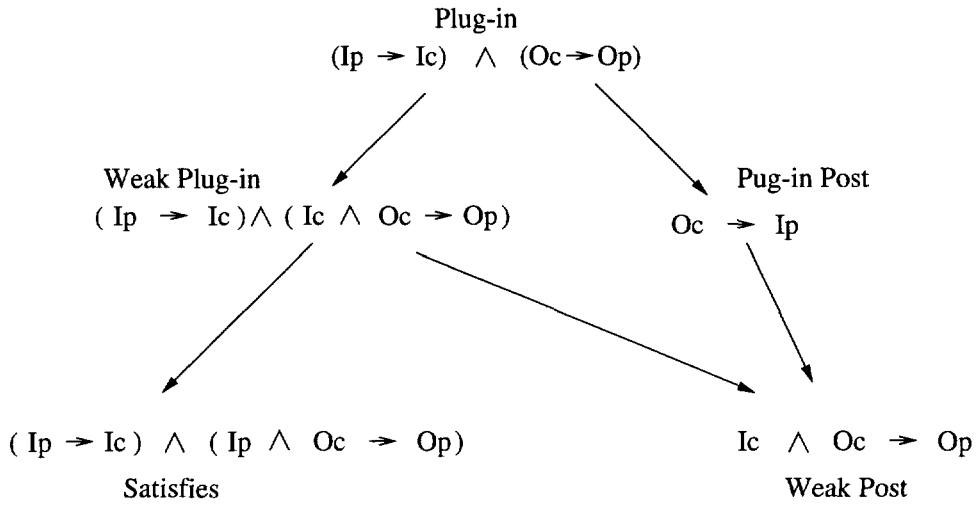

Fig. 2. Specification matching criterion to evaluate reusability

For any given component C and a problem specification P,
if $match(C, P)$ *then* $FS(C) \subset FS(P)$
where *match(C,P)* indicates that C matches P under any of the reuse matches in figure 2.

Although this is a logically weak assertion [4], it is not so clear as to how it may effect the precision of retrieval without actually building and testing the system. Since the primary objective of the classification scheme is to filter out unwanted components and give a set of components that have a greater potential for reuse, this assumption is acceptable in our case.

Given below is an example to illustrate the above scheme: Consider a component **CompSearch** where I and O represent its pre- and post-conditions respectively.

$$I = defined(reclist, elemkey) \wedge isinput(reclist) \\ \wedge isinput(elemkey) \wedge isoutput(elem)$$
$$O = contains(reclist, elem) \wedge Key2Rec(elemkey) = elem$$

Assume that the functions used are predefined in a domain theory. Given a possible feature definition of **Select** in figure 3, it can be observed that

$$I \wedge O \Rightarrow Exists(x : T3, y : T2) : isinput(x) \wedge isoutput(y) \wedge contains(x, y)$$

Thus **Select** can be assigned to the component.

2.2 Specification matching

The process of specification matching is greatly improved with the use of the feature based component classification scheme described above. Using the classification scheme, the conventional requirement of theorem proving over all the components in the library to establish their reuse potential is clearly obviated. Instead the specification matching is restricted to only the components that have the same feature as the problem specification. This follows from the fact that only the components with similar features have the potential to match the needed functionality with any of the specification matching criteria. The other components which have no common feature are automatically skipped by the classification mechanism. With the reduced theorem proving requirements, the specification matching process can be fully automated thereby providing fast and reliable results.

```
features [T1,T2 : TYPE+] : THEORY
 BEGIN

 T3: TYPE = list[T1]

 Importing domain_theory[T1,T2]
 Importing comp_search[T1,T2]

  Select: Theorem
  (Exists (x: T3,y:T2) :
   isinput(x) and isoutput(y) and contains(x,y))

  Nonmember: Theorem
  (Exists (x:T3, y:T2) :
   isinput(x) and isoutput(y) and not contains(x,y))

  Build: Theorem
  (Exists (x:T2 , y:T1) :
    isoutput(x) and isinput(y) and x = Key2Rec(y))

  Filter: Theorem
  (Exists (x,y:T3) :
    isinput(x) and isoutput(y) and
    (forall (z: T2):  contains(y,z) => contains(x,z)))
End features
```

Fig. 3. Example PVS theory for some list features

3 System Overview

The entire component retrieval system is shown within the dotted lines in figure 4. Components are specified in terms of pre- and post- conditions using Rosetta . After this file is successfully parsed, the parser generates an interface file that has all the information needed by the retrieval process. This file is the input to the retrieval system. The interface information is used to create a PVS theory with PRE and POST axioms. This PVS specification can be referred to as the problem specification. The next task is to classify this problem specification using the feature based classification scheme described earlier. The component classification mechanism is implemented using the PVS theorem prover, the feature definitions in PVS and the feature specific and/or generic prover strategies. After the problem specification is classified, its feature set is used to query the component library and retrieve the components that have the same features. The retrieved components are evaluated for the user specified specification matching criterion using the theorem prover. The output of the retrieval system are the components that match the problem specification with the matched criteria. The criteria can be relaxed in cases when the match results in few or no results. The entire process is automated using automated theorem proving in PVS.

4 Theorem proving for feature assignment

The theorem prover is a key component in the feature based classification process explained above. Classifying a component translates to creating its *feature set*. The feature sets of all the components classified so far are stored in a *feature-base*. The *feature-base* acts like a database and can be queried to get a list of components for a given feature. Thus, once a component is classified, its feature set elements become queries to the feature-base to retrieve potential reuse components that will be inputs to the specification matching process. Theorem proving to classify components needs a problem specification

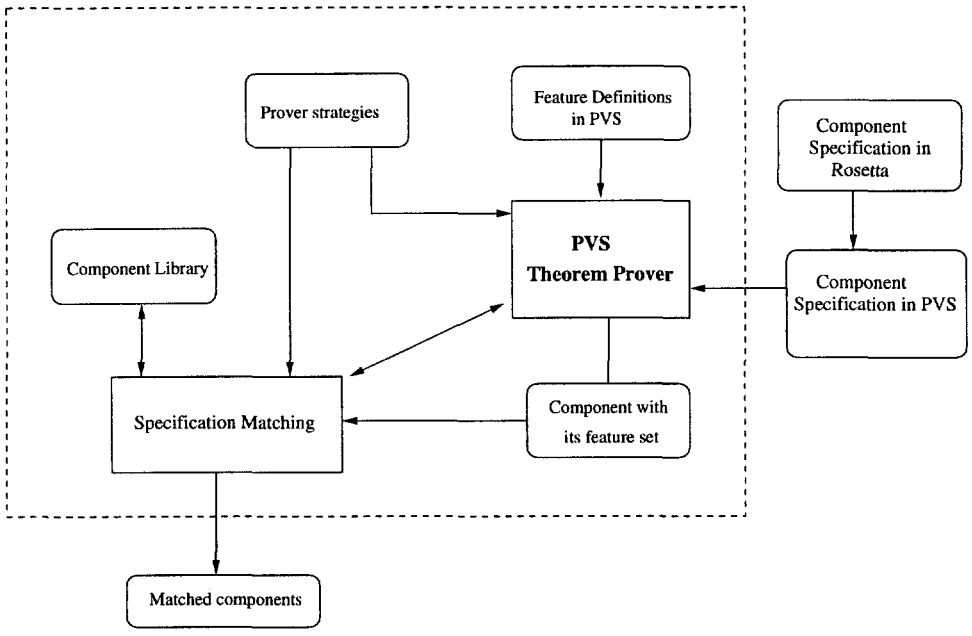

Fig. 4. Feature based component classification

in PVS with its *PRE* and *POST* axioms. A sample specification for the **Search** component is given in figure 6. Feature set definitions are also expressed as theorems in a separate feature theory. The theorem prover attempts to prove all the feature theorems using the component theory axioms. If the proof is successful, it implies that the corresponding feature can be assigned to that component.

Given below is a example to describe the process mentioned above: Consider a **List** domain and a sample component **listsearch**. The rosetta specification is shown in figure 5. The definitions for types and functions are assumed to be provided in the **list** domain. The Rosetta specification is converted into an equivalent PVS theory as shown in figure 6

```
use listdomain;
facet comp_search (reclist:: in listRec ;
                   elem:: out Item ;
                   elemkey :: in Key) is
begin logic
 pre:  defined(reclist,elemkey) ;
 post: contains(reclist,elem) and Key2Rec(elemkey) = elem ;
end;
```

Fig. 5. Rosetta specification for Search Component

The two axioms represent the pre- and post- condition for search component. The next step is to classify the component by assigning features from the feature set using PVS for theorem proving. The following subsection describes the modifications made to PVS to automates this theorem proving process.

4.1 PVS as the theorem prover

PVS provides a classical, typed high order logic specification language along with a powerful automated deduction mechanism. It has a set of pre-defined inference procedures that can be combined to generate

```
Comp_search[Key:TYPE+,Item:TYPE+] : THEORY
 BEGIN
  Importingdomain_theory[Key,Item]
  reclist: list[Key]
  elem: Item
  elemkey  : Key

 requires_SEARCH : Axiom
  defined(reclist,elemkey) and isinput(reclist)
  and isinput(elemkey)
  and isoutput(elem)

 ensures_SEARCH : Axiom
  contains(reclist, elem) and Key2Rec(elemkey) = elem
 END Comp_search
```

Fig. 6. PVS theory for Search Component

high-level proof strategies. The need in our case, is to automate the theorem proofs. To automate this process, the potential prover strategies are identified. PVS stores the proofs for a given theory in a **prf** file. Thus the proofs for each theorems are stored and can be rerun without the need to reprove the theorems. In our system instead of PVS generating the **prf** file after a manual theorem proving process, we generate this file before theorem proving with the adequate theorem proving strategies. The strategy can be generic or can be written specific to a feature. PVS always reads this file whenever it attempts to prove the entire theory. It then retries all the proofs again using the strategies specified in the **prf** file. The prover is invoked in a lisp code by using the command **prove-pvs-file**. Lisp is chosen as the language to enable smooth integration with lisp environment provided by PVS. The lisp code can be automatically loaded while starting PVS.

4.2 Issues involved with automated theorem proving

One of the main problems faced with automated proofs is the instantiations of quantified variables during the proofs. In cases where more than one instance of the variable of the same type are available it is imperative that both instances be substituted in the proofs. Relying on the theorem provers primitive rewrite procedures may lead to erroneous instantiations and the proof may fail. The above problem is to be tackled by having *dynamic strategies*. The strategies are created at run-time wherein all the possible variable instantiations are explicitly specified. Using the **try** construct it is possible for the theorem prover to attempt more than one strategy to prove the theorem.

The results obtained after the theorem prover is done, are stored in the status buffer. The buffer is scanned for proved theorems (possible features) by checking the status of a proof. The status is displayed as "proved-complete" only if the theorem and all the TCC's are successfully proved.

In our *list component* example, the feature set theory used for the sample **list** components is shown in figure 3. The lemma, instantiate and grind strategies of PVS are combined and written in the **pvs-strategies** file. The corresponding **prf** file that is recognized by PVS prover is updated with the new strategy. For the **CompSearch** component it can be seen that the **Select** and **Build** will be assigned to it.

4.3 Proof strategies

The component classification scheme assumes that the prover strategy is powerful and sufficient to prove all the features. Inadequate proof strategy may fail to take into account certain features and this adversely affects the *recall* of component retrieval. The precision of retrieval will depend upon the specification

```
(defstep TOP (&optional otheraxiom
 (try
 (try (STGY1) (fail) (skip))
 (SKIP)
 (STGY2))))
```

Fig. 7. PVS proof strategy

matching technique used. An example of the prover strategy used is shown in figure 7. The *try* construct provided by PVS is very useful in combining existing strategies to get a powerful generic strategy. **TOP** is a generic strategy which combines two other strategies **STGY1** and **STGY2**. TOP applies STGY1 to the theorem and if this doesn't prove the theorem, it tries again with strategy STGY2. The PVS **GRIND** strategy is particularly useful for most of the proofs.

5 Related work

[7] provided a foundation for studying the more general activity of specification matching. Our work uses the specification matching criteria described in this work. The NORAM/HAMMR [1] deductive retrieval tool built by Fischer and Schumann uses a model checker to filter our components to identify potential reusable components. The model checker filter mechanism is replaced by the theorem prover based component classification scheme in our system.

The Inquire retrieval mechanism [5], within the Inscape environment supports retrieval based on component specifications. The specification language used in their case is restricted to a set of preconditions and post conditions. The technique presented here is more flexible with the use of first-order logic to represent features. The features are not the complete representation of the components but represent certain properties that can be satisfied by the component specification.

Deductive program synthesis also use a formal technique to automate the software reuse process. The AMPHION system [2] successfully used deductive synthesis to synthesize software from a subroutine library for solar system geometry. The system specifies components as mathematical functions and their behavior is captured via a set of axioms. A program is synthesized by proving that, for any valid input, there exists an output that satisfies the specification.

6 Conclusion

Software reuse and formal specification are two methodologies that can have a significant impact on the software productivity and reliability. We presented a software retrieval system that matches semantics associated with component specifications to classify them and then use specification matching to establish their reuse potential. This reduces the computationally intensive and practically impossible need of theorem proving over all the library components. Features are assigned to the components based on specific necessary conditions satisfied by the component specification. The entire process is automated by encoding the functionality in a lisp code which directly interacts with the PVS theorem prover. The results of empirical evaluation of this technique [4], shows that this technique can provide retrieval performance comparable to the existing methods. The benefits of this technique are the efficiency and the speed of the retrieval process provided at a high level of automation.

Our future work will consist of using this feature based classification mechanism to do a specification match of the retrieved components and establish their reuse potential. Much of the automated theorem proving techniques required for specification matching can be adopted from the implemented classification scheme. Also a software reuse system essentially consists of a retrieval and adaptation phase. A long term goal can be to integrate a component adaptation system with the retrieval mechanism to build a fully integrated and automated tool for software reuse.

References

1. B. Fischer and J. Schumann. NORA/HAMMR: Making deduction-based software component retrieval practical. In *Proc. CADE-14 Workshop on Automated Theorem Proving in Software Engineering*, July 1997.
2. M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A formal approach to domain-oriented software design environments. In *Proceedings of the 9th Knowledge-Based Software Engineering Conference*, pages 48–57, Sept. 1994.
3. J. Penix. *Automated Component Retrieval and Adaptation Using Formal Specifications.* PhD thesis, University of Cincinnati, 1993.
4. J. Penix and P. Alexander. Efficient specification-based component retrieval. In *Automated Software Engineering*, volume 6, pages 139–170. Kluwer Academic Publishers, 1999.
5. D. E. Perry and S. S. Popovitch. Inquire: Predicate-based use and reuse. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pages 144–151, September 1993.
6. SLDG Lab. System level design group - the rosetta homepage. World Wide Web: `http://www.ittc.ukans.edu/Projects/SLDG/rosetta`
7. A. M. Zaremski and J. M. Wing. Specification matching of software components. In *3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Oct. 1995.

Proof generalization and proof reuse

Olivier Pons[1],[2]

[1] INRIA Sophia Antipolis
[2] IIE, CNAM
pons@cnam.fr

Abstract. This paper presents some experiments about the notion of generalization in type theory based proof assistants. We propose a mechanism which, starting from a proved theorem, makes it possible to get a less specific result that can be instantiated and reused in other contexts.

1 Introduction

Like the mathematician, the user of a proof assistant sometimes wants, even needs, to generalize previously obtained results to allow their reutilization in other developments. To do so, it should be shown that the theorems which have been proved remain true in a weaker context. For example, the basic results about addition on $\mathbb{N}$ can be generalized to any set provided with a structure whose internal laws satisfy the axioms of abelian group.

Thus, it is relevant to find which parts of a proof can be generalized for a later reuse, to understand the analogies between proofs, and, to study how those similarity can be used to help to develop new theories.

Generalization and reasoning by analogy have been intensively studied in Artificial Intelligence and in automatic systems, in which they are used to generate intermediate lemma in order to avoid diverging proof search. But little work has been done in the framework of interactive proof systems based on type theory such as Coq[INR96],Nuprl[CAB+86] or Lego[LEG].

The idea of generalization has been introduced by Plotkin[Plo69] via the notion of anti-unification. The problem is, given two terms t_1 and t_2 to find the tuple (g, θ, σ) where g is a term and θ and σ two substitutions such as $\theta(g) = t_1$ and $\sigma(g) = t_2$. Finding the most specific generalization of two first order terms is now a well understood problem. But generalizing of the higher order terms is more difficult. In [Pfe91b] Pfenning studies anti-unification for a subset of the terms of the Calculus of Constructions and shows some examples of experiments made in LF[Pfe91a]. Hasker and Reddy [HR92] have been interested in generalizing higher order terms with a categorical approach, and show that in the case of the second order, the complete set of most specific generalizations can be computed.

Less ambitious, Kolbe[KW94] proposes to abstract the functional terms. We followed the same way. Thus, we do not propose to find the least general generalization of two terms but simply to help the user to define generalizations of

theorems and proofs already written without assuming future instanciations of these theorems. This is to say that given a tuple (s, t, l) where s is a type, t is a term of type s, and l is a set of free identifiers occurring in s we want to find the smallest set l' of identifiers containing l such as if s' and t' are respectively the results abstracting the elements of l' in s and t, t' being of type s'. (The practical meaning of l' s' and t' will be highlighted in the first example).

This paper introduces the practical experiments done in this field. We do not present a complete achievement but we have tried to show what can be made and to trace the broad outline of an implementation of interesting tools. These experiments have been done using Coq but the idea is generic and could be adapted for any system based on type theory that represent theorems as types and proofs of these theorems as terms having the appropriate type.

2 A quick overview

2.1 Abstraction

As an example, we will work on the following lemma about multiplication on natural integers :

```
Lemma mult_permute :
    (n,m,p:nat) ((mult n (mult m p))=(mult m (mult n p))).
Proof.
Intros;Rewrite -> (mult_assoc_l m n p);Rewrite -> (mult_sym m n);Auto.
Qed.
```

We wish to generalize the statement of the theorem `mult_permute` to get a new statement in which the operator will not any more be `mult` but an unspecified operator. For that we will abstract the identificator `mult` in the statement of `mult_permute`. The generalized statement[1] we obtain is :
`(f:nat->nat->nat)(n, m, p:nat)((f n (F m p))=(f m (more n p))).`

But this statement is not true any more in the general case.

The problem is to find the properties which the operator `f` must check in order to assure that the new statement is still a theorem. Here, intuitively the statement will remain true for any associative and commutative function on integers: Although these two properties do not appear in the statement of `permute_mult` they appear in the proof script. But in the general case, to find which properties must be checked by the operator which is abstracted, it is necessary to examine *the proof term of* the initial theorem. We will also use this term to build a proof of the generalized statement. Thus let us observe the proof term of `permute_mult`:

[1] in which we denote `f` the abstracted operator to avoid the confusion with `mult`

```
PROOF TERM OF PERMUTE_MULT:
     [n,m,p:nat]
       ( eq_ind_r nat (mult (mult m n) p)
         [n0:nat](mult n (mult m p))=n0
         (eq_ind_r nat (mult n m)
           [n0:nat](mult n (mult m p))=(mult n0 p)
           ( mult_assoc_l n m p) (mult m n) ( mult_sym m n))
         (mult m (mult n p)) ( mult_assoc_l m n p))
```

In addition to `mult`, `nat`, and `=`, which already appeared in the initial statement, the free identifiers which appear in the proof are **eq_ind_r**, **mult_assoc_l** and **mult_sym**. Among the latter, it is necessary to determine those which are related to the identifier `mult` which has been abstracted in the statement.

Thus, we seek their statements (*i.e.* their type) in the current environment. We find respectively:

- `(A:Set; x:A; P:(A->Prop))(P x)->(y:A)y=x->(P y)`
- `(n,m:nat)(mult n m)=(mult m n)`
- `(n,m,p:nat)(mult n (mult p))=(mult (mult n m) p).`

It appears that only the last two statements refer to the identifier `mult`. Thus, we deduce, that the only properties of the multiplication which are used in the proof are symmetry and left associativity which correspond to `mult_sym` and `mult_assoc_l` [2].

Thus, it is also necessary to abstract these two identifiers in the generalized statement in order to express the constraints on the operator `f`. We get the following statement:

```
Lemma generalized_permute :
(f:nat->nat->nat)
(f_assoc_l :(n,m,p:nat) (f n (f m p))=(f (f n m) p))
(f_sym :(n,m:nat)(f n m)=(f m n))
(n,m,p:nat)
((f n (f m p))=(f m (f n p))).
```

[2] The proof of this theorem is obtained by abstracting `mult`, `mult_sym` and `mult_assoc_l` in the initial proof term. This leads to the following term :

On our example these two identifiers also appear in the script, this is not always the case.

```
PROOF TERM OF GENERALIZED_PERMUTE:
Proof [f:nat->nat->nat]
          [f_assoc_l:(n0,m0,p0:nat)(f n0 (f m0 p0))=(f (f n0 m0) p0)]
          [f_sym:(n0,m0:nat)(f n0 m0)=(f m0 n0)]
           [n,m,p:nat]
           (eq_ind_r nat (f (f m n) p) [n0:nat](f n (f m p))=n0
             (eq_ind_r nat (f n m) [n0:nat](f n (f m p))=(f n0 p)
               (f_assoc_l n m p) (f m n) (f_sym m n)) (f m (f n p))
             (f_assoc_l m n p))
```

2.2 Instanciations:

We now check that we can instanciate `generalized_permute`, with addition and the proof of its associativity and the commutativity, to obtain a more specific instance of the theorem. We get:

```
Lemma plus_permute :
   (n,m,p:nat) ((plus n (plus m p))=(plus m (plus n p))).
Proof (generalized_permute plus plus_assoc_l plus_sym).
```

In the following section we describe the broad outline of an interactive tool helping the user to carry out such generalizations.

3 A tool to assist the generalization

3.1 Basic principle

Given a statement `S` and a term `P` which is a proof of S, the user provides the name of the function that he wishes to abstract in `S` (to simplify the following explanation we suppose that he selects only one function identifier denoted by `f`). In a graphical environment such as *CtCoq*[BBCanFM96] or *ProofGeneral*[AGKS99], that can be done by a single mouse click. Next the following operations should be made automatically:

1. Find the type `t` associated with the identifier `f`
2. Recover the list of all the free identifiers which appear in the proof term `P` and do not already appear in the statement `S`.
3. Find the types associated with these identifiers.
4. Among these identifiers, select all those whose type refers to the identifier `f` that has been abstracted. We will denote f_{P_i} selected identifiers and t_{P_i} their types (which express the properties of `f` which have been used).
5. Build the statement of the theorem generalized by abstracting `f` and the fP_i in `S`. We obtain a generalized statement of the form:

$$(f : t)(f_{P_1} : t_{P_1}) \ldots (f_{P_i} : t_{P_i}) \ldots S$$

6. Build the proof term associated to this statement by abstracting on P. We get a term of the form :

$$[f : t][f_{P_1} : t_{P_1}] \ldots [f_{P_i} : t_{P_i}] \ldots P$$

3.2 Choosing the names of the abstracted identifiers

As one can see regarding the statement of the lemma `generalized_permute` assigned in the preceding section, the names given to the abstract identifiers are not randomly selected. We propose a small algorithm to name the properties of the abstracted terms.

It is usual to express a property of an operator by the name of this operator followed by the name of the property (often separated by a character *underscore*). To express that it corresponds to an unspecified function, we will use `f` (then f_i if we abstract more than one operator) to denote the abstract operator. Thus, it is also necessary to substitute `f` to all the occurrences of the abstracted operator in the statement.

Once we have recovered the list of identifiers resulting from step 4 in our algorithm, we use their name to build new names by substituting `f` to the name of the operator in the old name of the property; thus `mult_assoc` becomes `f_assoc` which remains significant. Then we substitute `f` to all the occurrences of the abstracted operator in the types of the identifiers recovered at step 3. Finally we substitute `f` and the names of properties formed on `f` in the proof term.

4 More abstraction

So far, we have limited the abstraction to function identifiers. Let us consider again the example of the section 2.1. The permutation property remains true for any function `f:E->E` associative and commutative, whatever the set `E`. Let us try to continue our generalization by abstracting the type of the function arguments. A new statement is obtained :

```
Lemma more_generalized_permute :
(E:Set)
(f:E->E->E)
(f_assoc_l :(n,m,p:E) (f n (f m p))=(f (f n m) p))
(f_sym :(n,m:E)(f n m)=(f m n))
(n,m,p:E)
((f n (f m p))=(f m (f n p))).
```

which is proved by abstracting on the proof term to get:

```
PROOF TERM OF MORE_GENERALIZED_PERMUTE:
Proof [E:Set]
        [f:E->E->E]
         [f_assoc_l:(n0,m0,p0:E)(f n0 (f m0 p0))=(f (f n0 m0) p0)]
          [f_sym:(n0,m0:E)(f n0 m0)=(f m0 n0)]
           [n,m,p:E]
           (eq_ind_r E (f (f m n) p) [n0:E](f n (f m p))=n0
             (eq_ind_r E (f n m) [n0:E](f n (f m p))=(f n0 p)
               (f_assoc_l n m p) (f m n) (f_sym m n)) (f m (f n p))
             (f_assoc_l m n p))
```

4.1 Instantiations :

Now, let us check that we can still instantiate this generalization with integer multiplication:

```
Lemma mult_permute :
    (n,m,p:nat) ((mult n (mult m p))=(mult m (mult n p))).
Exact (more_generalized_permute nat mult mult_assoc_l mult_sym ).
Subtree proved!
```

But we can also instantiate the theorem with other functions such as the addition on multi-variables monomial as defined by Pottier[3]. We get:

```
Lemma mon_permute :
(k:nat)(n,m,p:(mon k))
   ((mult_mon k n (mult_mon k m p))=(mult_mon k m (mult_mon k  n p))).
Exact [k:nat](more_generalized_permute  (mon k)
  ([i:(mon k)][j:(mon k)](mult_mon k i j ))
 ([i:(mon k)][j:(mon k)][l:(mon k)](mult_mon_assoc k i j l))
 ([i:(mon k)][j:(mon k)](mult_mon_com k i j))).
Subtree proved!
```

5 Limitations

We could be tempted to push the abstraction further. Indeed, why limit the statement to the equality relation `eq` The only property of this relation which is used is the principle of rewriting:
`eq_ind_r : (A:Set)(x:A)(P:A->Prop)(P x)->(y:A)y=x->(P y).`
The statement thus remains true for any relation `R` checking this principle. Thus it is possible to get the following generalization:

(http://www-sop.inria.fr/croap/CFC/buch/Monomials.html)

```
Lemma  more_more_generalized_permute :
(E:Set)
(f:E->E->E)
(R:(A:Set)A->A->Prop)
(R_ind_r:(A:Set)(x:A)(P:A->Prop)(P x)->(y:A)(R A y x)->(P y))
(f_assoc_l :((n,m,p:E)(R E (f n (f m p)) (f (f n m) p))))
(f_sym :(n,m:E)(R E (f n m) (f m n)))
(n,m,p:E)
(R E(f n (f m p)) (f m (f n p))).
```

which can be proved by the term below:

```
PROOF TERM OF  MORE_MORE_GENERALIZED_PERMUTE:
Proof [E:Set]
        [f:E->E->E]
          [R:(A:Set)A->A->Prop]
           [R_ind_r:(A:Set)(x:A)(P:A->Prop)(P x)->(y:A)(R A y x)->(P y)]
            [f_assoc_l:(n0,m0,p0:E)
                       (R E (f n0 (f m0 p0)) (f (f n0 m0) p0))]
             [f_sym:(n0,m0:E)(R E (f n0 m0) (f m0 n0))]
              [n,m,p:E]
              (R_ind_r E (f (f m n) p) [n0:E](R E (f n (f m p)) n0)
                (R_ind_r E (f n m) [n0:E](R E (f n (f m p)) (f n0 p))
                  (f_assoc_l n m p) (f m n) (f_sym m n)) (f m (f n p))
                (f_assoc_l m n p)).
```

5.1 Instantiations:

We still check this generalized theorem by instantiating it with the multiplication on the integers:

```
Lemma mult_permute :
   (n,m,p:nat) ((mult n (mult m p))=(mult m (mult n p))).
Intros.
Exact (more2_generalized_permute
         nat n m p mult eq eq_ind_r mult_assoc_l mult_sym).
Subtree proved!
```

But `eq_ind_r` characterizes the equality of Leibnitz, and we will not be able to re-use this lemma with other definite equalities.

Let us consider for example the polynomials such as they were defined by Pottier and Théry[Thé98]. They are represented by the table of their coefficients. The addition of two polynomials is obtained by summing their coefficients of

same degree. An equality relation `eqP` is defined to identify for example 0 and $0 + (0 * x)$ which are different for the equality `eq` .

Thus, we tried to prove `permutes` for the addition of these polynomials, that is to say the following lemma:

```
Lemma plusP_permute :
   (n,m,p:P) (eqP (plusP n (plusP m p)) (plusP m (plusP n p))).
```

But it is not possible to reuse our proof term. We can nevertheless complete the proof. We can try to establish a generalization of `eq_ind_r` such as for example in the case of the polynomials with integer coefficients.

```
Lemma eq_ind_r_int:
( x:(P nat); P1:((P nat)->Prop))
   (P1 x)->(y:(P nat))(eqP nat(eq nat) [x:nat]x=0  y x)->(P1 y)
```

But this result is not true, otherwise taking `[z:(P nat)]x=z)` as `P1` we could prove that `eqP` imply `eq`, which is not true.

We can nevertheless complete the proof of `plusP_permute` by the following script:

```
Intros.
(Apply eqP_trans with (plusP (plusP m n) p); Auto).
(Apply eqP_trans with (plusP (plusP n m) p); Auto).
```

Its proof term, given below, has nothing to do with the one we previously had:

```
Proof: [n,m,p:P]
        (eqP_trans (plusP n (plusP m p)) (plusP (plusP m n) p)
          (plusP m (plusP n p))
          (eqP_trans (plusP n (plusP m p)) (plusP (plusP n m) p)
            (plusP (plusP m n) p) (plusP_associative n m p)
            (eqP_sym (plusP (plusP m n) p) (plusP (plusP n m) p)
              (eqP_sym (plusP (plusP n m) p) (plusP (plusP m n) p)
                (eqP_sym (plusP (plusP m n) p) (plusP (plusP n m) p)
                  (plusP_compl (plusP m n) (plusP n m) p
                    (plusP_commutative m n))))))
          (eqP_sym (plusP m (plusP n p)) (plusP (plusP m n) p)
            (plusP_associative m n p)))
```

Thus, this example shows that it is not always desirable to generalize as much as possible (i.e. to abstract all the free identifiers appearing in a statement).

6 Robustness and problems

In practice, to guarantee that our generalization of a theorem `T` will work it is necessary that its proof refers only to the types of the generalized identifiers but does not use their internal structures (i.e. if the dependence with respect to these terms is opaque). For inductive types, this expresses that we do not reason by case or induction. Moreover, if it is always possible to abstract a recursor, although the preceding example shows that it is not very useful, nothing can be done when a `Fix` or a `Case` appears in the proof term.

This restriction is also true for functional terms, our first examples works only because the proof terms do not directly refer to the structure of `mult`; the properties of `mult` were proved separately and thus, can be abstracted. If these basic properties were used directly in the proof, abstracting would be impossible.

Indeed to prove the majority of the basic properties of functions, we need to access their basic structure. Thus, to prove the commutativity of the addition, we should seek the internal definition of addition to find the rewriting rules which make it possible for example to reduce `(0+x)` to `x`. The reduction of a functional term defined by such rules is called the ι-reduction. That led to very different proof for commutativity of the addition and commutativity of the multiplication on the integer, and to even more different proof for commutativity of the monomial multiplication .

Kolbe and Walter[KW94] studied the reuse of such proofs , but they seem to limit themselves to small examples and in spite of that the result still remains very dependent on the definite structure. Thus, some proofs made on the addition could be reused for the multiplication if one defines it by the rules: `0*m` $\leadsto$ `0` and `(S p)*m` $\leadsto$ `m + (p * m)` but not if one defines it by the system `0*m` $\leadsto$ `0` and `((S p)*m` $\leadsto$ `(m * p) + m)`, which is however equivalent.

7 Restriction in the use of the abstraction.

To circumvent these problems, we will make it possible to abstract a function identifier only if it is opaque[4] or, if we can check that it is not ι-reduced it in the proof (a less restrictive constraint). In the same way, we will authorize to abstract an inductive type only if it corresponds to a "silent type",i.e., if the proof does not contain a reasoning by case or induction on the structure of this type. To check that one does not reduce a term, we can track in the script the tactics which can perform reductions like `Simpl`, `Change` .... In the same way one will look the tactics destructing an inductive type like `Induction` , `Split`, `Case` .... It will also be checked that the constructors of the type appear neither in the script nor in the proof term. But this method is not really reliable simply because the user can define his own tactics.

[4] An identifier is opaque if we can access to its type but not to its proof.

8 Conclusion and perspective

The main idea of this tool is to avoid restarting from scratch when we will develop abstract theories such as for example the groups or rings theories. Indeed in an existing developments many proofs use only some properties of the "objects" on which they work.

We propose to provide a "debug mode" in which each property used in proof will be generalized and stored. When thereafter we have to define a new structure, we will give its abstract properties and by comparing with our storage and, modulo the restrictions of the § 7, we will be able to recover for free all the theorems in the existing developments which are still valid for our structure.

The certification of mathematical algorithms intended for computer algebra must be based on a certified implementation of the basic mathematical structures (groups, rings, algebra etc). In the long run, generalization should make it possible to recover, by generalizing them, the proofs which have been done for an instance of these theories such as integer or polynomials in the case of groups. In relation to the introduction of the concept of module modules in proof assistant, this could make it possible to develop parameterized theories starting from specific instances. This is more or less similar to Siff and Reps's works[SR96] in the field of programming languages, when they try to generate generic *C++* code starting from C code.

References

[AGKS99] David R. Aspinall, Healfdene Goguen, Thomas Kleymann, and Dilip Sequeira. Proof general 2.1, March 1999. http://www.dcs.ed.ac.uk/home/proofgen/ProofGeneral/doc/ProofGeneral_toc.html.

[BBCanFM96] Janet Bertot, Yves Bertot, Yann Coscoy, and Healfdene Goguen a nd Francis Montagnac. *User Guide to the CtCoq Proof Environment.* INRIA, Feb 1996.

[CAB+86] Robert Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harber, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, 1986.

[FH94] Amy Felty and Douglas J. Howe. Generalization and reuse of tactic proofs. *Lecture Notes in Computer Science*, 822:1–15, 1994.

[HR92] Robert W. Hasker and Uday S. Reddy. Generalization at higher types. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania. Available as Technical Report MS-CIS-92-86.

[INR96] INRIA. *The Coq Proof Assistant Reference Manual*, December 1996. Version 6.1.

[KW94] Thomas Kolbe and Christoph Walther. Reusing proofs. In *Proc. of the 11th ECAI*, pages 80–84, Amsterdam, The Netherlands, 1994.

[LEG] The LEGO World Wide Web page. url `http://www.dcs.ed.ac.uk/home/lego`.

[Pfe91a] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe91b] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[Plo69] Gordon D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163, Edinburgh, 1969. Edinburgh University Press.

[RS93] Wolfgang Reif and Kurt Stenzel. Reuse of proofs in software verification. *Foundations of Software Technology and Theoretical Computer Science*, 13, 1993.

[SR96] Michael Siff and Thomas Reps. Program generalization for software reuse: From C to C++. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 21_6 of *ACM Software Engineering Notes*, pages 135–146, New York, October 16–18 1996. ACM Press.

[Thé98] Laurent Théry. A certified version of Buchberger's algorithm. In *Automated Deduction (CADE-15)*, volume 1421 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, July 1998.

Composing Proofs of Security Protocols Using Isabelle/IOA

Oleg Sheyner and Jeannette Wing *

Carnegie Mellon University

Abstract. Complex security protocols require a formal approach to ensure their correctness. The protocols are frequently composed of several smaller, simpler components. We would like to take advantage of the compositional nature of such protocols to split the large verification task into separate and more manageable pieces.
Various formalisms have been used successfully for reasoning about large protocol compositions by hand. However, hand proofs are prone to error. Automated proof systems can help make the proofs more rigorous. The goal of our work is to develop an automated proof environment for compositional reasoning about systems. This environment would combine the power of compositional reasoning with the rigor of mechanically-checked proofs. The hope is that the resulting system would be useful in verification of security protocols of real-life size and complexity.
Toward this goal, we present results of a case study in compositional verification of a private communication protocol with the aid of automated proof tool Isabelle/IOA.

1 Introduction

Today's security protocols require a formal approach to ensure that they satisfy important correctness properties. Traditional ways of verifying correctness by hand are prone to error and require a large investment of human effort and patience. Furthermore, these problems tend to grow worse as the size and complexity of the system being verified both increase. Automated proof tools can help make the proofs more rigorous. Such tools also need a lot of human guidance, and the automation they do provide typically does not scale well with the size of the problem.

The protocols we are interested in are frequently composed of several smaller, simpler protocols. We would like to take advantage of the compositional nature of such protocols to split the large verification task into separate, more manageable pieces. Existing proof systems do not provide a structured environment for compositional reasoning about systems. The goal of our work is to develop such an environment. This environment would combine the power of compositional reasoning with the rigor of mechanically-checked proofs. We would like the resulting system to be useful in verification of security protocols of real-life size and complexity. Toward that goal we have conducted a case study in compositional verification of a private communication protocol with the aid of the automated proof tool Isabelle. This paper describes our experiences with the case study.

I/O automata [Lyn96,LT89] have been successfully employed in hand verification of large reactive systems. I/O automata express reactive distributed systems concisely as compositions of several smaller subsystems. Meta-theorems about compositional properties of I/O automata help prove correctness theorems about the systems they describe.

Lynch has applied the I/O automata formalism to verifying a private communication protocol [Lyn99]. In this paper we take a version of the same protocol and verify its properties

* Authors' contact information: Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, {oleg, wing}@cs.cmu.edu

using the theorem prover Isabelle and I/O automata meta-theory developed by Müller [Mül98]. The protocol is decomposed into two components whose properties are proven separately. The top-level proofs then combine correctness theorems about the components and obtain a correctness proof about the composite protocol. The resulting formal description could be further combined with other security protocol components, and such compositions can be verified in Isabelle using the same compositional reasoning techniques.

The paper is organized as follows. Section 2 gives an introduction to I/O automata and the meta-theorems used in the Isabelle proofs. Section 3 describes two recent efforts to incorporate I/O automata into mechanical theorem provers, PVS [ORSvH95] and Isabelle [Pau94]. Sections 4 and 5 discuss our experiences with verifying a private communication protocol of I/O using the Isabelle theorem prover. In Section 6 we discuss our results.

2 An Introduction to I/O Automata

The Input/Output Automaton (I/O Automaton) model [Lyn96,LT89] is a general model used for formal descriptions of distributed reactive systems. An I/O Automaton A is a state machine in which the state transitions are associated with named actions. The actions are classified as either *input*, *output*, or *internal*. The input and output actions are called *external* actions. We let $ext(A)$ designate the set of external actions of automaton A. External actions are used for communication with the automaton's environment, while the internal actions are visible only to the automaton itself.

The *composition* operator || allows an automaton representing a complex system to be constructed by composing automata representing individual system components. The composition identifies actions with the same name in different component automata. When any component automaton performs a step involving an action π, so do all component automata that include π. The state of the composition is the product of the states of its components.

A triple (s, π, s') is a *step* of an I/O automaton A if A has a transition from state s to state s' via action π. An *execution fragment* of A is a finite or infinite sequence $s_0\pi_0 s_1\pi_1 \ldots$ of alternating states and actions of A, where each subseqence $s_i\pi_i s_{i+1}$ is a step of A. An *execution* of A is an execution fragment whose first state is a start state of A. The *trace* of an execution α is the subsequence γ of α consisting of external actions of A. The set of all traces of A is designated $traces(A)$.

Let γ be a finite (possibly empty) sequence of external actions of automaton A, and let s and t be states of A. The triple (s, γ, t) is a *move* of A (written $s \stackrel{\gamma}{\Rightarrow}_A t$) if there exists a finite execution fragment α of A starting in s and ending in t such that $trace(\alpha) = \gamma$. Thus, a move $s \stackrel{\gamma}{\Rightarrow}_A t$ is a series of state transitions with the externally-visible behavior γ.

For reasoning about correctness properties of I/O automata, we use the notion of *implementation relation*, also called *trace inclusion*.

Definition 21 *Given two I/O automata A and C with sets of identical external actions, we say that C implements A (denoted $C \preceq A$) iff $traces(C) \subseteq traces(A)$.* □

Implementation relations are used to show that a concrete system C safely implements an abstract system A. Typically, A is a specification of *safety* properties we would like the concrete

system to exhibit. Proving the relation $C \preceq A$ guarantees that C exhibits only the external behaviors allowed by the specification A.

Implementation relations can be established by exhibiting *simulation relations* between the concrete and abstract automata.

Definition 22 *Let C and A be I/O automata with identical external actions. A* forward simulation *from C to A is a relation R over states$(C) \times$ states(A) that satisfies the following conditions:*

- *If s is a start state of C, then there is a start state s' of A such that $(s, s') \in R$.*
- *If state s is reachable in C, state $s' \in R[s]$ is reachable in A, $a \in ext(C)$, and (s, a, t) is a step of C, then there is a move $s' \stackrel{a}{\Rightarrow}_A t'$ in A, where $t' \in R[t]$.*

Intuitively, every externally visible step (s, a, t) of automaton C is simulated by a move $s' \stackrel{a}{\Rightarrow}_A t'$ of automaton A. The move must include exactly one external action a, but may include any finite number of internal actions.

We write $C \leq_F A$ when there is a forward simulation from C to A. The utility of forward simulations is established by the following theorem.

Theorem 1. *Let C and A be I/O automata with identical external actions. If $C \leq_F A$, then $C \preceq A$.*

The following theorem defines compositional properties of I/O automata and enables us to reason about individual components of complex systems.

Theorem 2. *Let $C = C_1 || \ldots || C_n$ and $A = A_1 || \ldots || A_n$ be parallel compositions of I/O automata, where $ext(A_i) = ext(C_i)$ and $C_i \preceq A_i$ for every i. Then $ext(A) = ext(C)$ and $C \preceq A$.*

Hence, if we can decompose complex systems C and A into simpler components, we can prove trace inclusion between C and A by proving trace inclusion between individual components and then applying Theorem 2.

3 I/O Automata and Mechanical Theorem Proving

When a trace inclusion proof is attempted using a typical generic theorem prover, many issues crop up. The first question is how I/O automata should be represented in the specification language of the theorem prover. The language may lack expressive power or convenient features because the language is tailored for the theorem prover, rather than the user's needs.

Once the representation has been designed, it is necessary to verify that the representation satisfies the meta-theorems about I/O automata, in particular Theorems 1 and 2. These essential theorems may be difficult to prove for the chosen representation of I/O automata. One possible solution is to supply these and other theorems to the theorem prover in the form of axioms. This approach defeats some of the value of mechanical verification, since we could not be sure that our representation of I/O automata is sound.

If we are verifying a complex composition of multiple smaller automata, each individual automaton has to be hand-translated to the input language of the theorem prover—a laborious process that is prone to error. In our experience, subsequent attempts to prove properties of the system reveal many more errors resulting from faulty translations than errors inherent in the original I/O automata specification.

The process of proving theorems about automata in a theorem prover can be tedious. Prover commands are typically very different from the reasoning steps that humans usually make. Even if the user knows how the high-level proof should go, translating this knowledge into a complete proof in a mechanical prover can be a frustrating experience.

Recent work has addressed these complications and attempted to make automated verification of I/O automata systems more closely resemble hand verification. Archer *et al.* recently developed TAME [AHS98], a high-level interface to the higher-order logic theorem prover PVS for specifying and proving invariant properties of I/O automata models. The TAME interface provides a template for translating I/O automata specifications into the PVS input language. A set of high-level commands lets the user prove invariant properties with the same type of steps that are commonly taken in hand proofs. However, TAME has significant shortcomings as well. There is no natural way to define an I/O automaton type and formalize I/O meta-theory, including the composition operator and theorems about simulations and compositional reasoning. This is due to restrictions in the polymorphic features of the PVS specification language. Hence, TAME is suitable primarily for verifying invariant properties of relatively small systems.

Müller formalized a large part of the basic I/O automata meta-theory using the theorem prover Isabelle [Mül98]. Isabelle's specification language has rich polymorphic mechanisms, making it suitable for concise specifications of I/O automata and associated operators. Müller's meta-theory includes a definition of the composition operator and proofs of Theorems 1 and 2. We used Müller's Isabelle/IOA system for our case study.

4 Trace Inclusion Proofs in Isabelle/IOA

The first step in the verification process is converting I/O automata specification and implementation (written in the traditional precondition-effect style) into the Isabelle input language. This task is reasonably easy because Isabelle/IOA contains the composition operator, an operator for hiding external actions (which helps make automata compatible for composition), and other standard operators from I/O automata theory. It is therefore not necessary to compose automata by hand, or otherwise modify them before doing the translation.

A template for formalizing automata in Isabelle's language is shown in Figure 1. The template assumes that the actions of the automaton have been defined as a datatype `action` in a separate theory named `Action`. To create a specific automaton out of the template, the user must fill in items 1 through 11 (marked in bold numbers in the figure), as follows.

The definition `auto_trans_def` specifies the transition relation on the state of the automaton using set comprehension notation. The relation is a set of triples $tr = (s, \alpha, t)$ satisfying the boolean `case` expression on the action name α. The user fills out items 6 through 8 to set up the transition relation for a specific instantiation of the template. Items 6 and 7 pair an action name with a boolean expression constraining the set of transitions labeled by the action name. All other actions of the automaton follow in item 8, using the same syntax.

```
auto = IOA + Action + ..1.. +
types
    auto_state = ..2..
consts
    auto_asig :: action signature
    auto_trans :: (action, auto_state) transition set
    auto_ioa :: (action, auto_state) ioa
defs
    auto_asig_def "auto_asig == ({..3..}, {..4..}, {..5..})"
    auto_trans_def "auto_trans ==
          { tr. let s = fst(tr);
                      t = snd(snd(tr));
                      α = fst(snd(tr))
                  in
                case α of
                    ..6.. ⇒ ..7.. |
                    .
                    8
                    .
          }"
    auto_ioa_def "auto_ioa == (auto_asig, {..9..}, auto_trans, {..10..}, {..11..})"
end
```

Fig. 1. Template for specifying I/O automata in Isabelle

Finally, the definition `auto_ioa_def` defines the entire I/O automaton as a 5-tuple consisting of the action signature, the set of initial states (item 9), the transition relation, and two types of fairness conditions (items 10 and 11). In this paper we will consider only safety properties, so the fairness conditions will always be empty sets. Section 5.1 contains an example translation of an I/O automaton into Isabelle using the template.

Once the I/O automata have been encoded in Isabelle, the next step is stating and proving invariant properties that will be used later in the implementation proof. A typical hand invariant proof proceeds by induction. For the base case, we show that the invariant holds in all initial states. For the inductive step, we check that each action preserves the invariant property. A similar strategy works in our Isabelle proofs of invariant properties. We have developed an Isabelle tactic (called `simplify_inv_goal_tac`) that takes the invariant goal, applies induction (thereby breaking the goal into subgoals for the base case and for each action) and automatically proves the "trivial" cases. In particular, the cases that do not modify the parts of the state involved in the invariant are proven automatically. After this tactic is applied, the user is left with the task of proving the remaining cases. In each case, the necessary reasoning is localized to the effects of one action, eliminating the need to reason about the entire automaton.

The final step in the implementation proof is exhibiting a simulation relation or a refinement mapping from the states of the implementation to the states of the specification. The proof that a function is a refinement mapping is structurally similar to the proofs of invariant properties. Once again, we apply induction on the length of the execution to the goal and automatically discharge the "trivial" cases among the resulting subgoals. The rest of the subgoals are proven in the manner similar to the hand proof. Each subgoal corresponds to one step of the implementation automaton; the user must exhibit a corresponding move of the specification automaton

and prove that the end states of the implementation step and the specification move are related by the refinement mapping.

When we want to generate a trace inclusion proof between two compositions of automata $C = C_1||\ldots||C_n$ and $A = A_1||\ldots||A_n$, we can take advantage of Theorem 2. Once we have obtained separate trace inclusion proofs for each pair of component automata C_i and A_i separately, we can apply the compositionality theorem to get trace inclusion between C and A. This step is easy, and requires only a side proof that the automata being composed are compatible with each other. We are developing Isabelle tactics that discharge most of this proof automatically.

5 Case Study: Verification of a Private Communication Protocol

We have taken a modified version of a private communication service protocol specified as I/O automata in [Lyn99] and used Müller's Isabelle/IOA to verify secrecy properties of the service. The main point of this exercise is to investigate the feasibility of using the theoretical machinery provided by I/O automata to perform compositional analysis of complex systems in an automated proof environment. A full description of the system appears in the Appendices. In the rest of the paper we give a high-level description of the system and discuss our experiences with Isabelle/IOA.

The private communication service is specified as an I/O automaton PC. The service lets clients exchange messages with each other using an insecure transmission channel. The specification guarantees that messages are delivered at most once, and their content remains secret from the adversary.

The service is implemented using a shared-key cryptosystem and contains a number of automata. Before going out on the *insecure communication channel*, each message passes through an *encoder* automaton and gets encrypted with a key that the encoder shares with a corresponding *decoder* automaton on the receiving side. The decoder decrypts the messages and passes them on to the client. The implementation model also includes a *passive eavesdropper* automaton. The eavesdropper can intercept messages appearing in the insecure channel and also compute new messages (via encryption and decryption functions) from the available information. Using a technique similar to assume-guarantee proofs, the *environment* automaton records our assumptions about the environment in which the service can operate correctly. In particular, the environment must not give away secrets to the eavesdropper.

The shared keys are generated by a *key distribution service*. The full implementation employs a version of the Diffie-Hellman protocol to generate and distribute shared keys. Since the analysis of key distribution is fairly involved, we decompose the implementation into two parts that can be verified independently. Figure 2 shows the structure of an I/O automata composition $PCImpl_1 = IC||Eve||KD||Enc||Dec||Env$ implementing specification PC.

In $PCImpl_1$, KD is a high-level specification, leaving out the details of key distribution and thus simplifying the structure of $PCImpl_1$. The Diffie-Hellman key distribution protocol can now be verified independently of the rest of the private communication protocol. The protocol consists of Diffie-Hellman nodes (one per client) and an insecure channel. Diffie-Hellman nodes exchange several messages over the channel in order to establish a shared key for a pair of clients. Just as in the private communication implementation, there is a passive eavesdropper and and

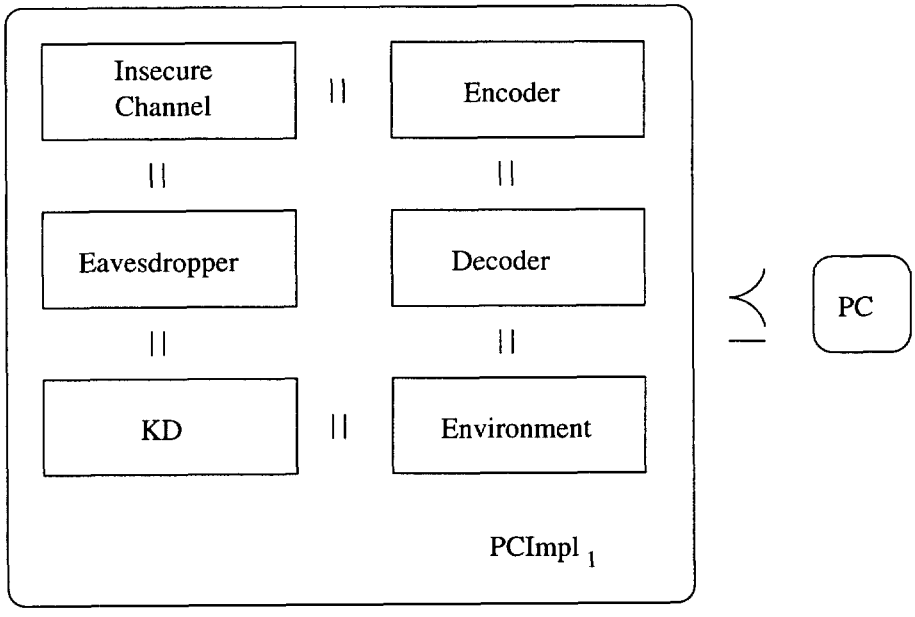

Fig. 2. Composition $PCImpl_1$ implements specification PC

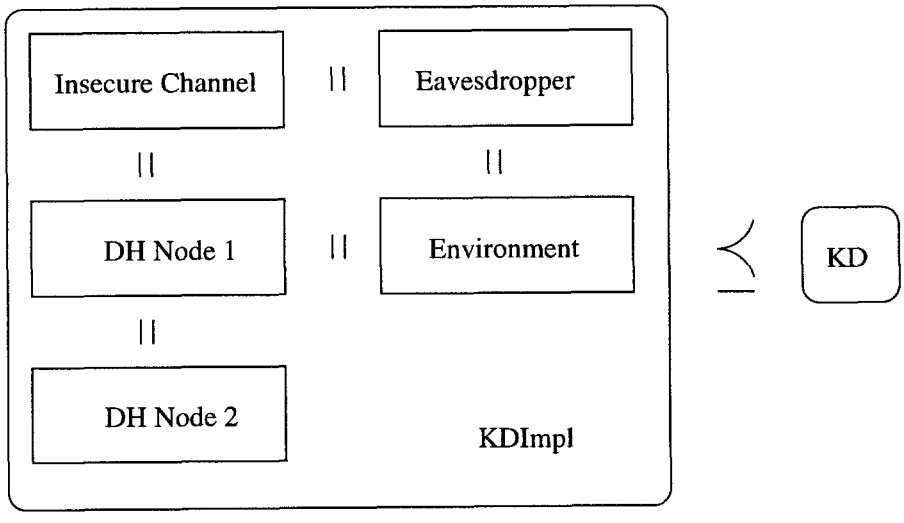

Fig. 3. Composition $KDImpl$ implements specification KD

an environment. Figure 3 shows the structure of an I/O automata composition $KDImpl = DH_1||DH_2||IC||Eve||Env$ implementing specification KD.

For simplicity, we assume (unrealistically) that the key distribution protocol and the private communication protocol have separate insecure channels and eavesdroppers, and the eavesdroppers do not communicate with each other. See [Lyn99] for a more realistic treatment combining the insecure channels and the eavesdroppers.

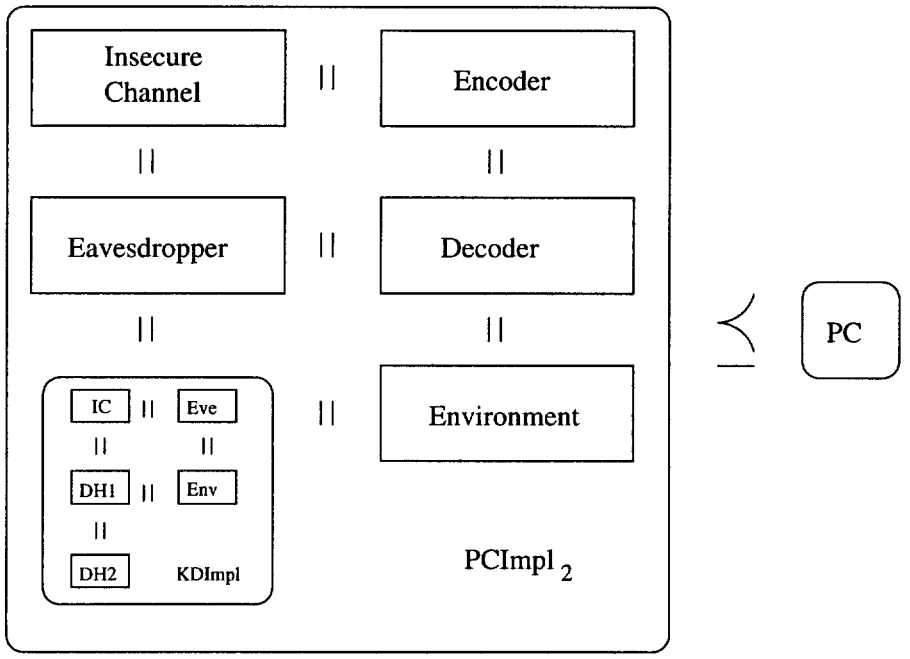

Fig. 4. Composition $PCImpl_2$ implements specification PC

Breaking up the implementation in this manner lets us take advantage of the compositionality theorem about I/O automata (Theorem 2). We prove trace inclusion for compositions shown in Figures 2 and 3 in Isabelle. Theorem 2 then lets us substitute the Diffie-Hellman implementation $KDImpl$ in place of the specification KD while preserving trace inclusion between $PCImpl_1$ and PC. The resulting implementation $PCImpl_2$ is shown in Figure 4.

Below we give a more detailed description of the PC and KD service specifications.

5.1 The Services

In this section, we describe the two services that are implemented by the protocols and verified in this paper. The use of input and output actions provides convenient ways of composing these automata with others, and of describing what is preserved by implementation relationships. These specifications describe only safety properties, although the same methods can be used to handle liveness properties, formulated as *live I/O Automata* [GSSL93].

Private Communication This section contains a specification of the problem of achieving private communication among the members of a finite collection P of clients. The specification expresses three properties: (1) only messages that are sent are delivered, (2) messages are delivered at most once, and (3) none of the messages are revealed by an "adversary." We describe the problem using a high-level I/O automaton specification $PC(U, P, M, A)$, where U is a universal set of data values, P is an arbitrary finite set of client ports, $M \subseteq U$ is a set of messages, and A is an arbitrary finite set of adversary ports. This specification makes no mention of distribution or keys; these aspects will appear in implementations of this specification, but not in the specification itself. The specification simply describes the desired properties, as an abstract machine. As usual for automaton specifications, the properties, listed separately above, are intermingled in one description.

$PC(U, P, M, A)$:

Signature:

Input:
$PC\text{-}send(m)_{p,q}$,
$m \in M$, $p, q \in P$, $p \neq q$

Output:
$PC\text{-}receive(u)_{p,q}$, $u \in U$, $p, q \in P$, $p \neq q$
$reveal(u)_a$, $u \in U$, $a \in A$

States:

for every pair $p, q \in P$, $p \neq q$:
$buffer(p, q)$, a multiset of M

Transitions:

$PC\text{-}send(m)_{p,q}$ T
Effect:
add m to $buffer(p, q)$

$PC\text{-}receive(u)_{p,q}$
Precondition:
$u \in buffer(p, q)$
Effect:
remove one copy of u from $buffer(p, q)$

$reveal(u)_a$
Precondition:
$u \notin M$
Effect:
none

he first two properties listed above, which amount to at-most-once delivery of messages that were actually sent, are ensured by the transition definitions for *PC-send* and *PC-receive*. The third property, privacy, is expressed by the constraint for *reveal*.

The following figure demonstrates the private communication specification translated into Isabelle/IOA. The translation fills in specific information about the specification into the template shown in Section 4.

```
PC = IOA + Action + InfMultiset +
types
  PC_state = "P × P ⇒ U tmultiset"
```

```
consts
    PC_asig :: action signature
    PC_trans :: (action, PC_state) transition set
    PC_ioa :: (action, PC_state) ioa
defs
    PC_asig_def "PC_asig ==
        ((UN m p q. {PC_send m p q}),
        (UN u msg p q. {PC_receive u msg p q}) ∪ (UN u a. {reveal u a}),
        {})"
    PC_trans_def "PC_trans ==
        { tr. let s = fst(tr);
                  t = snd(snd(tr));
                  α = fst(snd(tr))
              in
              case α of
                  reveal u a ⇒ u ∉ M_set |
                  PC_send m p q ⇒
                      (m ∈ M_set) &
                      (t = (λ (p', q').
                                if (p = p') & (q = q') then
                                    s (p', q') + {m}
                                else
                                    s (p', q'))) |
                  PC_receive u msg p q ⇒
                      (u ∈ s (p, q)) &
                      (t = (λ (p', q').
                                if (p = p') & (q = q') then
                                    s (p', q') - {u}
                                else
                                    s (p', q')))
        }"
    PC_ioa_def "PC_ioa == (PC_asig, {λ (p, q). ∅}, PC_trans, {}, {})"
end
```

The state of PC is represented as a function from a pair of clients of type P to a multiset of messages of type U. The definition of the transition relation gives a boolean expression for every triple (s, α, t), where s and t are states and α is an action of PC. The boolean expression includes the precondition of α and relates t to s via the effects of α. Thus, the expression is true if and only if (s, α, t) is a step of PC.

Key Distribution We use a drastically simplified key distribution service, which distributes a single key to several participants. We do not model requests for the keys, but assume that the service generates the key spontaneously. The simplified key distribution problem is specified by the automaton $KD(U, P, K, A)$, where U is a universal set of data values, P is an arbitrary finite set of client ports, $K \subseteq U$ is a set of keys, and A is a finite set of adversary ports.

6 Discussion

The benefits of decomposing large systems into smaller parts for verification are twofold. From the software engineering perspective, formalizing and reasoning about large monolithic systems quickly becomes unmanageable. The number of potential interactions between state components

typically increases exponentially with the size of the state and the size of the transition relation. When the system has more than a few state components, just formulating the necessary invariants can prove to be a daunting task. Compositional reasoning lets us take a modular approach to verification. We can focus on proving properties of self-contained systems of reasonable size and build up a component library for constructing larger systems. Compositionality results let us combine proven properties of components and obtain new results about the larger system without going through the verification process from scratch. One can imagine that somewhat more realistic versions of the *PC* and *KD* services and their implementations could be a part of a library of formalized security and cryptography components.

Decomposition also helps avoid the state explosion problems common to all automated verification tools. Isabelle's simplifier was valuable in reducing the human effort in our verification exercise, but in our experience its running time greatly depends on the size of automata being verified. The table below shows the running time on a set of theorems proven automatically by an identical invocation of the simplifier. Each theorem describes how a transition of an n-automata composition is projected onto the individual components. The table gives the timings for $n \in \{3, 4, 5, 6\}$.

n	3	4	5	6
time	5.5 *sec*	27.9 *sec*	3.8 *min*	40.1 *min*

We did not prove the theorems for higher values of n because for $n \geq 7$ the simplifier requires more than the 256MB of RAM available on the test machine. But the data in the table suggest that even without the space restriction, the automatic proof tools in Isabelle would not be able to handle larger systems in a reasonable amount of time, and without them the verification effort is prohibitively expensive. In the small example verified in this paper, we split the task of verifying trace inclusion for a nine-component system $PCImpl_2$ into two separate tasks, one of which deals with a six-component system $PCImpl_1$ and the other with a five-component system *KDImpl*. Notably, we could not prove the projection theorems for the nine-component case, but could do so for the smaller component cases. This modest division resulted in substantial savings primarily because complexity, running time, and space requirements appear to be exponentially related to problem size. In the context of real-world systems that can have dozens of such components, abstraction and decomposition become essential.

6.1 Observations on Benefits of Formal Verification

Refinement proofs turned out to be a more effective way of fleshing out specification problems than invariant proofs. Invariant proofs may touch only specific parts of the protocol state and leave untouched more abstract questions about what the protocol is doing. The refinement proof makes explicit all the assumptions about why the implementation does what the specification intended.

In particular, during the refinement proof for the implementation $PCImpl_1$ we were forced to go back and prove several auxiliary invariants whose utility were not obvious *a priori*. This in turn led us to typos and errors in our formalization of the cryptosystems and component automata. Although the bugs caught during the process of proving invariant lemmas and trace

inclusion were mostly errors in our formalization, we caught one error in the original description of $PCImpl_1$ protocol (some uninitialized variables led to failed proofs of the base case) and a typo in an invariant statement in [Lyn99].

6.2 Efficiency Issues

The human effort spent on the project included (1) twelve weeks for formalizing and verifying $PCImpl_1 \preceq PC$, (2) three weeks for verifying $KDImpl \preceq KD$, and (3) three days for verifying $PCImpl_2 \preceq PC$. A substantial fraction of the time in stage 1 was spent learning Isabelle/IOA and setting up the procedure for formalizing I/O automata, stating and proving invariants, and proving trace inclusion. This accounts for most of the difference in effort between stages 1 and 2. Stage 3 was much shorter due to our use of the compositionality theorem.

We believe that additional automation can reduce the human effort substantially in all phases of the verification process. At the level of the prover, additional tactics can automate tasks that commonly show up in reasoning about I/O automata. These tactics fall into two categories. One set of tactics would simulate the high-level proof steps used in human-style I/O automata proofs. These would be similar to the proof strategies offered by Archer's TAME environment for PVS. Another set of tactics would help the user deal with proof obligations specific to Isabelle and the Isabelle formalization of I/O automata meta-theory. For example, applying the compositionality theorem requires proofs for side conditions that the Isabelle type checker does not guarantee. It must be shown that the I/O automata definitions are well formed—the sets of input, output, and internal actions are disjoint, and the transition relation is defined only for the actions in the automaton signature definition. Furthermore, the user must show that the automata being composed are compatible with each other. These proofs have common structure and can therefore be effectively encapsulated in a higher-level Isabelle tactic. The tactic would be used with every application of the compositionality theorem.

There are also ways to improve efficiency at the user interface level. A compiler can take care of translating I/O automata (expressed in a suitable way) into an Isabelle formalization. It is also possible to generate a general framework for invariant definitions and trace inclusion proofs automatically, letting the user fill in definitions and proof script details specific to the problem.

One of the biggest obstacles to formal reasoning with theorem provers remains their cumbersome nature and the level of attention to low-level details required of the user. Isabelle is not an exception. Interacting with the bare-bones prover throughout the verification cycle can be a frustrating experience, which is why we emphasize the need to automate as much of the process as possible. With the enchancements discussed above, the task of formalizing the specification and setting up proof goals and induction can be substantially automated. Most user interaction with the prover would take place when reasoning about individual automata actions. The actions typically have a small and localized effect on the automaton state, which makes the proofs more manageable.

6.3 Technical Issues

In Müller's formalization of I/O automata meta-theory the binary automata composition operator has the following type, given in Isabelle's ML-like notation:

$$|| \;::\; (\alpha, \sigma)\ ioa \rightarrow (\alpha, \tau)\ ioa \rightarrow (\alpha, \sigma \times \tau)\ ioa$$

where α is the action type, σ and τ are state types of the automata being composed, and $\sigma \times \tau$ is the state type of the composition. The composition operator requires that both automata be defined over the same action space α. If we apply the operator multiple times to compose several automata, every action of every component must be a member of the same action space. Mechanized induction on the action datatype generates a subcase for each action in the action space, including those that do not belong to the component being verified. This means that inductive proofs do not scale well for large compositions of automata. This is a serious problem, as it undermines the primary benefit of compositional reasoning: scalability. It takes over an hour for Isabelle (ver. 99) to execute in interactive mode the invariant and refinement proof scripts developed in this project. The simplifier spends the majority of that time reducing inductive subcases for actions, considering many more cases than necessary.

Fixing the problem without completely revising the meta-theory requires a richer type system than supported by Isabelle/HOL. For example, in a polymorphic language with subtyping and union types [Pie91], the composition operator could be given the following type:

$$|| \;::\; (\alpha, \sigma)\ ioa \rightarrow (\beta, \tau)\ ioa \rightarrow (\alpha \vee \beta, \sigma \times \tau)\ ioa$$

The action type of the composition $\alpha \vee \beta$ is the union type derived from the action types α and β of the components. Assuming that the usual binary operators on sets (union, intersection, difference) have the type $\alpha\ set \rightarrow \beta\ set \rightarrow (\alpha \vee \beta)\ set$, the existing definition of the composition operator would still make sense in this setting.

7 Conclusions

Existing compositional proof methods, including implementation relations between I/O Automata, are adequate for handling large classes of verification problems. Numerous case studies have used these techniques by hand to prove global properties of non-trivial systems. Until recently, automated verification tools have not included compositional techniques in their repertoire. Yet, the strengths of compositional reasoning and automated reasoning have the potential to complement each other.

Automation demands that compositional proofs be made strictly rigorous. It does not tolerate typos or imprecise wording, which can lead to subtle errors in hand proofs. Forced to develop proofs according to these exacting standards, the user gains deeper understanding of the subtleties of the system and more confidence in the final product. Although time consuming to use, automated proof tools make proof re-checking much easier, which can result in substantial time savings in the iterative development/verification cycle. Conversely, compositional techniques offer the best hope of dealing with state explosion and complexity problems associated with automated verification of non-trivial systems.

Our experience with the Isabelle/IOA verification environment leads us to conclude that there is a lot of work yet to be done before the potential benefits of automated compositional reasoning are fully realized. Using Isabelle/IOA is a labor-intensive undertaking, and the environment does not appear to be sufficiently scalable. These issues can be resolved with additional effort, and we believe that the benefits of the compositional approach make the effort worthwhile.

References

[AHS98] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. *UITP '98*, July 1998.

[GSSL93] R. Gawlick, R. Segala, J.F. Sogaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, MIT, Laboratory for Computer Science, Cambridge, MA., December 1993.

[LT89] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[Lyn96] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, March 1996.

[Lyn99] N. Lynch. I/O automaton models and proofs for shared-key communication systems. *12th IEEE Computer Security Foundations Workshop (CSFW12)*, pages 14–29, June 1999.

[Mül98] O. Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Technische Universitäat München, 1998.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Pie91] B.C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[SW00] O. Sheyner and J. Wing. Toward compositional analysis of security protocols using theorem proving. Technical Report CMU-CS-00-106, CMU Computer Science Department, Pittsburgh, PA., January 2000.

Integrating SVC and HOL with the Prosper Toolkit*

Alan Stevenson[1] and Louise A. Dennis[2] **

[1] Department of Computing Science, University of Glasgow, G12 8QQ, UK
[2] Division of Informatics, University of Edinburgh, EH1 1HN, UK louised@dai.ed.ac.uk

Abstract. We describe an integration of the SVC decision procedure with the HOL theorem prover. This integration was achieved using the PROSPER toolkit. The SVC decision procedure operates on rational numbers, an axiomatic theory for which was provided in HOL. The decision procedure also returns counterexamples and a framework has been devised for handling counterexamples in a HOL setting.

1 Introduction

The PROSPER project[1] is researching and developing a toolkit [4] that allows an expert to easily and flexibly assemble *proof engines* from existing tools to provide embedded formal reasoning support inside applications. The ultimate goal is to make the reasoning and proof support invisible to the end-user—or at least, more realistically, to incorporate it securely within the interface and style of interaction to which they are already accustomed. One of the main facets of the project is the use of plugin technology in order to prevent the continual re-implementation of existing techniques.

This paper describes the construction of a PROSPER plugin based upon the Stanford Validity Checker (SVC)[14]. This work has included providing an axiomatic theory for rationals in HOL and a framework for handling counter-examples.

In §2 and §3 we give overviews of the PROSPER toolkit and SVC respectively. In §4 we discuss our integration of the two systems including an overview of the theory of rational numbers and the framework for counterexamples. In §5 we describe the results of experimental evaluation of the system. In §6 we look at related work and in §7 we discuss further work needed on the plugin.

2 The Prosper Toolkit

A central part of PROSPER's vision is the idea of a proof engine—a custom built verification engine which can be operated by another program through an Application Programming Interface (API). A proof engine can be built by a system developer using the toolkit provided by the project. A proof engine is based upon the functionality of a theorem prover with additional capabilities provided by 'plugins' formed from existing, off-the-shelf, tools. The toolkit includes a set of libraries based on a language-independent specification for communication between components of a final system. The theorem prover's command language is treated as a kind of scripting or glue language for managing plugin components and orchestrating the proofs.

The central component is based on a theorem prover because this comes with ready made concepts of term, theorem, and goal, which are important for managing verifications. A side benefit is that all the functionality in the theorem prover (libraries of procedures, tactics, logical theories, etc.) becomes available to a developer for inclusion in their custom proof engine. This does not prevent theorem proving being very lightweight if desired.

* The authors would like to thank Tom Melham for his help with the research reported here.
** Corresponding Author
[1] http://www.dcs.gla.ac.uk/prosper

A toolkit has been implemented based around HOL98, a modern descendent of the HOL theorem prover [5]. The toolkit provides several plugin components based on external tools which offer APIs to a proof engine. It also provides support to enable developers of other verification tools to offer them as PROSPER plugins.

The application, proof engine and plugins act as separate components in the final system (Figure 1). In the first prototype they are also separate processes. Communication between them is treated in a

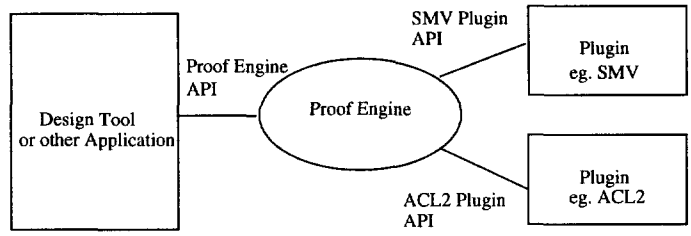

Fig. 1. A system built with the PROSPER toolkit

uniform manner specified by the PROSPER *Integration Interface.*

2.1 The Prosper Integration Interface

A major part of the PROSPER methodology is the PROSPER Integration Interface (PII), a language-independent specification of communication for verification. This specification is currently implemented in several languages (C, C++, ML, Java, Ada, λProlog and Python) allowing components written in these languages to be used together.

The PII consists of several parts. The first is a datatype, called *interface data*, for all data transferred between an application and a proof engine and between a proof engine and its plugins. A major part of the datatype is the language of higher order logic used by HOL and so any formula expressible in higher order logic can be passed between components. Many plugins operate with logical data that is either already a subset of higher order logic (e.g. predicate calculus and propositional logic) or embeddable in it (e.g. CTL). The second part consists of a datatype for the results of remote function calls and support for installing and calling procedures in an API. There are also parts for managing low level communication, which are largely invisible to an application developer.

The developer of a verification tool can adapt it so that it can be used as a PROSPER plugin. A plugin developer programs both in ML and in the plugin's own implementation language. The developer will place chosen entrypoints to the plugin into an API database. In the plugin's implementation language they will translate any arguments needed by these functions into interface data. In the theorem prover's command language they will need to unpackage these entrypoints again so they present themselves as language-specific bindings in that language (ML). In particular any additional theories required (i.e. an embedding of the logic used by the plugin into higher order logic) should be provided by the plugin developer.

3 SVC

SVC stands for the Stanford Validity Checker. This tool has been in development at Stanford for several years, and is used both in research, and as a formal hardware verification tool, as seen in such papers as [8, 1]. SVC allows users to check formulae based on a subset of first order logic and boasts of its efficient and automatic decision procedures.

The logic for these decision procedures includes booleans, uninterpreted functions and linear arithmetic. There are also interpreted functions such as array operations, bit vectors and parts of linear arithmetic.

SVC provides a variety of commands to the user, from a traditional command-line user interface. At its simplest level, proof is carried out through use of the `check_valid` command, which is given an expression in SVC logic, and will return an answer based on the validity of that expression, and possibly a counter-example, if the expression is found to be invalid.

SVC also provides numerous other commands which aid in proof. These can relate to various command flags, which affect the way SVC works, or manipulate SVC's *context* which stores proven results.

4 Integrating SVC and HOL

We attempted to integrate SVC and HOL following, as closely as possible, the methodology set down by [4]. This methodology outlined the following set of tasks:

1. Provide an implementation of the PII in the implementation language of SVC.
2. Identify entry points into SVC to make accessible to HOL.
3. Provide a translation between interface data and the formula syntax of SVC.
4. Provide ML language bindings in HOL to make the operation of SVC appear as natural as possible to a HOL user.

We have provided a link up based on this methodology [2].

4.1 Implementing the PII

SVC is written in C++. An implementation of the PII already existed in C and it was a relatively trivial exercise to adapt this to C++, in fact much of the relevant code was simply imported directly into C++ without any change or alteration at all. All the work required was standard reworking of C code to allow it to be called from C++.

4.2 Entry Points into SVC

SVC has a number of access points through its command line interface. Ideally we would have liked to have provided most of these to HOL users. However, given time constraints, we chose to provide only `check_valid` and `reset`. `check_valid` is SVC's primary command for proof, to which the user supplies an expression, and the system will return a result based on the validity of that expression. This can be either VALID, INVALID or INVALID with a counter-example. `check_valid` will do this by setting up a context for the current proof, in which the actual validation takes place, and the results will be stored, which can then be used in further proofs.

`reset` is the command used to clear the current context, when it is felt that previous results will no longer be needed, allowing the user to start with a "clean" SVC session.

These entry points were placed in a PROSPER API for SVC.

4.3 Translation between interface data and SVC's formula syntax

SVC has a large formula syntax for formulas expressed in arithmetic, records and bit-vectors. We chose for a first implementation only to cater for a subset of this language which corresponded to the arithmetical expressions.

[2] `http://www.dcs.gla.ac.uk/~stevenat/svc_plugin.tar.gz`

The translation from interface data into formula syntax thus depended upon the representation of rational numbers in interface data. To determine this we needed to know how the rationals were represented in HOL since that would determine their presentation in interface data (the translation between HOL terms and interface data is already decided and implemented and acts as the gold standard for other translations). Our theory of rational numbers is described below.

The translation on the C++ side of the interface is performed simply, by taking a PII term, and identifying the kind of term we have (equality, variable, application etc). The PII provides similar methods to HOL for the decomposition of terms, and using these functions we extract the necessary information from the term (which can be identified further by checking constituent parts for known function names, such as those defined in the rational number theory) such as function arguments, variable names and numerical constants. With this information, the appropriate SVC expression is then built up, using constructors provided by SVC. For example, specific objects are provided for numerical expressions, function expressions and addition expressions. These all stem from a generic expression object within SVC.

The Theory of Rationals SVC assumes that all numbers appearing in arithmetical expressions are rational numbers. There is no full theory of rationals in HOL although John Harrison provided a theory of *half-rationals* to support his construction of a theory of real numbers [15]. It is preferable to provide *definitional* theories for HOL (i.e. constructing a new theory from an old one by definitions and then proving the axioms of the new theory). However it is possible and in many cases simpler to provide an *axiomatic* theory which we chose to do in this case. Our theory of rationals is based upon the description in *Axiomatic Set Theory* [16].

The theory provides a new HOL type, *rat*, with which to represent rational numbers. It provides the usual constants representing addition, multiplication and inequalities, and also defines the axioms for the rational numbers. The main problems came with the representation of numbers with this rational theory. This is solved by providing three functions:

1. `rat` - takes in something of type `num` (a HOL type for natural numbers) and returns a `rat`. This allows us to parse numerical constants as rational numbers.
2. `rneg` - takes in something of type `rat` and returns its negation. This function allowed us to define negative rational numbers from within HOL.
3. `rdiv` - Taking in two numbers of type `rat` this constant is defined to allow us to represent division of rational numbers. This is especially important as it allows us to define fractions.

By defining the axioms and functions in this way, it is hoped that this will allow for possible expansion of the theory, by implementing the functions, and by properly proving the axioms.

With this in place we were able to handle a reasonable subset of SVC's input language. We have a representation within HOL with which users can specify boolean and rational number predicates, including most of the usual logical operators (AND, OR, NOT) as well as rational functions (negation, addition, multiplication and inequalities) and fractional numbers. These statements allow us to represent a large part of SVC syntax, although at current standing, bit-vectors and records cannot be represented within HOL, due to the lack of a suitable HOL theory, although there is nothing to suggest that this could not change in future years, allowing the expansion of our theory to encompass more, if not all, of the SVC syntax.

4.4 ML language bindings for the SVC API

Recall that we had placed two SVC functions in its PROSPER API. We now needed to present these in some way to HOL users. We implemented two HOL entry points based upon `check_valid`. The first (also called `check_valid`) is of type `Term -> Term`. The user supplies a term which represents a boolean statement. This statement can be composed of boolean and rational expressions as shown above. This

term will then be sent to the plugin using the PII where an SVC context is created, and validity checks performed. The user will then get another term as a result (assuming an error has not ocurred) which will be one of `True`, `False` or a term representing the counter-example calculated by SVC (This counter-example is naturally presented in the same syntax as the initial term provided by the user).

Check valid provides an SVC centric entrypoint which is not of a great deal of use for proof in HOL unless the formula sent to SVC is a subgoal of a proof in its own right. A more HOL centric entrypoint is `svcprove` which will return a HOL theorem which can then be saved and recalled when the result is needed at a later date. This function is essentially a wrapper for `check_valid` which parses the returned result, and constructs the HOL theorem based on whether the result was `True` or `False`. If the result is a counter-example, then the original Term is equated to False, and the counter-example is provided as an assumption to the theorem.

A major part of HOL is the exploitation of conversions. "A conversion is a rule that maps a term to a theorem expressing the equality of that term to some other term" [5]. They are particularly useful for providing rewrite rules to rewriting tactics. `SVCprove` is a conversion which means it can be naturally and easily used in the course of HOL proof, in particular it can be exploited by rewriting techniques.

Counter Examples SVC can return counter examples for non-theorems. Counter examples are not of direct use in HOL proofs however they can be very useful in "debugging" non theorems and in guiding proof. Hence counterexamples can provide useful information to a HOL user or potentially to the user of a design tool which incorporates PROSPER style verification capabilities. We needed to provide a framework in which counterexamples could be returned to a HOL user and possibly to a design tool beyond.

SVC provides counter-examples in the form of formulae representing a set of circumstanes under which the original expression will fail. These counter-examples are stored within the context and are accessed through the context's `Splitter` functions. These functions allow you to access the most recent splitters (counter-examples) and will provide them as SVC expressions. These expressions are then translated into PII terms in a similar way to which PII terms become SVC expressions (in that the specific type of expression is identified, and then the methods SVC provides for decomposing these expressions are employed to retrieve the appropriate information, which is then supplied as arguments to PII methods for constructing terms).

Counter-examples are returned as assumptions to a HOL theorem equating the goal expression to false. This was felt to be a natural way of viewing the available information, as well as supplying it to the user in a way which allows it to be stored for later use (as opposed to simply storing the truth of the original term, and having the counter-example separate).

This is a generic approach. As well as the advantages stated above (the ability to store counter-example and theorem as one statement for future reference, as well as human readability – `if` *counter-example* `then` *Term* `equals False`) HOL provides simple functions for the decomposition of these expressions, making it easy to extract the counter-example or leave it in as necessary. If the counter-example and theorem were separate there could be difficulties combining them later on (e.g. if the counter-example is kept in a type that cannot be easily stored for future use).

5 Evaluation

Ideally we would have liked to have tested the efficiency of the SVC plugin against an internal HOL decision procedure. However, no appropriate decision procedures exist for rational numbers. As a result we can only note that it is now possible to work with rationals in HOL which it wasn't previously, and that with relatively little work we were able to give access to a powerful decision procedure in that domain without having to implement it from scratch. In this way we have provided a new theory in HOL with sophisticated theorem proving support with relatively little effort.

Below are a variety of test examples, which show the usage of the rationals theory, as well as the SVC commands we have provided.

5.1 A Simple Example

This is a simple example showing a trivial use of `check_valid`.

```
- check_valid (Term '(x = rat 6)');
> val it = '(x = rat 6) = T' : Term.term
```

5.2 Addition and Multiplication

This next example uses `svcprove` to decide an equality expression involing addition and multiplication.

```
- svcprove (Term '((rat 3) rtimes m) rplus ((rat 4) rtimes n) = n rplus n rplus n
rplus n rplus (m rtimes (rat 3))');
> val it =
[]
|- (rat 3 rtimes m rplus rat 4 rtimes n =  n rplus n rplus n rplus n rplus m rtimes
rat 3) =  T
: Thm.thm
```

5.3 Inequalities and negation

This example shows a proof involving inequalities, as well as the `rneg` function:

```
- svcprove(Term 'rneg (rat 4) rless rat 3');
> val it = [] |- rneg (rat 4) rless rat 3 = T : Thm.thm
```

5.4 Division by Zero

This final example shows the result of trying to divide by zero:

```
- check_valid(Term 'rdiv (rat 1) (rat 0) = x');
! Uncaught exception:
! Fail "Term is not an SVC term"
```

6 Related Work

Several decision procedures have been converted into PROSPER plugins, most notably the SMV model checker [9], the first order logic reasoner Gandalf [13, 7], the ACL2[2] theorem prover and Prover technology's proof tool based on Stälmarck's decision procedure [11, 10]. The Prover plugin also deals with the return of counter examples. It presents a countermodel, a list of pairs of expressions and their truth values, as a third result of the decision procedure (along with true and false). This presentation is rather ad hoc and doesn't present the counter model together with the original goal as we do, nor does it allow the results of the decision procedure to be used in conversions. It wouldn't be difficult to wrap up the Prover plugin's output to match our representation. The approach we have adopted here integrates the plugin more seamlessly within HOL and is general enough to cover the Prover plugin's output.

SVC has been integrated into Isabelle as part of Isabelle/DC, a proof assistant for the real-time logic Duration Calculus[6]. However, as a result of using PROSPER methodologies to construct the SVC plugin, it now shares a consistent programming interface with other PROSPER plugins, which will lead to easier future development.

7 Future Work

To properly assess and evaluate the SVC plugin our axiomatic theory of rationals needs to be converted into a definitional one. One route to pursue here might be to treat the rational numbers as a restriction of the reals.

We would also like to expose more of SVC's entrypoints to HOL. We do not at present, have any plans to extend the formula syntax handled since HOL has no appropriate theories for records or bit vectors. However should these become available then it would be interesting to evaluate the use of the decision procedure in conjunction with them.

8 Conclusion

We have integrated SVC into HOL using the technology of the PROSPER toolkit. This integration has succeeded with encouraging results and serves as an example of how decision procedures can be integrated into HOL using the PROSPER approach without the necessity of re-implementation.

One of the problems we encountered was a mismatch between the objects handled by SVC (rationals, records and bit vectors) and the theories in HOL. While we have provided an axiomatic theory for rational numbers in HOL it seems likely that if the PROSPER project wishes to encourage more integration of this kind then a wider set of theories will need to be implemented since it would be unrealistic to expect tool vendors to implement their own theories for standard types such as rational numbers.

We have developed a framework within which counterexamples should be presented to HOL. We intend that this framework should be general and provide a standard format for accessing counterexamples. In this way design tools wishing to exploit counter-examples (to provide debugging information to a user, for instance) need only know that a plugin can return counter-examples and need not worry about the manner in which the counter-example is presented.

References

1. C. Barrett, D. Dill and J. Levitt, Validity Checking for Combinations of Theories with Equality, M. Srivas and A. Camilleri (eds), *Formal Methods In Computer-Aided Design (FMCAD'96)*, Lecture Notes in Computer Science 1166, Springer-Verlag, pp 187–201, 1996.
2. B. Brock, M. Kaufmann, and J Moore, ACL2 Theorems about Commercial Microprocessors. M. Srivas and A. Camilleri (eds), *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*, Lecture Notes in Computer Science 1166, Springer-Verlag, pp. 275–293, 1996.
3. A. Church, A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, vol. 5, pp. 56–68, 1940.
4. L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon and T. Melham, The PROSPER Toolkit, S. Graf and M. Schwartzbach (eds). *TACAS 2000*, Lecture Notes in Computer Science 1785, pp. 78–92, 2000.
5. M. J. C. Gordon and T. F. Melham (eds), *Introduction to HOL: A theorem proving environment for higher order logic*, Cambridge University Press, 1993.
6. S. T. Heilmann, *Proof Support for Duration Calculus*, Department of Information Technology, Technical University of Denmark, PhD-thesis, January 1999
7. J. Hurd, Integrating Gandalf and HOL. Y. Bertot, G. Dowek, A. Hirshowitz, C. Paulin and L. Théry (eds). *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 1690, Springer-Verlag, pp. 311–321, 1999.
8. J. R. Levitt, Formal Verification Techniques for Digital Systems. *PHD Thesis*, Stanford University, December, 1998.
9. K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers. 1993.
10. M. Sheeran and G. Stålmarck, A tutorial on Stålmarck's proof procedure for propositional logic. *The Second International Conference on Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science 1522, Springer-Verlag, pp. 82–99, 1998.

11. G. Stålmarck and M. Säflund, Modelling and Verifying Systems and Software in Propositional Logic. *Proceedings of SAFECOMP '90*, Pergamon Press, pp. 31–36, 1990.
12. M. Staples, Linking ACL2 and HOL, Technical Report No. 476, University of Cambridge Computer Laboratory. 1999.
13. T. Tammet, A resolution theorem prover for intuitionistic logic. *13th International Conference on Automated Deduction*, Lecture Notes in Computer Science 1104, Springer-Verlag, pp. 2–16, 1996.
14. SVC Home Page, *The Stanford Validity Checker*, http://verify.stanford.edu/SVC.
15. John Harrison, *Constructing the real numbers in HOL*, University of Cambridge.
16. Patrick Suppes, *Axiomatic Set Theory*, D. Van Nostrand Company, Inc, 1960.

The De Bruijn Factor

Freek Wiedijk
<freek@cs.kun.nl>

Department of Computer Science
Nijmegen University
P.O. Box 9010
6500 GL Nijmegen
The Netherlands

Abstract. We study de Bruijn's 'loss factor' between the size of an ordinary mathematical exposition and its full formal translation inside a computer. This factor is determined by a combination of the amount of detail present in the original text and the expressivity of the system used to do the formalization. For three specific examples this factor turns out to be approximately equal to *four*.

1 Loss Factor

In 'A survey of the project Automath' de Bruijn wrote (p. 160 in section A.5 of [9] which is a reprint from [1]):

> *A very important thing that can be concluded from all writing experiments is the* constancy of the loss factor. *The loss factor expresses what we loose in shortness when translating very meticulous 'ordinary' mathematics into Automath. This factor may be quite big, something like 10 or 20, but it is constant: it does not increase if we go further in the book. It would not be too hard to push the constant factor down by efficient abbreviations.*

Here* we briefly study this loss factor, which we call the *de Bruijn factor.*

When writing a 'formal proof' (a proof that is entered in a computer in full detail in such a way that the computer can check the correctness) there are basically two approaches:

- One takes an existing, non-formal, mathematical text and translates it – more or less faithfully – into a computer representation.
- One 'programs' the proof directly into the formal system, without first creating a 'natural language' counterpart.

The first method has the advantage that the formalization automatically will be well documented, and also it generally seems to be easier to translate a pre-existing text than to think about the proof and about the mechanics of the formalization at the same time. The de Bruijn factor of course only can be objectively measured for a formalization of the first kind.

The de Bruijn factor of a formalization depends on two aspects. On the one hand there is the level of detail in the original, which depends on the 'character' of the text that is being translated. There exist a wide range of mathematical styles, which each have their own level of precision at which the proofs are elaborated. For instance there are:

- Books that give a detailed development of a subject for foundational purposes, like Whitehead and Russell's *Principia Mathematica* [10].

* The full Automath, Mizar and TEX files that are discussed here can be found on the World Wide Web at the address <http://www.cs.kun.nl/~freek/factor/>.

- Ancient mathematics, like Euclid's *Elements* [4].
- Textbooks for education.
- Handbooks about specific mathematical subjects.
- Papers in computer science that have a strong mathematical flavor.
- Mathematical research papers.

The three cases studied in this note are respectively of the first, fifth and fourth kinds.

On the other hand there is the system that is being used. Some systems have more automation and more (as de Bruijn called it) 'efficient abbreviations' than others. So the de Bruijn factor measures how efficient a system is. One might imagine a 'benchmark' for proof assistants consisting of a number of mathematical texts in various styles to be represented. However, the technology currently seems not to be well-developed enough to already put together such a benchmark.

2 Apparent and Intrinsic De Bruijn Factors

The size in bytes of the files of a formalization is not a very meaningful measure. It depends on such matters as the choice of variable names and the use of whitespace: these factors don't seem to mean much for the contents of the files. For instance if indentation is done with tab characters that part of the file will be eight times as small as when it's done with space characters: but the file will look the same in both cases. As another example, the TEX macro name '`\Leftrightarrow`' for the '$\Leftrightarrow$' symbol uses 15 characters, while an encoding like '`<=>`' uses only 3.

To compensate for these effects it seems natural to 'squeeze out' trivial redundancy by compressing the files before calculating the ratios of their sizes. In fact, use of the tab character can be seen as a crude way of compressing long runs of spaces.

We will call the ratio of the uncompressed file sizes the *apparent* de Bruijn factor, and that of the compressed file sizes the *intrinsic* de Bruijn factor.

If one uses the intrinsic de Bruijn factor, it isn't useful any more to remove the white space from the computer representation of a proof to get a better ratio, because this kind of optimization only has a minor effect on the size of the compressed file.

Surprisingly it turns out that generally both the 'natural language' and the 'computer' versions of a proof compress similarly well. This means that the apparent and intrinsic de Bruijn factors turn out to be approximately the same.

3 Arithmetic in Automath

The first example for which we will calculate the de Bruijn factor is Jutting's classic Automath translation (see section D.2 of [9], a reprint from [6]) of Landau's *Grundlagen der Analysis* [7], a cute little book about the basic laws of arithmetic up to the complex numbers.

To give an impression of the text and its translation, here is a small fragment of a proof (of 'Satz 27' on p. 37 of [7]) in the *Grundlagen*:

> *1 gehört zu $\mathfrak{M}$ nach Satz 24. Nicht jedes x gehört zu $\mathfrak{M}$; denn für jedes y aus $\mathfrak{N}$ gehört $y+1$ nicht zu $\mathfrak{M}$, wegen*
>
> $$y + 1 > y.$$

together with its rendering in the AUT-QE dialect of the Automath language:

```
s@[n:nat]
t1:=[x:<n>p]satz24a(n):lbprop(1,n)
s@t2:=[x:nat]t1(x):lb(1)
[l:[x:nat]lb(x)][y:nat][yp:<y>p]
```

```
t3:=satz18(y,1):more(pl(y,1),y)
t4:=satz10g(pl(y,1),y,t3):not(lessis(pl(y,1),y))
t5:=th4"l.imp"(<y>p,lessis(pl(y,1),y),yp,t4):not(lbprop(pl(y,1),y))
t6:=th1"l.all"(nat,[x:nat]lbprop(pl(y,1),x),y,t5):not(lb(pl(y,1)))
t7:=mp(lb(pl(y,1)),con,<pl(y,1)>l,t6):con
l@t8:=someapp(nat,p,s,con,[x:nat][y:<x>p]t7(x,y)):con
```

Note the reference to 'Satz 24' at the start of both versions of the fragment.

The sizes of both the TEX and Automath versions of this book, both uncompressed and compressed (using the Unix `gzip` utility), and the corresponding de Bruijn factors are given in the following table:

	informal	formal	*de Bruijn factor*	
uncompressed	189 K	736 K	apparent	3.9
compressed	42 K	155 K	intrinsic	3.7

Apparently the de Bruijn factor of Automath for this kind of text is slightly less than *four*.

4 Computer Science in Mizar

The second example that we will calculate the de Bruijn factor for, is a section from a paper about 'finite topology' (pp. 12–17 of [8]), really a mathematical development of the theory of digital filtering of one-bit images. The translation [5] is in Mizar, which is a more accessible and more high level system than Automath.

As an example, here's a fragment of the proof of 'Theorem 2.1' (p. 16 of [8]):

> *Let B and C be non-void subsets of A such that $B \cap C = \emptyset$ and $B^b \cap C = \emptyset$. Then, there exists an element x in B, and we can construct a set P_n as a procedure described previously and $P_{n+1} = P_n$.*

with as Mizar translation:

```
given B, C being Subset of the carrier of FT such that
  A26:A = B U C and
  A27:B <> ∅ and
  A28:C <> ∅ and
  A29:B ∩ C = ∅ and
  A30:B^b ∩ C = ∅;
A31: B c= B^b by Th18 ;
A32: B^b ∩ A = B^b ∩ B U ∅ by A26,A30, BOOLE:70
           .= B^b ∩ B by BOOLE:60
           .= B by BOOLE:42, A31 ;
consider x being Element of B ;
x ∈ A by A26, BOOLE:def 2,A27 ;
then consider S being FinSequence of bool the carrier of FT such that
  A33:len S > 0 and
  A34:π(S,1) = {x} and
  A35:for i being Nat st i > 0 & i < len S holds π(S,i+1) = π(S,i)^b ∩ A and
  A36:A c= π(S,len S) by  A25 ;
```

Note the close syntactical similarity of the requirements on B and C in both versions of the text.

Here are the statistics of this example:

	informal	formal	*de Bruijn factor*	
uncompressed	7.7 K	35.3 K	apparent	4.6
compressed	2.6 K	8.0 K	intrinsic	3.1

So the de Bruijn factor of this text is not much better than that of the previous one. An explanation might be that the paper that is translated contains much less detail than the *Grundlagen* book, so the extra power of Mizar is compensated for by the more loose style of the informal article. For instance, a number of statements at the end are not proved, but instead it is just stated that:

The following facts are easily derived.

5 Mathematics in Mizar

The final example for which we calculate the de Bruijn factor, is part of an ongoing effort at the Mizar project to translate a complete mathematical book [3] into the Mizar language. For the book a handbook from mathematical logic was chosen, which presents the theory of 'continuous lattices.' The translation of this book (which currently is halfway finished) consists of a large number of Mizar articles with names starting with 'YELLOW' and 'WAYBEL'.

The article that we analyze here [2] is the translation of four pages of the book. Again we give an example of the style of both the original and the translation. The statement of 'Corollary 1.13' (on p. 106 of [3]) is:

If L is a continuous lattice, then $(L, \sigma(L))$ is a quasicompact and locally quasicompact sober space. In particular, $(L, \sigma(L))$ is a Baire space.

which gets translated into Mizar as:

```
L is continuous implies L is compact locally-compact sober Baire
```

Then the proof of this 'Corollary' starts with the following reasoning:

We have to show that a point $x \in L$ has a basis of quasicompact neighborhoods. By 1.10 the sets $\uparrow y$ with $y \ll x$ form a basis for the neighborhoods of the point. But as we know, if $x \in U \in \sigma(L)$, then actually we have a $y \in U$ with $y \ll x$; hence, $\uparrow y \subseteq U$, and so the sets $\uparrow y$ can be used as neighborhoods.

to which corresponds the following fragment of the Mizar proof:

```
 thus A5: L is locally-compact
 proof let x be Point of L, X be Subset of L such that
A6: x ∈ X and
A7: X is open;
   reconsider x' = x as Element of L by STRUCT_0:def 2;
   set bas = { wayabove q where q is Element of L: q << x' };
A8: bas is basis of x by A1, WAYBEL11:44;
   consider y being Element of L such that
A9: y << x' & y ∈ X by A1, A6, A7, WAYBEL11:43;
   X is upper by A7, WAYBEL11:def 4; then
A10: uparrow y c= X by A9, WAYBEL11:42;
   set Y = uparrow y;
 take Y;
```

```
    wayabove y ∈ bas by A9; then
A11: wayabove y is open & x ∈ wayabove y by A8, YELLOW_8:21;
    wayabove y c= Y by WAYBEL_3:11; then
    wayabove y c= Int Y by A11, TOPS_1:56;
  hence x ∈ Int Y by A11;
  thus Y c= X by A10;
```

Because this book is 'more mathematical' and hence reasons at a higher level than the previous two examples, the de Bruijn factor is a bit higher:

	informal	formal	*de Bruijn factor*	
uncompressed	11.7 K	78.4 K	apparent	6.7
compressed	4.0 K	16.3 K	intrinsic	4.1

6 The De Bruijn Threshold

It seems plausible that there is a certain value for the de Bruijn factor such that, when proof checkers become sufficiently powerful that their factor drops below it, people will start using them for serious work (like verifying the correctness of their mathematics and communicating the precise details of their work to others). We suggest to call this value the *de Bruijn threshold.* Like the de Bruijn factor of a system, it probably depends on the kind of mathematics.

As it probably is not possible to get a formalization to be as short as its informal version, it is to be hoped that the de Bruijn threshold of something interesting won't be less than *one.*

References

[1] N.G. de Bruijn. A survey of the project Automath. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, New York, London, 1980.

[2] C. Bylinski and P. Rudnicki. The Scott topology, Part II. *Journal of Formalized Mathematics*, 9, 1997. MML Identifier: **WAYBEL14**.

[3] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *A Compendium of Continuous Lattices.* Springer-Verlag, Berlin, Heidelberg, New York, 1980.

[4] Sir Th.L. Heath. *The Thirteen Books of Euclid's Elements.* Dover Publications, New York, dover edition, 1956. First edition 1908, second edition 1925.

[5] H. Imura and M. Eguchi. Finite Topological Spaces. *Journal of Formalized Mathematics*, 4, 1992. MML Identifier: **FIN_TOPO**.

[6] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the Automath system.* Number 83 in Mathematical Centre Tracts. Mathematisch Centrum, Amsterdam, 1979.

[7] E. Landau. *Grundlagen der Analysis.* Chelsea Publishing Company, New York, fourth edition, 1965. First edition 1930.

[8] Y. Nakamura, Y. Fuwa, and H. Imura. A Theory of Finite Topology and Image Processing. *Journal of the Faculty of Engineering, Shinshu University*, 69:11–24, 1991.

[9] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics.* Elsevier Science, Amsterdam, etc., 1994.

[10] A.N. Whitehead and B. Russell. *Principia Mathematica.* Cambridge University Press, Cambridge, paperback edition, 1962. First edition 1910, second edition 1927.

Correct Code-Generation in a Generic Framework

Burkhart Wolff[1] and Thomas Meyer[2]

[1] Institut für Informatik, Albert-Ludwigs-Universität Freiburg
wolff@informatik.uni-freiburg.de
[2] Bremen Institute of Safe Systems (BISS), FB 3, Universität Bremen
tm@informatik.uni-bremen.de

Abstract. One major motivation for theorem provers is the development of verified programs. In particular, synthesis or transformational development techniques aim at a formalised conversion of the original specification to a final formula meeting some notion of executability. We present a framework to describe such notions, a method to formally investigate them and instantiate it for three executable languages, based on three different forms of recursion (two denotational and one based on well-founded recursion) and develop their theory in Isabelle/HOL. These theories serve as a semantic interface for a generic code-generator which is set up for each program notion with an individual code-scheme for SML.

1 Introduction

This paper is concerned with the combination of specification notations and program notions, or more precisely, data-view oriented specification formalisms and functional programming languages. While many specification formalisms (such as B, KIV/VSE, but also COQ [3,5,2]) come with a built-in notion of program or executability, formalisms like CASL, Z or HOL[1,22,16] do not. For them, there is a need to fill the gap to programs and their verification — in fact, our original motivation for this work has its roots in our interest in generic formal transformational development and its implementation in the TAS-system [14].

Still, the question arises, why not simply use a specification formalism with a built-in program notion? We see three major reasons for this: First, there are several choices in setting up a program notion — why should a general purpose language like HOL impose just one? Second, with a flexible technique of integrating *arbitrary* program notions, one can choose one very close to a particular programming language paving the way for the verification of language-specific optimisations. Third, we are concerned with the correctness of the compilation; in the past, many code-generators of tools with built-in program notions turned out error-prone, indicating that the task has been underestimated.

Thus, we are interested in a *method* to establish and formally investigate concrete program notions, together with their compilation to code. Our program notions will have the property that the question "is it a program?" – often deliberately left to a non-formalised meta-level – is decidable. Since code-generators in formal development environments serve as a bridge between a theorem prover and the outside world, it is not completely possible to *verify* a code-generator inside a logic; the proof must be extended by meta-logical arguments, which are preferably as *minimal* and *simple* as possible. Finally, we are interested in a technical *framework* supporting the method, providing a prototypical evaluator inside the logic and a code-generator for external code that is intended to produce the same result as the evaluator. The technique should support datatype and higher-order function declarations.

We provide a formal and (in its crucial parts) formally provable method for defining and investigating *program notions* and such a technical framework for *correct code-generation* inside HOL. The choice for HOL – being largely equivalent to Z (see [20] for details) – is motivated by its greater flexibility to embed other formal methods, such that there is a greater potential for reuse of this work for these embeddings. In this paper, we will exemplify this framework for three different program notions and its

code-generation for languages (and compilation schemes) with increasing semantic complexity. The first language is the language of well-founded recursion *Wf* without exceptions; this language is particularly interesting since many library definitions in various HOL-systems can thus be converted into code yielding implementations for end systems or animations of specifications. The second language *Fix* is a call-by-name functional programming language with (one) exception – here, the subtle issues between lazy and eager evaluation have to be covered during the code-generation. Third, we use the relational language *Lfp*; this language is designed as an executable fragment of the specification language Z (see [22], [12]).

On the technical side, our framework is supported by a generic implementation in form of an SML functor that allows the reuse of common functionality in code-generators. As target language, we chose SML. The motivation for this choice is both conceptual and pragmatic. Conceptually, SML has the advantage of a formally defined semantics [15] that can be used (although in a massaged form) as a basis for correctness proofs. Pragmatically it is most compelling to have an SML code-generator since most HOL implementations are based on SML. In particular, we picked Isabelle/HOL as our implementation basis.

2 The Conceptual Framework for Correct Code-Generation

2.1 Fundamentals

The following diagram may illustrate the basic concepts of our work in more detail. Here, with *logic* we denote the set of terms of our programming or specification language – in our case, this will be higher-order logic (HOL) including some conservative extensions such as library theories or specification language encodings. A subset of these terms will be *abstract programs* which again will be a superset of the *abstract values*.

The logic is mirrored by the corresponding sets of (concrete) *programming language* terms and its subset of (concrete) *values*. Both worlds are connected by the code-generation function *convert* that is required to be total on the domain of abstract programs and has the term set *code* as range.

The two relations $\rightarrow_A$ and $\rightarrow_C$ represent the operational semantics of the two languages; we require that they represent partial functions from programs to values. We define a *program notion* as a configuration consisting of a concrete set of abstract program terms and abstract value terms, a set of target programs and concrete value terms, the two fixed evaluation relations $\rightarrow_A$ and $\rightarrow_C$ and the function *convert*. If all components of one program notion include all components of another, we will say that the former has a greater *coverage* than the latter (we owe this terminology to [6]). We will call the code-generation for a program notion *correct* if the above diagram commutes, i.e. iff $convert((\rightarrow_A)t) = (\rightarrow_C)(convert\ t)$. Note that both $\rightarrow_A$ and $\rightarrow_C$ should be undefined for the same t. Provided we have a correct program notion, we can thus convert an abstract program into code and compute the value of program terms outside of our theorem prover, which can be significantly more efficient and should be just the final product of a formal program development we are interested in.

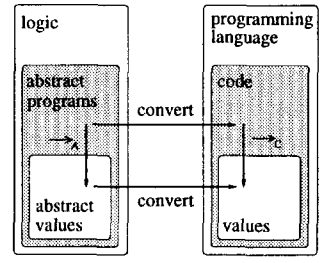

Fig. 1. Basic Concepts

2.2 Correct Code-Generation: A Critique

There are some fundamental problems with the provability of correctness in the sense of the previous section. Since code-generators serve as a bridge between a theorem prover and the outside world, it is not completely possible to *verify* a code-generator inside a logic; the proof must be based on meta-logical arguments. However, we will show how the first principle of LCF-style theorem prover design, namely the enclosure of extra-logical machinery in a *minimal* kernel, can be adopted to the construction of external

code in order to increase the *trustworthiness* of such a code-generation. We propose two technical design principles to increase trustworthiness of code-generation:

- *convert* should be implemented as a primitive, one-to-one converter
- from the definition of $\rightarrow_A$, a number of theorems should be derived that mirror the rules of $\rightarrow_C$ (as given in the definition of our language) syntactically.

Of course, this syntactic correspondence of the rules of $\rightarrow_A$ and $\rightarrow_C$ does not formally guarantee that they are identical because its symbols are interpreted in different contexts. Moreover, correspondence between $\rightarrow_A$ and $\rightarrow_C$ does not guarantee the correct implementation of $\rightarrow_C$ (i.e. compiler correctness with respect to $\rightarrow_C$ – its specification – is assumed throughout this paper). Still, both design principles together force a shift of many compilation oriented activities in the overall translation into tactic theorem proving based on derived rules and thus into the safe core of an LCF-style theorem prover.

3 Preliminaries: SML, *AbstractSML* and Isabelle

Clearly, the direct use of the formal operational semantics of SML described in [15] would be most convincing for our goal of bridging the gap between a logic and a programming language. Unfortunately, since SML is a very rich language, a presentation of the formal semantics is still too complex even if restricted to the relevant language fragment; this is mostly due to the fact that SML contains imperative constructs which are out of the scope of this paper. Instead, we use the more "vanilla" operational semantics of an eager language following closely [24]. We consider the task of bridging the gap between "Real-SML" in the sense of [15] and our language – which we call *AbstractSML* – as routine.

3.1 Expressions of *AbstractSML*

The key ingredient of this operational semantics is an inductively defined subset of all terms, the so called set of *canonical forms*. The judgment $t \in C_\tau$ states that t is a canonical form of type τ:

Ground type:	$C_{int} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ and $b \in C_{bool} = \{true, false\}$
Product type:	pairs of canonical forms are canonical, i.e. $\langle t_1, t_2\rangle \in C_{\tau_1 * \tau_2}$ if $t_1 \in C_{\tau_1}$ and $t_2 \in C_{\tau_2}$
Function type:	closed abstractions are canonical forms, i.e. $(fn\ x \Rightarrow t) \in C_{\tau_1 \rightarrow \tau_2}$ if $(fn\ x \Rightarrow t) : \tau_1 \rightarrow \tau_2$ and t closed

We can now give the rules for the evaluation relation of the form

$$t \rightarrow_C c$$

where t is a typable term and c is a canonical form, meaning t evaluates to c. In the following, c, c_1, c_2 and c_3 range over canonical forms.

$$c \to_C c \qquad \text{(identity)}$$

$$\frac{t_1 \to_C c_1 \quad t_2 \to_C c_2}{t_1 \ \mathbf{op}\ t_2 \to_C c_1 \ \mathbf{op}\ c_2} \qquad \text{(operations)}$$

$$\frac{t_1 \to_C \mathit{true} \quad t_2 \to_C c_2}{\mathbf{IF}\ t_1\ \mathbf{THEN}\ t_2\ \mathbf{ELSE}\ t_3 \to_C c_2} \qquad \text{(ite-true)}$$

$$\frac{t_1 \to_C \mathit{false} \quad t_3 \to_C c_2}{\mathbf{IF}\ t_1\ \mathbf{THEN}\ t_2\ \mathbf{ELSE}\ t_3 \to_C c_2} \qquad \text{(ite-false)}$$

$$\frac{t_1 \to_C c_1 \quad t_2 \to_C c_2}{\langle t_1, t_2\rangle \to_C (c_1, c_2)} \qquad \text{(product)}$$

$$\frac{t_1 \to_C c_1 \quad t_2 \to_C c_2}{\mathbf{fst}\langle t_1, t_2\rangle \to_C c_1} \qquad \text{(fst)}$$

$$\frac{t_1 \to_C c_1 \quad t_2 \to_C c_2}{\mathbf{snd}\langle t_1, t_2\rangle \to_C c_2} \qquad \text{(snd)}$$

$$\frac{t_1 \to_C \ \mathbf{fn}\ x \Rightarrow t \quad t_2 \to_C c_2 \quad t[x := c_2] \to_C c}{t_1\ t_2 \to_C c} \qquad \text{(function)}$$

$$\frac{t_1 \to_C c_1 \quad t_2[x := c_1] \to_C c}{\mathbf{LET}\ x = t_1\ \mathbf{IN}\ t_2 \to_C c} \qquad \text{(let)}$$

$$\mathbf{REC}\ y.\ (\mathbf{fn}\ x_1 \Rightarrow \ldots x_n \Rightarrow t) \to_C$$
$$\mathbf{fn}\ x_1 \Rightarrow \ldots x_n t[y := \mathbf{REC}\ y.\ (\mathbf{fn}\ x_1 \Rightarrow \ldots x_n \Rightarrow t)] \qquad \text{(rec)}$$

With these rules, we also implicitly introduce the syntax of our target language.

3.2 Declarations in *AbstractSML*

In order to scale up to larger units, a programming language comprises mechanisms to extend the set of initially defined operator symbols, also called the *basic environment*, by user defined ones. We will make this more precise and formally define an *environment* Γ as a set of *type judgments* $x::\tau$ assigning to an identifier x a type τ. For *AbstractSML*, the base environment Γ_0 comprises judgments such as $-2 :: \mathit{int}, \mathit{true} :: \mathit{bool}$ or $+ :: \mathit{int} * \mathit{int} \to \mathit{int}$. A term t *conforms to* Γ, if all free identifiers in t are declared in Γ and if t is typable. The effect of a declaration:

$$\mathbf{VAL}\ x = t \qquad \text{(ValDec)}$$

is to extend the implicit environment Γ (on which we make the assumption that t must conform to Γ and therefore have some type τ) to the environment $\Gamma' = \Gamma\backslash\{x :: \_\} \cup \{x :: \tau\}$. Moreover, the evaluation relation $\to_C$ is extended by the rule:

$$\frac{t \to_C c}{x \to_C c} \qquad \text{(Unfold)}$$

For datatypes, we proceed analogously. The declaration

$$\mathbf{DATATYPE}\ s = \mathbf{con}_1 :: \tau_1 \mid \ldots \mid \mathbf{con}_{1_n} :: \tau_n \qquad \text{(TypeDecl)}$$

produces the new environment $\Gamma' = (\Gamma\backslash\{\mathbf{con}_j :: \_\}) \cup \{\mathbf{con}_j :: \tau_j\}$. For a legal datatype declaration, we assume s to be a fresh type and the τ_i to have either the type s or $\tau_i' \to s$. Moreover, Γ' will be

extended by an additional constant symbol $CASE_s$ (case distinction) for which the evaluation relation is extended by rules of the following scheme:

$$\frac{f_i \rightarrow_C c}{\mathbf{CASE}_s\ \mathbf{con}_i\ f_1 \ldots f_n \rightarrow_C c} \qquad \text{(CaseMatchCon)}$$

$$\frac{t \rightarrow_C c_1 \quad f_i\ t \rightarrow_C c_2}{\mathbf{CASE}_s\ (\mathbf{con}_i\ t)\ f_1 \ldots f_n \rightarrow_C c_2} \qquad \text{(CaseMatchFun)}$$

Note that we also introduce a new set of canonical forms C_s.

This concludes the definition of the "dynamic" part of our target language *AbstractSML*. From here, we can already foresee the major technical requirements for our code-generator: It must be

- able to deduce the underlying datatypes from a sequence of input theorems
- find a sequence of the input theorems that corresponds to a series of declarations
- able to convert each of the input theorems to declarations, i.e. check that the expressions on the right hand side of the declaration are indeed an *AbstractSML* expression.

Below, we turn to the logical environment (implemented in SML) in which our programming languages will be represented and in which the non-trivial conversion into code will be performed.

3.3 Isabelle

Isabelle [10] is a *generic* theorem prover that supports a number of logics, among them first-order logic (FOL), Zermelo-Fränkel set theory (ZF), constructive type theory (CTT), the Logic of Computable Functions (LCF), and others. We only use its set-up for higher order logic (HOL). Isabelle supports natural deduction style. Its principal inference techniques are resolution (based on higher-order unification) and term-rewriting. Isabelle provides syntax for hierarchical theories (containing signatures and axioms).

Isabelle belongs to the family of LCF-style theorem provers. This essentially means that the abstract data type "thm" (protected by the SML type discipline) contains all the formulas accepted by Isabelle as theorems. thm-objects can only be constructed via operations of the logical kernel of Isabelle. This architecture allows for user-programmed extensions of Isabelle without corrupting the logical kernel.

In the sequel, all Isabelle input and output will be denoted in this `font` throughout this paper. For the mathematical symbols $\forall$, $\exists$, $\wedge$ and $\vee$ we use the Isabelle notations `!`, `?`, `&` and `|`.

3.4 Higher Order Logic (HOL)

In this section, we will give a short overview of the concepts and the syntax. Our logical language HOL goes back to [7]; a more recent presentation is [4]. HOL is a classical logic with equality formed over the usual logical connectives $\neg, \wedge, \vee, \Rightarrow$ and $=$ for negation, conjunction, disjunction, implication and equality. It is based on total functions denoted by λ-abstractions like $\lambda x.x$. Function application is denoted by $f\ a$. Every term in the logic must be typed, in order to avoid Russels paradox. Isabelle's type discipline incorporates polymorphism with type-classes (as in Haskell). HOL extends predicate calculus in that universal and existential quantification $\forall x.P\ x$ rsp. $\exists x.P\ x$ can range over functions.

Most HOL-systems are used in a particular methodology: Since adding arbitrary axioms to a basic logical system like HOL is extremely untrustworthy, these systems support particular schemes of axioms – so called *conservative extensions* – that ensure consistency when building up larger libraries (see [16]). Following common usage, we will use the term HOL-theory also for all its conservative extensions.

4 The Generic Framework

In this section, we will explain the overall structure of our generic coding scheme in more detail. The technical aspects of our generic coder are discussed later in section 5. As a starting point,we will refine the diagram of Fig. 1 and develop a separation of our coding scheme into different phases.

In accordance to Fig. 1, we subdivide *abstract programs* into two languages $\langle X \rangle Lang$ and $\langle X \rangle AbstractSML$ – here, X stands as a placeholder for concrete program notions (such as *Wf*, *Fix*, *Lfp* to be discussed later). $\langle X \rangle Lang$ is the language that is the source of the coding process. Since we instantiate our target language with SML throughout this paper, we use $\langle X \rangle AbstractSML$, which is the semantic interface to our target language SML. The abstract operational semantics $\rightarrow_C$ must only be established for $\langle X \rangle AbstractSML$. In such a setting, the coding process can now be described as a sequence of six compilation phases. Except for the last one, the phase *convert* already discussed in the introduction, they consist of tactics based on derived rules. In more detail, the five phases are:

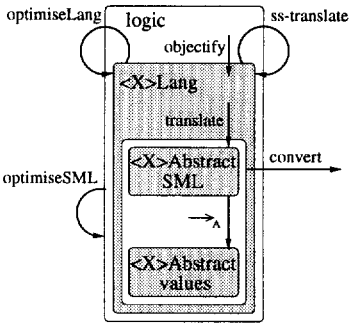

Fig. 2. The redefined coding scheme

- *objectify* attempts to convert an arbitrary term of our logic in a term of our programming language. This phase is the key-ingredient for increasing the coverage of the coder.
- *optimiseLang* is used for language-dependent compiler optimisations, e.g. data type dependent rules like the associativity of concatenation of lists, which improves the efficiency of the evaluation since `(a @ b) @ c` is usually more expensive than `a @ (b @ c)`.
- *ss-translate* is a source-to-source translation preparing the next phase called *translate*. We allow *ss-translate* to produce terms that contain language constructs belonging to the $\langle X \rangle AbstractSML$-language.
- *translate* maps (impure) $\langle X \rangle Lang$-terms to $\langle X \rangle AbstractSML$.
- *optimiseSML* can be used as code-optimisation on the *AbstractSML*-level.

In the next section, we will describe three instantiations of our generic scheme mapping X to *Wf*, *Fix* and *Lfp*, representing a program notion for total functions, functional programming and logic programs.

5 The Instances WF, FIX and LFP

We are now ready for our main task, the representation of three different programming languages in HOL, their proof of correctness in our sense, and the development of derived rules and tactics that perform the major part of the coding.

We will demonstrate our technique in most detail for the first language, which is also the simplest one. In the theory `WFLang`, besides the syntax, we define the semantics of the operators and language constructs of our language in terms of HOL. Subsequently, in the theory `WFSML`, the semantics of language constructs of SML in terms of `WFLang`. Due to the deliberate simplicity of `WF` (that is designed to form a reasonable subset of HOL-formulas that should be converted into SML-code), these two language mappings are mostly trivial, resulting in a correctness proof (i.e. a derivation of $\rightarrow_C$ that is extremely simple).

The language mappings in the subsequent instances will contain more and more complexity demonstrating the flexibility of our concepts. `FIX` will introduce exception and recursion partiality into the language, while `LFP` extends it by some controlled form of backtracking. Intuitively speaking, the "distance" expressed in the mapping between $\langle X \rangle Lang$ and $\langle X \rangle SML$ grows during the sequence of these instantiations.

5.1 The instance WF

The theory `WFLang` is based on HOL-theories providing basic semantics for boolean and numerical operators:

```
WFLang = WF + ... +
consts
(* library of basic operations *)
TRUE :: bool
...
ZERO :: nat
...
```

Analogously, we declare the operator symbols FALSE, NOT, AND, OR, ONE, TWO, SUC, LEQ, etc. These constant symbols represent the sets of canonical forms C_{bool} and C_{nat}. Note that the constant ϵ x.True (also called arbitrary) cannot be included in the set of canonical forms since the Hilbert-Operator is interpreted by definition differently in any model of an HOL-formula; correct code, however, will have to produce values that exist in all models. On top of the denotations for canonical forms, we will now define the operators of our language such as:

```
LESS :: [nat,nat] -> bool
        LESS a b = a < b
...
```

As a consequence of the fact that arbitrary is not a canonical form, we must rule out partial operators like div or hd (on lists) from the language WF. A special case for the operators is the equality, which is declared polymorphically for all canonical terms, but has to be constrained to a class EQ of base types and cartesian products over them, ruling out the function types, for which $\rightarrow_C$ cannot be used to compute EQ and is therefore also ruled out in SML:

```
EQ :: ['a:EQ,'a] -> bool
```

The definition of these constructs one-to-one corresponds to the HOL operations, as a consequence of the design of WF:

```
TRUE_def  "TRUE = True"
...
ZERO_def  "ZERO = 0"
```

Next, we define the basic datatypes unit and pair: the operators UNIT, PAIR, FST, and SND that were trivially represented by their HOL-counterparts (), (_,_), fst and snd respectively.

We may now turn to the core of WFLang, i.e. the main language constructs. Again, the constant definitions are straight-forward mappings to standard operations:

```
^!      :: ('a => 'b) => 'a => 'b
           f ^! x    == f x
Lam     :: ('a => 'b) => ('a => 'b)
           Lam f     == f
IF      :: [bool, 'a, 'a] => 'a
           (IF a THEN b1 ELSE b2) == (if a then b1 else b2)
LET     :: ['a, 'a => 'b] => 'b
           LET s f   == f ^! s
REC     :: ('a * 'a)set => (('a=>'b) => ('a=>'b)) => 'a => 'b
           REC(m)(f) == wfrec m f
```

The characterising feature of each program notion is the notion of recursion, i.e. some instance of the general scheme:

$YF = F(YF)$, where F must fulfill some requirement A

The "workhorse" for most definitions of total functions in the library of HOL is the well-founded recursion. Even the definitions of primitive recursive functions such as concatenation on lists is internally mapped to well-founded recursion. Thus, it is suggestive to define REC_m (parameterized by a well-founded ordering m) in WFLang by

```
wfrec :: ('a * 'a)set => (('a=>'b) => ('a=>'b)) => 'a => 'b
```

developed in the theory WF in the library in Isabelle/HOL. The main result of the theory of well-founded recursion is:

```
wf(r) = wfrec r H a = H (cut (wfrec r h) r a) a                [wfrec]
```

where the predicate `wf : ('a * 'a)set -> bool` states the well-foundedness of a relation and where `cut f r x` constructs a function that is identical to `f` for all smaller values than `x` w.r.t. the ordering `r` and undefined (i.e. ϵ`x.True`) for all larger values.

`wfrec` is already close to our desired recursion scheme. The missing link is the concept of *coherence*:

```
!a. H (cut (wfrec r H) r a) a = H (wfrec r H) a
```

which essentially states that the body `H` uses the function `wfrec r H` (hence the recursive "call") always with smaller arguments. Well-foundedness and coherence together establish the desired fixpoint property for `wfrec` along `[wfrec]`. The problem with our representation of REC_m is that we need well-foundedness and coherence, i.e. additional semantic information for each occurance of REC_m in an abstract program assuring that the fixpoint property holds – this problem will reappear in different form in our other program notions. This leads to the definition of a kind of SML-like statement that contains the code plus the semantic information necessary to establish the fixpoint property of the recursor:

```
WFProg :: [('a * 'a)set, 'c, (('a => 'b) => ('a => 'b)),
           ('a => 'b) => 'c] => bool
WFProg m f F E == (f = (let x = (wfrec m f) in (E x)
                  & (wf m)
                  & (!a. cut(wfrec m F) m a = (wfrec m F))))
```

The following syntactic sugar paraphrases this complex definition as an SML-like statement annotated with semantic information:

```
val f = let fun F in E measure m
```

This completes the definition of WFLang. We turn now to our semantic interface to SML, called WFSML, which is defined as a theory extension of WFLang.

```
WFSML = WFLang +
```

and provides definitions of operators TRUE', FALSE', NOT', ..., ZERO', ONE', ..., LESS', ..., UNIT', PAIR', ..., all defined identical to their unprimed counterparts from WFLang. Here, we only show the definitions for the core language constructs:

```
^!'     :: ('a => 'b) => 'a => 'b
           f ^!' x == f ^! x
Lam'    :: ('a => 'b) => ('a => 'b)
           Lam' f == f
IF'     :: [bool, 'a, 'a] => 'a
           (IF' a THEN' b1 ELSE' b2) == (IF a THEN b1 ELSE b2)
LET'    :: ['a, 'a => 'b] => 'b
           LET' s f == LET s f
REC'    :: ('a * 'a)set => (('a=>'b) => ('a=>'b)) => 'a => 'b
           REC' == REC
```

As a next step, we show the correctness of our language representation for SML, i.e. that we can derive the operational semantics of *AbstractSML* rules (in the sense of chapter 3). First, we define the evaluation relation $\rightarrow_A$ by the semantical equality:

```
WFSML_CT = WFSML +
constdefs     eval     :: ['a,'a] => bool
              eval s t == (s = t)
syntax        -A->     :: ['a,'a] => bool
translations  s -A-> t == eval s t
end
```

The predicate `cf c` (`c` is a canonical form) is simply set to `true` in `WF` since we leave this check to the meta-level. Now we derive the operational semantics:

```
[| cf c1; cf c2; t1 -A-> c1; t2 -A-> c2 |] ==>
         (LESS' t1 t2) -A-> (c1 < c2)
[| cf c2; t1 -A-> TRUE'; t2 -A-> c2 |] ==>
         (IF' t1 THEN' t2 ELSE' t3) -A-> c2
[| cf c2; t1 -A-> FALSE'; t3 -A-> c2 |] ==>
         (IF' t1 THEN' t2 ELSE' t3) -A-> c2
[| cf c1; cf c2; t1 -A-> c1; t2 -A-> c2 |] ==>
         PAIR' t1 t2 -A-> (c1,c2)
[| cf c1; cf c2; t1 -A-> c1; t2 -A-> c2 |] ==>
         FST'(PAIR' t1 t2) -A-> c1
[| cf c1; cf c2; t1 -A-> c1; t2 -A-> c2 |] ==>
         SND'(PAIR' t1 t2) -A-> c2
[| cf c; cf c2; t1 -A-> (LAM' x. t x); t2 -A-> c2;
         (t(c2)) -A-> c |] ==>
         (t1 ^!' t2)  -A-> c
[| cf c; cf c1; t1 -A-> c1 ; (t2(c1)) -A-> c |] ==>
         LET' t1 (%x. t2 x)  -A-> c
[| ! a. cut (wfrec m f) m a = (wfrec m f); wf m |] ==>
         REC m (% X. (LAM' x1. f X x1)) -A->
         (LAM' x1. f (REC m (%X. (LAM' x1. f X x1))) x1)
```

and we are home and dry! By syntactical correspondence, we check that our derived formal rules for $\rightarrow_C$ correspond to the rules $\rightarrow_C$ in chapter 3.1.

The remaining steps are merely technical: First, the code-generator has to be set up to generate the definitions for the implicit `CASE'`-rules (cf. section 3.2) and to generate code for datatypes including the involved recursors. Second, we have to describe the compilation phases in the sense of chapter 4. For `WF`, these phases are fairly trivial – for `ss-translate` and `optimiseSML` we use just the identity and for `optimiseLang` just a simplification tactic for some set of equations like associativity on lists. For `translate` we use a simplification tactic that folds all definitions of `WFSML` from right to left. In this setting, the preparational `objectify` is the most complex phase. We use it to convert equations from primitive recursive definitions like the following for the concatenation of lists (drawn from the Isabelle/HOL-library):

```
primrec "op @" list
  [] @ ys   = ys
  (x#xs)@ys = x # (xs @ ys)
```

into the term of the abstract programming language:

```
val (op @) = let fun f x ys = CASE x OF
                                [] => ys
                              | (x # xs) => x # (f ^! xs ^! ys)
             in f measure length
```

Constructing this representation also requires reasoning over the internal representations of datatypes and subterm orderings used in Isabelle/HOL's datatype package.

5.2 The instance FIX

The language WF, constrained to total functions, had to rule out values like `hd []` or `3 div 0`. When admitting partial functions, there is the well-known choice for the semantics of function application reflecting *call-by-value* evaluation or *call-by-name* evaluation (cf. [24]) . An appropriate semantical framework for tackling these issues is denotational semantics, which we use as basis for our second program notion FIX.

There are many known formalisations of denotational semantics in HOL-systems. For Isabelle/HOL, there is most notably HOLCF [19]. Instead, we will use the generic theory of Scott-cpo's `Fix.thy` described in [23]. Both theories have much in common and could be exchanged in this context with minor effort; we preferred our own version mostly due to its lightweightness.

In the following, we briefly review `Fix.thy` and its pivotal definitions. cpo's are introduced by a sequence of *axiomatic class* definitions, i.e. an extension of the Haskell class system with semantic constraints in Isabelle. For instance, the class of partial order types `order` is defined by the statement:

```
axclass order < ord
  le_refl    x <= x
  le_trans   [| x <= y; y <= z |] => x <= z
  le_antisym [| x <= y; y <= x |] => x = y
```

After the usual definitions for bottom $\perp$, directed sets, upperbounds, least upperbounds etc., the class `order` is extended to the class `cpo`. In this class, the predicate for continuity `cont::('a::cpo -> 'b::cpo) -> bool` and the fixpoint operator `fix::('a::cpo -> 'a) -> 'a` (defined as least upperbound of the directed set of function iterations) were defined and provide as main result:

```
cont f => fix f == f (fix f)                          [knaster-tarski]
```

which will give semantics to our recursor `REC` in our program notion `FIX` defined in `FixLang.thy`:

```
FixProg f F == f = fix F & cont F
```

Having settled the fundamental questions, we turn now to the basic datatype representations `bool` and `nat`. In order to embed them into cpo's, we employ the well-known lifting technique into flat domains by defining the type constructor:

```
datatype 'a up = lift 'a | down
```

After identifying `down` with $\perp$ and providing the usual ordering, the fact that each instance of type-constructor `up` is of class `cpo` is made explicit to Isabelle's type-system:

```
instance up :: (term)cpo
```

Analogously, pairs and function spaces are shown to be instances of `cpo` provided that both arguments resp. the last argument of the type-constructors are instances of `cpo`. The basic types `Bool`, `Nat` and `Unit` in `FixLang` were defined by lifting the underlying corresponding types of the HOL-library via `up`. The definitions of the basic operations are straight-forward as strict extensions of the underlying HOL-operations. In contrast to `WF`, however, we can define partial functions analogously to `DIV`:

```
constdef DIF :: [Nat, Nat] -> Nat
         DIV == strictify(%x::nat. strictify(%y.
                                    if y=0 then UU
                                    else lift(x div y)))
```

The definition of the language constructs of our functional language FixLang is also straight-forward in denotational terms of our cpo-theory Fix. As example, we only show the application, that is directly mapped to the HOL-application since we already have proven that the standard function space has cpo-structure:

```
constdef ^!      :: ('a -> 'b::cpo) -> 'a -> 'b
         F ^! x == F x"
```

At this point, we conclude our presentation of FixLang and turn to the semantical interface FixSML of our call-by-value target language. Here, a crucial point is an adequate denotational representation of abstractions that were treated as *closures* in the operational semantics. In order to distinguish LAM x.⊥::'b, that is equal to the least element in the function space ⊥::'a→'b, from LAM' x.⊥, that is the *closure* of the computation yielding ⊥ (as in SML) and hence a canonical form or a value, LAM' must lift any function (see also [24], pp.188). Thus, it is suggestive to introduce an own type constructor for the lifted function space →! and define:

```
FixSML = FixLang +
types  ('a,'b) "=>!" = "('a => 'b) up"
...
constdefs
  ^!'        :: ['a=>'b::cpo,'a] => 'b
  F ^! x     == if x = UU then UU
                          else if F = UU then UU
                                         else (drop F) x
  Lam'       :: ('a::cpo =>'b::cpo) => ('a =>! 'b)
  Lam' f     == lift f
  ...
```

where drop is just the inverse to lift. The definitions for IF', LET', REC' etc. are straight-forward and omitted together with the mappings of the FixLang-operators to the FixSML-operators, which are just appropriate liftings w.r.t. →!.

The key question for the code-generation in FIX is the translation of the call-by-name versions of Lam and ^! to their call-by-value counterparts Lam' and ^!' respectively (for pairing and projection, the situation is similar). The key for a solution is the definition of the suspension resp. the forcing functions (see also [9]):

```
delay      :: 'a::cpo => 'a del
delay f    == (LAM! x. f)
force      :: ('a::cpo)del => 'a
force f    == (f ^! UNIT)
```

where 'a del is a type synonym for Unit ->! 'a. Note that delay and force are already "pure SML" and can thus be converted easily. These definition leads to the following derived theorem, that allows the exchange of all lazy applications by eager ones:

```
(f ^! a) = ((forcify f) ^!' (delay a))                    [lazy2eager]
```

where forcify is defined by LAM! x. f (force x)). The translation is possible by using this rule in *all* applications in FixLang; however, this technique leads to quite inefficient code. A remedy to this problem – well known from compiler-construction [9] – is a strictness-analysis that we mimic in our approach by the following derived rules:

```
is_strict f ==> (f ^? a) = ((lift f) ^! a)
is_strict(LAM! x. x)
is_strict(%x. UU)
is_strict(strictify f)
is_strict(NOT)
is_strict(SUC)
[| !a. is_strict f |] ==> is_strict (%x. f^! x ^! a)
[| !a. is_strict (f ^! a) |] ==> is_strict ((lift f) ^!' a)
is_strict f ==> is_strict (%x. (IF (f x) THEN (g x) ELSE (h x)))
...
```

where `is_strict f` is defined by `f` $\bot=\bot$. The first rule of the list above represents our optimised translation, that requires strictness, while the other rules try to establish this property (note that the list is incomplete). For all applications, where this did not succeed, the rule `lazy2eager` is applied.

We now briefly describe the overall technical organization of the coding in phases: *objectify* maps applications, abstractions, and constructs like if_then_else from HOL to *FixLang*-terms. *optimiseLang* and *optimiseSML* are again set to identity. The translation from `^!` to `^!'` (as described above) is a classical source-to-source-translation and goes to *ss-translate*. Finally, *translate* is used to map basic operators from *FixLang* to *FixSML*. A little example may illustrate the steps in more detail:

```
((% x y. y) (DIV ONE ZERO) TWO)
```

$\Downarrow$ {*objectify*}

```
(LAM!x y. y ^! (DIV ^! ONE ^! ZERO) ^! TWO)
```

$\Downarrow$ {*ss_translate*}

```
(LAM!'x. lift (LAM!y. y) ^!'
          delay (lift(lift DIV ^!' ONE) ^!' ZERO') ^!' TWO)
```

$\Downarrow$ {*translate*}

```
(LAM!'x y. y ^!' delay (DIV' ^!' ONE' ^!' ZERO') ^!' TWO')
```

The function `%x y. y` is strict in its second, but not in its first argument. Hence, the evaluation of the first argument must be delayed – which happens to be undefined in this example. For the second argument, no suspension is needed and can be avoided for efficiency reasons.

Finally, we turn to the question of correctness of this coding scheme. We define the relation $\rightarrow_A$ in `FIXSML_CT` analogously to `WFSML_CT` (see previous section) except that we define canonical forms `cf` as "not being $\bot$". Thus, we consider cases like `DIV ONE ZERO` as an *exception*. In `FIXSML_CT`, we derive all operational rules of section 5.1. (although they are based for a completely different semantic interpretation). Additionally, we also have operational rules that cover exceptional behaviour:

```
[| ~cf c1; t1 -A-> c1; t2 -A-> X |] ==> (t1 ^!' t2) -A-> UU
[| ~cf c2; t1 -A-> X; t2 -A-> c2 |] ==> (t1 ^!' t2) -A-> UU
```

Although these rules do not appear in [24], they can be justified by the *exception convention* (cf. [15], pp. 40) in the SML-standard. With this proof of correctness, which is still fairly simple in Isabelle/HOL but no longer trivial as in *Wf*, we conclude the presentation of the coding scheme for *Fix*.

5.3 The instance LFP

The language LFP is inspired by the semantic embedding of Z in HOL (see [12]) and previous work on animation tools for Z-specifications (see [6]), [8]). Z is based on a typed set-theory – as available in HOL – and represents all functions by their graph, i.e. a set of pairs:

```
LfpLang = Set + Lfp + EqnSyntax + Arith +
types ('a,'b) "<=>" = ('a*'b) set
...
LAM     :: ['a => 'b]    => ('a <=> 'b)
%^      :: ['a<=>'b,'a] => 'b
...
```

This results in a formulation of partial functions that is similar to *Fix* with respect to the necessary conversion between call-by-name-semantics in LfpLang to call-by-value semantics in LfpSML. In the setting of LFP, the recursor REC is based on the usual least fix-point `lfp::['a set -> 'a set] -> 'a set` enjoying the property:

```
mono(f) => lfp(f) = f(lfp(f))                          [lfp_Tarski]
```

which is already established in the Isabelle theory LFP in the library. This is exactly what we want for our program statements which we define as follows:

```
LfpProg f F == f = lfp F & mono F
```

Although the semantic foundation – as outlined above – is totally different, the coding machinery is similar to the one described in FIX. Hence, we will refrain from a further formal presentation of this program notion and concentrate on the new aspects here. In this case, there is a new additional basic datatype `'a set`, that is represented by a lazy list `'a seq` in *AbstractSML* and SML (in both languages, `'a seq` can be defined on top of the already introduced language constructs; see [18]).

When implementing a set by a sequence, we require that the sequences must be duplicate free in order to establish in a simple way evaluation fairness, i.e. any element of a set will be constructed eventually by the evaluation, provided there are "enough tail selections" into the lazy list. These semantic side conditions have to be encapsulated similarly to the side-conditions for the recursor in program-statements. When these side-conditions are fulfilled, it is easy not only to provide a MAP and FILTER on sequences, but also a UNION as an interleave of two sequences. The definitions of MAP, FILTER and UNION can be expressed in terms of a program statement on top of the LfpLang; they do not add extra semantical complexity. Thus, the set of natural numbers N characterized by `lfp(%X. UNION{ZERO} (MAP SUC X))` is a program. Moreover, since ZF-expressions like $\{x : N \mid even\ x\}$ can be seen as an equivalent to a FILTER on N and is consequently also a program. This turns many definitions in the Z-library, the *Mathematical Toolkit* into programs, among them the definition for cartesian products and finite function spaces.

In order to reveal the power of this program notion, we show the example EightQueens stemming from [11]. In the EightQueens-problem, eight queens must be placed on a chess board with eight files and eight ranks such that no queen attacks any other, i.e. sits in the same file, rank or diagonal. We use Z-Notation here in order to keep the presentation compact:

Lib

$$up, down : SQUARE \rightarrow DIAG$$

$$\forall f : FILE; r : RANK \bullet up(f, r) = r - f \wedge down(f, r) = r + f$$

EightQueens

$squares : FILE \twoheadrightarrow RANK$

$\{squares \lhd up, squares \lhd down\} \subseteq SQUARE \mapsto DIAG$

where FILE and RANK are sets from 1 to 8 and SQUARE is the set of pairs of FILE- and RANK-positions a queen may sit in. The function definitions for up and down map to any queen position its up resp. down diagonal number. The set of EightQueen-solutions is described in the schema *EightQuenns*: Since SQUARE can be viewed as a relation, it is possible to require that each concrete solution squares must be a bijective function $\twoheadrightarrow$). Moreover, if the functions up and down where constrained to the positions of a solution squares, they must yield injective functions.

And here is the point: This example of a fairly declarative specification for a small tricky problem represents a program in LFP. All involved sets (including the set of bijective functions from FILE to RANK) are representable as combinations of MAP, FILTER, and UNION, such that the specification above (based on a small library of LfpLang-programs defining $\twoheadrightarrow$ and $\lhd$ etc.) can be translated into SML-code, that eventually enumerates the set Queens (patience required!).

6 The Generic Code-Generator

In this chapter we will shortly describe the SML-based implementation of the generic code-generator. The general idea is to implement the coder as an SML functor. This functor will be instantiated with three structures that implement our three programming languages WF, FIX and LFP described in chapter 5. The following SML signature shows the language dependent interface to the functor:

```
signature LANGUAGE =
sig
  val target     : theory; (* semantic interface to SML *)
  val lang       : theory; (* syntax and semantics of abstract
                              programming language *)
  val objectify        : thm -> thm
  val optimizeLang     : thm -> thm
  val ss_translate     : thm -> thm
  val translate        : thm -> thm
  val optimizeSML      : thm -> thm
  val convert          : thm -> Absy.absy
  ...
end;
```

Here, target of the Isabelle type theory represents the semantic interface to SML, i.e. the theory that describes the abstract programming language. The theory lang represents the source language of the coding process.

The six coding phases described in chapter 4 are implemented by the corresponding functions objectify, optimiseLang, ss_translate, translate, optimiseSML and convert. All these functions get a theorem of the Isabelle datatype thm that represents a program as argument. Except convert, all functions return again a theorem. The function convert returns the term of a program in the abstract syntax of SML, where term is the basic Isabelle data structure for terms and absy the type for the abstract syntax of SML of New Jersey.

The signature also provides functions that get information on datatype declarations based on an arbitrary datatype package. The signature of the functor looks as follows:

```
signature CODER =
sig
  exception NoCode of string
  val coder : string * (thm list) -> ()
end;
```

This signature provides the main coding function `coder`. It gets a string representing the name of the SML structure to be generated and a list of theorems that represent the programs to be compiled. There is an exception `NoCode` that will be raised if the process of code-generation fails. First, the function `coder` has to generate a graph representing the call dependencies of the various programs. Based on this graph, `coder` will sort the programs topologically. If any call cycles are detected the exception NoCode is raised. Then, `coder` retrieves the datatype informations and generates the corresponding SML datatypes and recursors. Now the six coding functions can be called. If the function `objectify` rejects a program not to be compilable, again the exception `NoCode` is raised. Finally, `coder` generates the abstract syntax tree of the final program and writes the pretty printed string to a file.

7 Conclusion

First, we have presented a method to formally investigate the correctness of code-generation schemes. Second, we have demonstrated its feasibility by instantiating it for three program notions, ranging from executablility suited for HOL, functional programming and Z. Third, we provide a technical framework for *implementing* trustworthy *code-generators* (i.e. in its crucial parts formally proven correct) based on the set of abstract programs, i.e. the executable sublanguage of HOL, and *AbstractSML*, i.e. a semantic interface to the target language. Our technique is based on a so called *shallow embeddings*, i.e. no explicit syntax is used to represent *AbstractSML*; rather, the semantics is represented via semantic operators directly embedded in HOL.

We argued that the formal proof of correctness of a code-generation in an absolute sense is impossible – at the very end, extra-logical arguments have to be used anyway.

By shifting the formalisation of *canonical forms* partially to the meta-level (including appropriate checks on the SML-level in the implementation), it is possible to stick to shallow embeddings and to make the proofs of correctness *substantially easier*. But there is a price to pay: In our approach, the proof of "canonicity" or normal-formedness is left to extra-logical reasoning. Still, we believe our approach represents a good compromise in the attempt to minimise the set of extra-logical assumptions and to base the phases of a code-generation on derived rules controlled by tactics.

From our experience with the nitty-gritty details of our code-generation schemes, it is not fully understandable why code-generation is traditionally treated as a side-issue; bugs in a code-generator are as damaging for the overall correctness than bugs in the logical engine of a prover. We hope that our technique can contribute to turn the formal investigation of code schemes used in compilers into a routine task.

7.1 Related Work

In the literature, there is a large body of papers in compiler verification. Typically, two explicit abstract syntaxes for the input and output language of the compiler were defined as data types, then a compiler function connects them. The proof of correctness is then based on two semantical interpretation functions and their commutability via the compiling function. In contrast to work along this style of representation – so called deep embeddings – our work is based on *shallow embeddings* for reasons discussed above. Moreover, our work is intended to be a component of a formal development environment. On the basis of shallow abstract-language encoding, far more activities can be founded than just compilation – interactive verification and transformation, for example.

Beyond this classical compiler verification projects, there are attempts to integrate programming- and specification languages. The work of Slind [21] presents a pure syntactical approach to the representation of programs on the level of the input of a theorem prover (Isabelle and HOL). The user can define functions in a very rich and compact functional notation employing powerful pattern-matching, that is parsed away into an WFREC-style semantic representation when the input-file is loaded. As a consequence, it is not possible to *derive* programs during theorem proving, which was one of our major goals.

In [6], a bridge from code-generation to specification animation is built. Here, the idea is to represent sets in Z-specifications (corresponding exactly to sets in HOL) by enumeration functions that produce the elements of a set (*animate* it) one by one. In this view, set comprehensions are constructed by powerdomains, such that the operational view of the animation is deliberately different to the Z standard semantics.

7.2 Future Work

We will attempt to improve the portability of the code-generator to other SML-Compilers (such as POLY or Harlekin) and, to a lesser extent, to other SML-based theorem prover environments like LAMBDA or HOL/HOL. Moreover, our actual ad-hoc treatment of datatypes should be replaced by a more general mechanism, possibly better integrated in a future version of Isabelle.

It is worth investigating the increase of genericity with respect to the target language: It should not be too difficult to develop other converters (or other, more general intermediate abstract syntaxes) to languages like C++, Java or Haskell; for the latter, a code-scheme could be conceived supporting some type classes of Isabelle.

References

1. The common algebraic specification language. http://www.brics.dk/Projects/CoFI/.
2. The coq project. http://pauillac.inria.fr/coq/biblio-eng.html.
3. Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
4. P.B. Andrews. *An Introduktion to Mathematical Logic and Type Theory: To Truth Through Proof.* Stockholm Studies in Philosophy. Academic Press, Stockholm, 1986.
5. M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with kiv. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering.*, number 1783 in LNCS. Springer, 2000. http://www.informatik.uni-ulm.de/pm/kiv/tools/index.html.
6. P.T. Breuer and J.P. Bowen. Towards correct executable semantics for Z. Availiable online.
7. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56– 68, 1940.
8. W. Grieskamp. *A Set-Based Calculus and its Implementation.* PhD thesis, Technische Universität Berlin, 1999. http://uebb.cs.tu-berlin.de/w̃g/diss.ps.gz.
9. J. Hatcliff and O. Danvy. Thunks and the lambda calculus. *Journal of Functional Programming*, 7:303– 319, 1997.
10. The Isabelle documentation page. www.informatik.tu-muenchen.de/ nipkow/isabelle.
11. J. Kacky. *The Way of Z - Practical Programming with Formal Methods.* Cambridge University Press, 1997.
12. Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle. In J. von. Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1125 in LNCS, pages 283 – 298. Springer Verlag, 1996.
13. P.G. Larsen and P.B. Lassen. An executable subset of meta-iv with loose specifications. *Formal Software Development Methods*, 1, 1991.
14. C. Lüth and B. Wolff. Generic window inference with tas. In *Proc. TPHOLs '00*, lncs. Springer — to appear, 2000.
15. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML (revised).* MIT Press, 1997.
16. M.J.C.Gordon and T.M.Melham. *Introduction to HOL: a Theorem Proving Environment for Higher order Logics.* Cambridge University Press, 1993.

17. G. O'Neil. Automatic translations of vdm specifications into standard ml programms. *The Computer Journal*, 35:623– 624, 1992.
18. L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.
19. F. Regensburger. *HOLCF: Eine konservative Einbettung von LCF in HOL.* PhD thesis, Technische Universität München, 1994.
20. T. Santen. On the semantic relation of z and hol. In J. Bowen and A. Fett, editors, *Proc. ZUM '98*, number 1493 in lncs, pages 96—115. Springer, 1998.
21. K. Slind. Function definition in higer order logic. In J. von. Wright, J. Grundy, and J. Harrison, editors, *TPHOLs 96*, number 1125 in LNCS. Springer Verlag, 1996.
22. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1992. 2nd edition.
23. H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Proceedings of the FME '97 — Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 318–337. Springer, Berlin, 1997.
24. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.

A New Proof Format for Linking Theorem Provers

Wai Wong and *László Németh*
Hong Kong Baptist University

Abstract. This paper describes a proof file format for linking theorem provers. The format is an extension from previous work by Wong and Curzon. The new format is an application of the Extensible Markup Language (XML), i.e., proof files in this format will be valid XML documents. A proof recorder and a checker for this format have be implemented in HOL98.

1 Introduction

Mechanized and automated verification tools, such as HOL[GM93], Coq[Coq00], PVS[SORSC99], to name but a few, have been used in many formal verification projects. Each of these systems is based on a formal logic, and each has its own strength on certain application areas.

In large formal verification project, it is often necessary to involve more than one team of researchers. Each team may use a different verification system. Information exchange takes place between these teams of researchers. In addition to textual documents, theories, theorems and proofs in machine-readable form need to be passed around.

The objective of the work described in this paper is to develop a format that is suitable for exchanging formal proofs, theorems and theories between theorem provers. This work is based on the previous works by Wong and Curzon[WC97] in which a proof format for recording and checking HOL proofs had been developed.

In order to achieve this goal, the proof format has to be flexible, simple and reasonably compact. To be flexible means that the proof format is able to accommodate proofs generated by different versions of provers based on the same logic as well as provers based on different logics. To be simple means that it should be easy to add support to provers to generate and to receive proofs in such a format. To be compact means that the size of the proof in such a format should be small.

In fact, formal proofs can be considered as a well structured documents. Based on the previous proof format, we model a format proof as a sequence of inference steps, i.e., Hilbert style. This is a fairly general and simple model. Thus, a proof can be structured into a list of inference steps. Using the Extensible Markup Language (XML), that is rapidly becoming very popular for structured documents, to encode the proofs, we can take advantages of many general tools for processing XML documents. Furthermore, the extensibility of XML provides flexibility so that proofs in many different logics can be accommodated.

2 An Introduction to XML

Extensible Markup Language (XML) [W3C98] is a relatively new language to describe structured documents. The specification of XML became the official recommendation of W3C in February 1998. Two of its properties make it attractive as a language to describe proofs: its

simplicity and the wide availability of third-party tools to process, transform, display XML documents. Its verbosity however is a definite drawback, but it has some facility that can relieve this problem (to be described later).

XML documents consist of two main parts: the prolog and the body. The prolog includes a grammar, known as the Document Type Definition or DTD. The DTD may be stored in a separate document with a hyperlink pointing to it. The DTD is essentially BNF, and it constrains the body of the document and amongst others it includes language identification instructions. XML tools are expected to handle a wide range of character encodings including Unicode. This makes XML particularly suitable to describe proofs as fancy mathematical symbols are easily available.

An XML document is *well-formed* if (1) it matches the root element specified in the DTD, (2) satisfies all the well-formedness constraints of the specification and (3) each of the referenced entities (either directly or indirectly) is well-formed. *Validity* of XML documents is a stronger criterion, which ensures that the document is syntactically correct according to the DTD. We use the well-formedness and validity of XML documents to ensure syntactic consistency of encoded proofs. However, XML is not quite rich enough to ensure correctness of the proofs.

Within the DTD the declaration of a non-terminal begins with the `!ELEMENT` keyword, and the declaration is surrounded by < and >. Just like in BNF, XML uses the | symbol to express alteration, comma to express sequential composition, and parenthesis are used to denote grouping. The * symbol is used to denote zero or more occurrence and the + symbol to denote one or more occurrence of the preceeding non-terminal.

Another very useful feature of XML is the entity declarations. These are like macro expansion, except that they cannot have arguments. We use these entity declarations to abbreviate constants that appeared in a proof frequently, therefore drastically reducing the proof size.

3 The Proof Format

The grammar, i.e., the DTD, describing the new proof format is given in Fig. 1. It is not much different from proof formats previously proposed by Wong [WC97], but instead of complicating the proof format with size reducing techniques, it uses the entity declaration feature of XML to achieve the same goal. It employs some of the techniques proposed by Wong to reduce the proof's size, for example constant tables to avoid repeating frequently occurring constants and their types. From the programmer's point of view these tables are handled transparently. It also uses some tricks specific to XML to achieve further reduction. For example the entity names we generate are in base 52 and mapped onto letters only, so every name is a valid XML name. This gives us extremely short references and cuts down the proof size approximately by 10%.

Since the proof format grew out of earlier work on HOL proof recorders it is biased towards a forwards style proof: the proof is a sequence of basic inference steps. Nevertheless, the recorder is still capable of recording proofs performed in backwards style. The meaning of the inference steps is not formalised within the format. More refined proof formats, for example where the names of inference steps are enumerated, are possible, but in most cases it results in larger files as the added structure needs more tags. This also would require changing the proof format whenever a new inference rule is added to HOL.

The most significant advantage of using XML in the proof format is that the proof format becomes extensible. This means, a DTD describing a new logic can be embedded in the proof

```
<!ELEMENT ProofFile (Name, Proof+)                          >
<!ELEMENT Proof     (Name, Hyp*, Step*)                     >
<!ELEMENT Hyp       (Term+)                                 >
<!ELEMENT Term      (Const|Var|(Term, Term)|(Bvar, Term))   >
<!ELEMENT Bvar      (Term)                                  >
<!ELEMENT Const     (Name, Type)                            >
<!ELEMENT Var       (Name, Type)                            >
<!ELEMENT Type      (TyVar|TyConst|(TyOp, Type+))           >
<!ELEMENT Step      (Rule, E+, Hyp)                         >
<!ELEMENT E         (Hyp|Term|Type|(E, E))                  >
<!ELEMENT Rule      (#PCDATA)                               >
<!ELEMENT TyVar     (#PCDATA)                               >
<!ELEMENT TyConst   (#PCDATA)                               >
<!ELEMENT TyOp      (#PCDATA)                               >
<!ELEMENT Name      (#PCDATA)                               >
```

Fig. 1. The new, XML based, proof format

file together with the proof(s). The receiving prover will be able to examine the DTD and determine how to handle the proofs. On the other hand, the proof generating prover is able to dynamically generate proofs in different versions or different logics.

4 The Implementation

4.1 The Recorder

Having designed the proof format, a proof recorder in HOL98 was implemented. The implementation of the recorder is straightforward and follows earlier work. Each basic inference rule in HOL is augmented with a call to a function which records the given inference rule. It saves the name of the inference rule, all the arguments and the resulting theorem into a list. Once the proof of the theorem being proved is finished the list is traversed, constants (with their types) occurring more than once are collected into a constant table and written into a file. This table is implemented as a sequence of XML `ENTITY` declaration. Then the proof itself is written into a separate file which uses references to the constant table. The saving facilitated by the lookup table is significant: it generally reduces the size of the proof to one fourth of its original size.

This arrangement allows more than one proof to be stored in one proof file, which is very beneficial if the proofs are related: constants will then be shared. It is the responsibility of the user to structure her proofs into meaningful chunks. It wouldn't be hard to relieve the user from this responsibility and transparently structure proofs in a way which allows smallest proof files, but the implementation language of HOL98, which is Moscow ML is an interpreter and it is already having a hard time to cope with the massive memory requirements of the recorder. Keeping all the inference steps in memory would lay undue burden on the underlying bytecode interpreter.

In order to estimate the overhead of proof recording we used the same benchmark [Gor83] as in Wong's original paper. Direct comparison with earlier work is quite impossible since that was based on HOL88 and HOL90, furthermore those machines are not available to us, but Wong [WC97] reports the following given in Fig. 2.

	without recording (s)	with recording (s)	File size (MB)
HOL88	30	761	59.6
HOL90 (compact)	30	373	39.8
HOL90 (format 2.2)	30	349	12.1

Fig. 2. Summary of previous results

	without recording (s)	with recording (s)	File size (MB)
HOL98	12	450	75

Fig. 3. The overhead of proof recording

Comparison of the run-times in Fig. 2 with Fig. 3 shows an overall increase, which is most probably due to Moscow ML being an interpreter while HOL90 used SML/NJ which is well-known for it's heavy-weight optimisations. The overhead of running the proof with the recorder is a factor of 25, 12, 11 for Wong's earlier results and 30 for ours. The size of the resulting file also increased, which isn't surprising as XML is a fully parenthesised language: end-tags need to match start-tags. However, given the nature of the XML syntax, it is very simple to use general compression tools, such as gzip and so on, to reduce the size of the proof file before sending to the receiving prover. The compression ratio of 50 to 1 can easily be achieved as shown in previous work.

4.2 The Reader and Checker

We added an independent, in the sense that it is capable of reading any proof format and in more general any XML document, XML reader to Moscow ML. On the top of the XML reader we implemented a proof checker, which uses the built-in HOL rules. Of course, when the checker is used to check a proof generated by HOL98 itself the result is dubious, but checking proofs generated by other versions of HOL will increase the confidence in the implementation of the prover. Checking proofs from other provers is more interesting: it would be relatively simple to check Isabelle proofs, for the same logic, if proof recording were added to Isabelle.

The implementation of the reader builds on the publicly available validating XML parser, RXP [Tob99], written in C, which seems to be one of the fastest implementations around. The great advantage of using RXP is that one gets validation for free, as it happens transparently for the user.

A very simple interface between RXP and Moscow ML has been implemented. It allows one to call up RXP within Moscow ML and to return an abstract syntax tree representing the XML document.

The performance of the checker is quite similar to that of the reader. Surprisingly, most of the time is spent on validation and reading the big proof files, as opposed to actually checking well-formedness of the proofs.

5 Discussion and Future Work

Recent work by Necula and Lee [NL98] may point to a better representation of proofs. Unfortunately, their idea of stripping unnecessary parts of a proof is not in the spirit of proof checking

and linking of theorem provers. However, the reduction they achieve in proof size is such that it could result in an order of magnitude speedup both for recording and checking.

A converter between HOL proof and COQ has been developed. It takes proofs generated by HOL98 in the format described above, and converted them to a form that can be input into COQ. This is described in a separate paper[Den00]. This demonstrates that the extensible proof format in XML is useful in linking up theorem provers.

In summary, the work described in this paper shows that a framework for linking theorem provers can be built using an extensible proof format based on XML. At this stage, we can link up theorem provers that are based on similar logics. To link up provers based on different logics require much more work.

On the other hand, since we can store a proof in a structured document conforming to an open standard, we can take advantage of many useful tools to analyse and process the proof so that to extract useful information. For example, we may be able to analyse the proof and derive a human readable description of the proof.

References

[Coq00] The Coq Project. *The Coq Proof Assistant: Reference Manual*, ver. 6.3.1 edition, May 2000.

[Den00] Ewen Denney. A prototype proof translator from hol to coq. In *Proceedings of the 13th International Conference on Theorem Proving and Higher Order Logics*, August 2000.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL—a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[Gor83] Michael J C Gordon. LCS LSM, a system for specifying and verifying hardware. Technical Report 41, University of Cambridge Computer Laboratory, 1983.

[NL98] George Necula and Peter Lee. Efficient representation and validation of proofs. In *Proceddings of LICS'98*, 1998.

[SORSC99] N. Shanker, S. Owre, J.M. Rushby, and D. D. J. Stringer-Calvert. *PVS Prover Guide.* Computer Science Laborartory, SRI International, Menlo Park, CA, U.S.A., September 1999.

[Tob99] Richard Tobin. RXP — and XML parser available under the GPL. on the web with URL http://www.cogsci.ed.ac.uk/richar/rxp.html, 1999.

[W3C98] W3C. Extensible markup language (XML) 1.0, February 1998.

[WC97] Wai Wong and Paul Curzon. Towards an efficient proof recorder for hol90. In Elsa L. Gunter and Amy Felty, editors, *Supplementary Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics: TPHOLs'97*, pages 135 - 149. Bell Laboraries, 1997.

Embedding and Verification of an MDG-HDL Translator in HOL

Haiyan Xiong[1], Paul Curzon[1], Sofiène Tahar[2], and Ann Blandford[1]

[1] School of Computing Science, Middlesex University, London, UK
{h.xiong, p.curzon, a.blandford}@mdx.ac.uk
[2] ECE Department, Concordia University, Montreal, Canada.
tahar@ece.concordia.ca

Abstract. We investigate the verification of a translation phase of the Multiway Decision Graphs (MDG) verification system using the Higher Order Logic (HOL) theorem prover. In this paper, we deeply embed the semantics of a subset of the MDG-HDL language and its Table subset into HOL. We define a set of functions which translate this subset MDG-HDL language to its Table subset. A correctness theorem for this translator, which quantifies over its syntactic structure, has been proved. This theorem states that the semantics of the MDG-HDL program is equivalent to the semantics of its Table subset.

1 Introduction

The application of BDD (Binary Decision Diagram) [3] based tools in digital circuit synthesis and verification has been a breakthrough for the use of formal verification by industry. However, many questions remain about whether they work effectively or not. Ideally verification systems should themselves be formally verified using a verification system with a different architecture. Based on this consideration, we investigate the verification of aspects of the Multiway Decision Graphs (MDG) verification system [6] using the Higher Order Logic (HOL) theorem prover [9].

A variety of technologies have been used to ensure the correctness of verification systems. In a sense, which method is appropriate depends on the architecture of the verification system. The Edinburgh LCF (Logic of Computable Functions) [8] family of theorem provers (including HOL) uses an abstract data type (Thm) to represent theorems. The type checker ensures the theorems can be constructed only by applying a small number of primitive inference rules. There is no method to construct a theorem except by carrying out a proof based on the primitive inference rules and axioms. It effectively increases the reliability of the system. In this way if we guarantee the primitive inference rules correct then invalid theorems can be avoided. Moreover, the LCF approach permits proofs to be recorded. Proofs can be stored in files and be represented by lists of inferences. It allows us to make use of the availability of the sequence of inferences and to check the consistency of each inference automatically.

The architecture of a symbolic state enumeration based verification system is different. In this kind of system, higher level languages such as hardware description languages are used to describe the specifications and implementations. The specifications and implementations are then translated into decision diagrams. A series of algorithms in the system is used to efficiently and automatically deal with the decision diagrams and obtain the correctness results. The following two aspects of the system need to be verified:

1. the correctness of translation from the higher level languages into decision diagrams, and
2. the correctness of algorithms that are used to manipulate the decision diagrams.

In this paper, we prove the correctness of the translation phase of the MDG system. We need to verify that the semantics of a program is preserved in the semantics of its translated form. In this sense, it is a similar problem to that of compiler verification [4]. The contribution of this paper is to demonstrate how

compiler correctness work can be applied to a hardware verification system. In doing such a verification, we do more than just prove the correctness of the system, but also build a solid foundation to combine the HOL and MDG systems in a trusted way. Because we use a deep embedding semantics, the compiler correctness theorem can be combined with theorems converting MDG results into a form that can be easily reasoned about in HOL [16]. We thus obtain theorems that convert low level results actually proved in the core of the hardware verification systems (e.g., about decision graphs) to results about circuits in high level languages in a form that can be reasoned about in a theorem prover. We are thus able to import the MDG results into HOL based on a trusted MDG system.

The structure of this paper is as follows. In Section 2, we review related work. In Section 3, we overview the MDG verification system. In Section 4, we give the formal syntax and semantics of the subset of the MDG-HDL language we use. Here a set of functions for translating this subset language to their Table equivalent is given. Furthermore, the correctness theorem we have proved about the translation, which quantifies over its syntactic structure, is described. Finally, our conclusions and ideas for further work are presented in Section 5.

2 Related Work

There have been several previous projects concerned with the validation of results from verification systems.

Wong [15] developed a proof checker to examine the correctness of proof files–lists of inferences generated by the HOL system. The proof checker first took a proof file as an argument and then checked whether the proofs were correct or not. A log file was then produced that contained the hypotheses, lemmas used by the proof and the resulting theorem of the proof. Von Wright [13] formalised the specification of a proof checker in HOL. He also demonstrated how the HOL system could be used to formally verify the specification of a proof checker for higher-order logic proofs [14]. Another method of using refinement to verify the proof checker had been suggested by von Wright [12]. The proof checker also provided an independent means of ensuring the validity and consistency of proofs. Some other theorem provers such as Nqthm, Nuprl and Coq already store proof trees upon which a proof checker could work in the system. Boyer and Dowek [2] specified and implemented a proof checker in Nqthm logic.

Homeier and Martin [10] used the HOL system to verify a verification system called a verification condition generator (VCG) for a simple programming language. The proof of correctness of the VCG can be considered as an example of a compiler correctness problem, since the VCG translated the annotated programs to the lists of verification conditions. The semantics of the annotated programs and verification conditions were formalised in HOL. The correctness theorems showed that the truth of the verification conditions implied the truth of the annotated programs.

Chou and Peled [5] used the HOL system to verify a non-trivial algorithm–implementing a Partial-Order reduction technique, used in the protocol verification tool SPIN, which cuts down the state-space exploration performed by model checkers. They built up the groundwork of a formal infrastructure that included the mathematical support for proving various automatic verification algorithms. Their results not only gave more confidence in the algorithm but also demonstrated formal verification is a practical and useful tool.

In this paper, we verify the translation phase of the MDG system by using HOL. We need to verify that the compiler preserves the semantics of a program through the translation between languages as suggested for Homeier and Martin's work [10]. They used compiler verification methods to verify a software verification system. We use a similar method to verify a hardware verification system–the MDG system using HOL. In our study, we deeply embed a subset of the MDG-HDL language and its Table subset in HOL and verify the correctness of the translation between these two languages. Curzon et al. [7] did some basic work which verified the MDG components library in HOL. In their work, the semantics of the TABLE was first formalised in HOL. The TABLE construct is one of the basic hardware components used to define both behavioral specifications and structural specifications. Other components

such as logic gates can be defined in terms of it. They verified the Table implementations of each of the hardware components that were implemented in terms of tables in the MDG system. They used a shallow embedding semantics [1]–only the semantics is represented in the HOL logic not the syntax. Whilst this can be used to prove that each individual component implementation meets its specification, it cannot be used to give a general correctness theorem about the whole MDG-HDL language. We verify that the translation process is correct based on a deep embedding semantics [1] (i.e., we represent the abstract syntax of HDL programs by terms then define within the logic semantic functions that assign meanings to the programs). This allows us to prove a theorem that quantifies over the syntactic structure of the MDG-HDL language. That is we can prove "for all MDG-HDL programs, the semantics of a program is preserved in the semantics of its translated form".

3 The MDG System

MDG-HDL [17] is a Prolog-style hardware description language, which allows the use of abstract variables for representing data signals. In MDG, a circuit description file declares signals and their sort assignment, components network, outputs, initial values for sequential verification and the mapping between state variables and next state variables. In the components network, there is a large set of predefined components such as logic gates, flip-flops, registers, constants, etc. Among the predefined components there is a special component called a table which is used to describe a functional block in the implementation and specification. The table constructor is similar to a truth table but allows first-order terms in rows. It also allows the description of high-level constructs as ITE (If-Then-Else) formulas and CASE formulas.

Most of the components have their own tabular code and are compiled into their tabular code first. Tabular code can then be compiled into an internal MDG decision graph. Some components such as registers are implemented directly in terms of MDGs. However, in theory these components also could be implemented as tables. In this paper, we defined corresponding Tables for these components (register, fork, etc.). Since we are considering only a boolean subset of the language here, the Table representation of these components can be defined in terms of the corresponding input values (true or false). These definitions can be implemented in the MDG package. Non-boolean sorts could be handled by introducing an additional variable into the table. We assume the MDG-HDL program is firstly translated into a Table program and then the Table program is translated into MDG. In this situation, the MDG system could be specified as indicated in Figure 1:

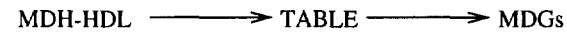

Fig. 1. Overview of the MDG Translation Phases

Adopting this approach makes the translation phase more amenable to verification. We are not verifying the actual MDG implementation. Rather our formalisation of the translator is a specification of it. Once combined with a translator from Tables to MDGs, it would be specifying the output required from the implementation. This would be used as the basis for verifying such an implementation. Effectively we split the problem of verifying the translator into the two problems of verifying that the implementation meets a functional specification, and that the functional specification then meets the requirement of preserving semantics. We are concerned with the latter step here. This split between implementation correctness and specification correctness was advocated by Chirica and Martin [4] with respect to compiler correctness.

4 MDG Translator Verification

The intention of our research is to explore a way of combining the MDG system and the HOL system in a trusted way as shown in Figure 2. This work can be divided into two steps. We first must verify the correctness of the MDG system using the HOL system (1) based on the semantics of the MDG input language. This part of the work consists of two phases–(1a) verification of the translator and (1b) verification of the algorithms. (2) We then must verify the HOL theorem generator which formalises the MDG verification results of different MDG applications and then converts them into the traditional HOL hardware verification theorems. All of these are based on the deep embedding of the semantics. By combining the separate correctness theorems from these two steps, we obtain a result that justifies the use of "theorems" imported from MDG [17].

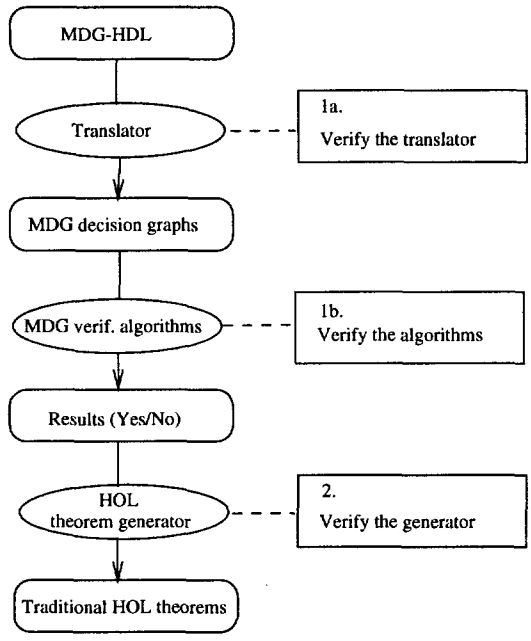

Fig. 2. Combining MDG and HOL in a Trusted Way

During this study, we considered a subset of the MDG-HDL language that did not contain two MDG predefined components (Multiplexer and Drivers) and nor do we consider the Transform construct used to apply functions. These components were omitted from our initial subset as they have non-boolean inputs or outputs. We also assume that the inputs and outputs of each component had Boolean sorts. We keep the subset simple here since we want to explore the feasibility of this method. However, we could extend our formalisation to accommodate different types as explained in [7]. As a result, the syntax of this language will be more complex. To distinguish between our subset of the MDG-HDL language and the Table subset, in the rest of this paper we will refer to the Table subset as the Table language.

In this paper, we concentrate on the verification of the translation phase of the MDG system (step (1a) from Figure 2) based on the semantics of the MDG input language using the HOL theorem prover. Step 2 is described elsewhere [16]. We first define the syntax and the semantics of the subset MDG-HDL and Table language. We then define a set of functions, which translate the program from the MDG-HDL language to the Table languages. For each component in MDG-HDL, a compilation operator is defined as a function, which returns its Table code. A translation function $TransGT$ is applied to each MDG-HDL program p so that the corresponding Table code is established. In other words,

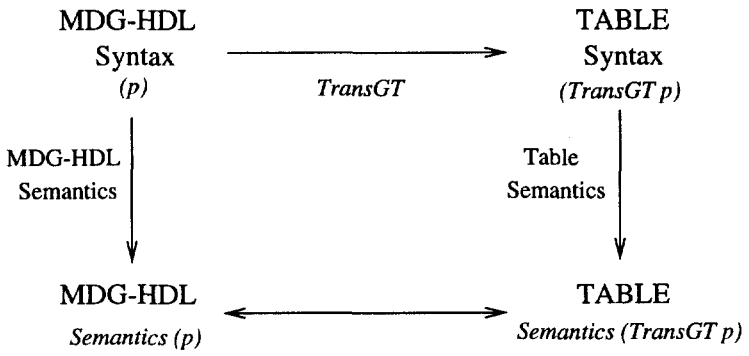

Fig. 3. Compilation Correctness

```
⊢ ∀ p. TransGT p = CorrespondingTablecode
```

The standard approach to proving a translator between two languages, is in terms of the semantics of the languages, as shown in Figure 3. Essentially the translation should preserve the semantics of the source language, which has the traditional form of compiler specification correctness used in the verification of a compiler [4]. The analogous method can be used to specify and verify the MDG system. For the translation to Table the correctness theorem has the form

```
⊢ ∀ p. Semantics (p) = Semantics (TransGT p)
```

4.1 MDG-HDL Syntax

In an MDG-HDL program, there is much information that is used in the MDG algorithm. When we write the syntax and semantics of programs, we can ignore this part of the information. Following the approach taken in other compiler correctness work, we abstract the useful information from the MDG-HDL program and work with an abstract syntax rather than the concrete syntax of the language. It would be straightforward to write a parser that translates the MDG-HDL into the form that we want.

For example, the MDG-HDL file of three *NOT* gates connected in series is given below.

```
...
signal(ip,bool).
signal(op,bool).
signal(u_B,bool).
signal(v_B,bool).
component(u_comp_B,not(input(ip),output(u_B))).
component(v_comp_B,not(input(u_B),output(v_B))).
component(op_comp_B,not(input(v_B),output(op))).
outputs([op]).
...
```

The abstract syntax of this file is

```
INTERNAL v_B (INTERNAL u_B (SEQ (NOT ip v_B
                                 (SEQ (NOT v_B u_B) (NOT u_B op)))))
```

where *INTERNAL*, *SEQ* and *NOT* are syntactic constructors of the subset of the MDG-HDL language. More details will be given later.

The full abstract syntax of the subset of the MDG-HDL language is given in Figure 4. The MDG-HDL commands consist of predefined MDG-HDL components, an operation to set the initial value

of a variable, a next state variable command, a composition operation and a localisation operation. The syntax of this language introduces a specially-defined recursive data type *mdg_hdl* to provide an explicit representation in logic of the MDG-HDL commands. We define a recursive type *mdg_hdl* with 35 constructors. The first 28 constructors are gates, flip-flops and registers. For example, the circuit term '*NOT ip op*' represents a *NOT* gate with one input labelled *ip* and one output labelled *op*.

The constructor *CONST1* declares a constant in a circuit. The constructor *FORK* represents the equality checker. The constructor *INIT* represents the initial value of a state variable. '*INIT(v,T)*' declares that the initial value of the variable v is true. The *SNXT* constructor maps between a state variable and a next state variable. '*SNXT v nv*' states that *nv* is the next state variable of the state variable v. The *SEQ* constructor represents the composition operation. If *c1* and *c2* are two values of type *mdg hdl*, then the term '*SEQ c1 c2*' represents the composition of the two terms represented by *c1* and *c2*. The *INTERNAL* constructor represents the localisation operation. If c is a term representing a circuit and x is a string (internal wire), then the circuit '*INTERNAL x c*' represents the circuit obtained by hiding the wire labelled x in the circuit represented by c.

The constructor *TABLESYN* represents the syntax of the MDG table component which has five arguments. The first argument is a list of inputs. The second argument is the single output. Its output could be either a current state variable or a next state variable. We define a new HOL type *out type* to represent these options:

```
out_type = NOWV of string|
           NEXTV of string
```

The third argument to a table is a list of table rows. Each row is a list itself, giving one allocation of values to the inputs. The entries in the list can be either actual values or a special don't care marker. This is realised by defining a new type (as given in [7]).

```
Table_Val = TABLE_VAL of α | DON'T_CARE

TableVal_to_Val (TABLE_VAL (v:α)) = v
```

The fourth argument is a list of output values that correspond to the values in input rows. The final argument is the default value, taken by the output if the input values do not match any row. The default value could be an arbitrary value, a current state variable or a next state variable. Again we define a new HOL type *default_type* in terms of the type *out type*.

```
default_type = DENORMAL of num->bool|
               DEOUT of out_type|
```

For example, the syntax of a *NOT* gate table is given below:

```
TABLESYN [ip] (NOWV op) [[TABLE VAL F];
                         [TABLE VAL T]]
                       [TSIG;FSIG] (DENORMAL ARB)
```

where "ARB" is the predefined HOL term representing an arbitrary value of a given type. The syntax of the MDG-HDL program can be any *mdg_hdl* term.

```
program = PROG of mdg_hdl
```

4.2 Table Syntax

The MDG Table language is a subset of the MDG-HDL language. It only consists of five of the constructors that we mention above– *INIT, SNXT, TABLESYN, SEQ* and *INTERNAL*. We do not define a new type for the MDG Table language. However, when we translate the MDG-HDL program into the MDG Table program, the Table program only consists of those five constructors. For example, the Table code of the three *NOT* gates is

```
out_type = NOWV of string |
           NEXTV of string

default_type = DENORMAL of num->bool |
               DEOUT of out_type |
               DECONST of string

Table_Val = TABLE_VAL of α | DON'T CARE

mdg_hdl = NOT of string =>string |
          AND of string=>string=>string |
          OR of string=>string=>string |
          NAND of string=>string=>string |
          XOR of string=>string=>string |
          NOR of string=>string=>string |
          AND3 of string=>string=>string=>string |
          OR3 of string=>string=>string=>string |
          NAND3 of string=>string=>string=>string |
          NOR3 of string=>string=>string=>string |
          AND4 of string=>string=>string=>string=>string |
          OR4 of string=>string=>string=>string=>string |
          NAND4 of string=>string=>string=>string=>string |
          NOR4 of string=>string=>string=>string=>string |
          AND5 of string=>string=>string=>string=>string=>string |
          OR5 of string=>string=>string=>string=>string=>string |
          NAND5 of string=>string=>string=>string=>string=>string |
          NOR5 of string=>string=>string=>string=>string=>string |
          AND6 of string=>string=>string=>string=>string=>string=>string |
          OR6 of string=>string=>string=>string=>string=>string=>string |
          NAND6 of string=>string=>string=>string=>string=>string=>string |
          NOR6 of string=>string=>string=>string=>string=>string=>string |
          JKFF of string=>string=>string |
          RSFF of string=>string=>string |
          JKFFE of string=>string=>string=>string |
          AO of string=>string=>string=>string=>string |
          REGCON of string=>string=>string |
          REG of string=>string |
          FORK of string=>string |
          CONST1 of bool=>string |
          INIT of (string#bool) |
          SNXT of string=>string |
          TABLESYN of (string list)=>out_type=>((bool Table_Val list) list)
                      =>((num->bool) list)=>default_type |
          SEQ of mdg_hdl=>mdg_hdl |
          INTERNAL of string => mdg hdl

program = PROG of mdg hdl
```

Fig. 4. The Syntax of the MDG-HDL Program

```
INTERNAL v_B (INTERNAL u_B
 SEQ (TABLESYN [ip] (NOWV u_B) [[TABLE_VAL F];
                                [TABLE_VAL T]]
                               [TSIG;FSIG] (DENORMAL ARB)
      SEQ (TABLESYN [u_B] (NOWV v_B) [[TABLE_VAL F];
                                      [TABLE VAL T]]
                                     [TSIG;FSIG] (DENORMAL ARB)
           TABLESYN [v_B] (NOWV op) [[TABLE_VAL F];
                                     [TABLE VAL T]]
                                    [TSIG;FSIG] (DENORMAL ARB))))
```

4.3 The Semantics of the MDG-HDL Program

We have defined the syntax of the MDG-HDL language. In this section, we will show how to define the semantics of an MDG-HDL program. First of all, the semantics of the MDG-HDL program is in terms of environment [11]. An environment is a function that has type :string $\rightarrow\delta$. This function maps a variable name (modeled by strings) to the value of that variable. In our language, the environment s is for state variables and signals. Its value is a history function and has a type :num$\rightarrow$bool that represents functions from time (natural number) to the value at that time.

We define a semantic function *SemMdghdl* for MDG-HDL programs. The first 28 components are mainly logic gates and flip-flops. Traditional hardware semantics can be given. The semantics of a component is then a relation between the input values and the output values. For example, the *NOT* gate can be expressed by

```
SEM_NOT ip op (s:string->num->bool) = ∀ t. (s op t) = ~(s ip t)

SemMdghdl (NOT ip op) s = SEM NOT ip op s
```

The semantics of *CONST1* represents a constant in a circuit which takes a constant *const* as its value. The output value does not change at any time.

```
SEM_CONST const op (s:string->num->bool) =
                        (∀ t. s op t = const)
```

The semantics of *FORK* represents the equality of two state variables. On each cycle, the output's value '*s op*' and input's value '*s ip*' are identical at that time.

```
SEM_FORK ip op (s:string->num->bool) = ∀ t. ((s op) t = (s ip) t)
```

The constructor *INIT* has two arguments. They are represented as a pair whose first component is a state variable and whose second component is a Boolean value. The semantics of *INIT* assigns an initial value (at time zero) to the value of the variable.

```
SEM_INIT (y:string#bool) (s:string->num->bool) =
                                (s (FST y)) 0 = SND y
```

The semantics of *SNXT* represents a relation between a state variable y and a next state variable *ny*. It declares that the next state variable of y is *ny*. In other words, the value of the variable y at the time t is equal to the value of the variable *ny* at the following time.

```
SEM_SNXT ny y (s:string->num->bool) = (∀ t. s ny (t+1) = s y t)
```

Sequencing is defined inductively in terms of the component commands. The semantics of *SEQ* is the conjunction of the corresponding semantics of each sub-command.

```
SemMdghdl (SEQ c1 c2) s =
                    ((SemMdghdl c1 s) ∧ (SemMdghdl c2 s))
```

The semantics of *INTERNAL* uses existential quantification to hide the local variable from the environment. It adds another entry to environment *s*. *s* is still the environment for the external wires. However, the extra entry for the new internal wire is first checked. This effectively hides the internal wires in circuit term *c*.

```
SemMdghdl (INTERNAL x c ) s =
     ∃ z. SemMdghdl c (λy.( if (y = x) then z else s y))
```

The semantics of *TABLESYN* follows the semantics of the table that was given by Curzon et al [7]. They firstly defined a predicate *Table_match* to check if the input values match the table values.

```
Table_match inputs [] t = T ∧
Table_match inputs (CONS v vs) t =
 (((HD (inputs) t) = TableVal_to_Val (v: α Table Val)) ∨
        (v = DON'T_CARE)) ∧
    (Table_match (TL inputs) vs t) )
```

The function table checks if there is a match on each row. If there is then the output has the corresponding value. Otherwise, the output equals the default value.

```
(table ip (op:num ->β) ([]: α Table_Val list) list) V_out default t =
         (op t = default t)) ∧
(table ip op (CONS v vs) V_out default t =
    ((Table_match ip v t) =>
        (op t = (HD V_out) t) |
        (table ip op vs (TL V_out) default t)))
```

The semantics of the *table* is

```
TABLE ip (op:num ->β) (V_outs:(α Table_Val list) list) V_out default =
       ∀ t. table ip op V_outs V out default t
```

The semantics of *TABLESYN* is defined in terms of the function *TABLE*

```
SemMdghdl (TABLESYN ip (op:out_type) y3 y4 y5)) s =
   TABLE (MAP s ip) (SEM_OUTVAR op s) y3 y4 (SEM_DEFAULTVAR y5 s)
```

For example, the semantics of the Table code of the *NOT* gate is

```
SemMdghdl (TABLESYN [ip] (NOWV op) [[TABLE_VAL F];[TABLE_VAL T]]
           [TSIG;FSIG] (DENORMAL ARB)) s =
              TABLE (MAP s [ip]) (SEM_OUTVAR (NOWV op) s)
                    [[TABLE_VAL F];[TABLE_VAL T]]
                    [TSIG;FSIG] (SEM_DEFAULTVAR (DENORMAL ARB) s)
```

Finally, the semantics of a whole MDG-HDL program is expressed as a function *SemMdghdl* inside the logic:

```
(SemMdghdl (NOT ip op) s = SEM_NOT ip op s) ∧
......
(SemMdghdl (FORK ip op) s = SEM_FORK ip op s) ∧
(SemMdghdl (TABLESYN ip op y3 y4 y5) s =
          TABLE (MAP s ip) (SEM_OUTVAR op s) y3 y4
                    (SEM_DEFAULTVAR y5 s)) ∧
(SemMdghdl (SEQ code1 code2) s =
        ((SemMdghdl code1 s) ∧ (SemMdghdl code2 s))) ∧
(SemMdghdl (INTERNAL x code) s =
         ∃ z. SemMdghdl code (λy. (if (y = x) then z else s y)))
```

4.4 Compiling MDG-HDL into the Table Language

The first step in specifying a compiler for MDG-HDL is to define a set of functions for compiling the MDG-HDL program into the Table language. For each component in MDG-HDL, a compilation operator is defined as a set of functions that return its table code. For example, a *NOT* gate is compiled into

```
TRANS_NOT (ip:string) op =
       TABLESYN [ip] (NOWV op) [[TABLE VAL F];
                                [TABLE_VAL T]]
                               [TSIG;FSIG] (DENORMAL ARB)
```

For the MDG-HDL program, we define a function *TransGT* inductively over the syntactic structure and this function translates the MDG-HDL program into the equivalent Table language.

```
(TransGT (NOT ip op) = TRANS_NOT ip op) ∧
......
(TransGT (SEQ (code1:mdg_hdl) code2) =
             SEQ (TransGT code1) (TransGT code2)) ∧
(TransGT (INTERNAL x code) = INTERNAL x (TransGT code))
```

For example, the following theorem obtained by rewriting with the definitions illustrates the translation of the MDG-HDL program of three *NOT* gates discussed above

```
⊢ TransGT (INTERNAL v_B (INTERNAL u_B
   (SEQ (NOT ip v_B (SEQ (NOT v_B u_B) (NOT u B op)))) =
     INTERNAL v_B (INTERNAL u_B
       SEQ (TABLESYN [ip] (NOWV u_B) [[TABLE_VAL F];
                                      [TABLE_VAL T]]
                                     [TSIG;FSIG] (DENORMAL ARB)
            SEQ (TABLESYN [u_B] (NOWV v_B) [[TABLE VAL F];
                                            [TABLE VAL T]]
                                           [TSIG;FSIG] (DENORMAL ARB)
                 TABLESYN [v_B] (NOWV op) [[TABLE VAL F];
                                           [TABLE VAL T]]
                                          [TSIG;FSIG] (DENORMAL ARB))))
```

4.5 Compiler Correctness Theorem

To verify the correctness of a translator as we suggested in the beginning of this section, we have to obtain a theorem that quantifies over its syntactic structure stating that the semantics of the MDG-HDL program is equivalent to the semantics of the Table program used in MDG implementation. For our subset language, we have proved a theorem by using HOL:

```
⊢ ∀  p. SemMdghdl p s = SemMdghdl (TransGT p) s
```

where p represents any MDG-HDL program and *TransGT* is the function defined earlier which translates the MDG-HDL program to its Table code. s is the environment disscussed earlier for variables, respectively. The correctness theorem is proved by structural induction on the syntax domain of the MDG-HDL program.

5 Conclusions and Further work

In this paper, we prove the correctness of the translation phase of a decision diagram system (the MDG system) using a theorem proving system (the HOL system). We have defined the syntax of a subset of the MDG-HDL language and the Table language in higher-order logic. The semantic function is defined

by structural induction over their syntactic structure. A set of functions that translate the syntax of an MDG-HDL program to the syntax of the Table language has been defined. The correctness theorem, which quantifies over its syntactic structure, has been verified. This theorem states that the semantics of the original MDG-HDL program is equivalent to the semantics of the Table program used in the MDG implementation.

Our motivation for deep embedding the MDG-HDL and Table languages into HOL is not only to verify aspects of correctness of the MDG system, but also to make use of the semantics to formally import the MDG results into HOL based on a trusted MDG system (Figure 2). We have formally imported the correctness results produced by four different hardware verification applications into HOL [16]. We have in each case proved a theorem that translates them into a form usable in a traditional HOL hardware verification, i.e., that the structural specification implements the behavioral specification. The applications considered include combinational verification, sequential verification and invariant checking. Therefore, we can obtain theorems that justify the conversion of low level results proved in the MDG system to results about circuits in high level languages in a form that can be reasoned about in the HOL system.

The work presented in this paper is part of a larger project to verify a combined HOL-MDG system. We need to prove that the translation from the tabular code to MDG decision graphs is correct. We also need to prove that the MDG algorithms are correct. We need to extend the subset considered to deal with, for example, sort declaration for the verified system to be applicable to real designs.

Acknowledgments

We are grateful to Dr. Richard Boulton at University of Glasgow and Dr. Skander Kort at Concordia University for their help. This work is funded by EPSRC grant GR/M45221, and a studentship from the School of Computing Science, Middlesex University. Travel funding was provided by the British Council, Canada.

References

1. R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van-Tassel. Experience with embedding hardware description language in HOL. In T. F. Melham and R. T. Boute, editors, *Theorem Provers in Circuit Design*, pages 129–156. North-Holland, 1992.
2. R. S. Boyer and G. Dowek. Towards checking proof checkers. In *Workshop on Types for Proofs and Programs (Type'93)*, 1993.
3. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions in Computers*, 35(8):677–691, August 1986.
4. L. M. Chirica and D. F. Martin. Toward compiler implementation correctness proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, April 1986.
5. C. T. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes in Computer Science, pages 241–257, 1996.
6. F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, 10(1):7–46, 1997.
7. P. Curzon, S. Tahar, and O. Aït-Mohamed. Verification of the MDG components library in HOL. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher-Order Logics: Emerging Trends*, pages 31–46. Department of Computer Science, The Australian National University, 1998.
8. M. J. Gordon, R. Milner, and C. P. Wadsworth. Edinburgh LCF: A mechanised logic of computation. Number 78 in Lecture Notes in Computer Science, 1979.
9. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-order Logic*. Cambridge University Press, 1993.
10. P. V. Homeier and D. F. Martin. A verified verification condition generator. *The Computer Journal*, 38(2):131–141, July 1995.

11. T. F. Melham. *Higher Order Logic and Hardware Verification.* Cambridge Tracts in Theoretical Computer Science 31. Cambridge University Press, 1993.
12. J. von Wright. Program refinement by theorem prover. In *Proc. 6th Refinement Workshop*, London, January 1994. Springer–Verlag.
13. J. von Wright. Representing higher-order logic proofs in HOL. *The Computer Journal*, 38(2):171–179, July 1995.
14. J. von Wright. The formal verification of a proof checker. SRI internal report, November 1998.
15. W. Wong. Validation of HOL proofs by proof checking. *Formal Methods in System Design*, 14(2):193–212, March 1999.
16. H. Xiong, P. Curzon, and S. Tahar. Importing MDG verification results into HOL. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics*, number 1690 in Lecture Notes in Computer Science, pages 293–310. Springer-Verlag, September 1999.
17. Z. Zhou and N. Boulerice. *MDG Tools (V1.0) User Manual.* University of Montreal, Dept. D'IRO, 1996.