Software Design for Reliability and Reuse
Preliminary Method Definition

R. B. Kieburtz

October 12, 1993

This document describes the SDRR method for designing a software component generator. This is the view seen by a team of computer scientists responsible for producing a design for a new class of applications. The view of a domain expert who will use the generator to create a specific software component is far simpler. A generated software component will typically be incorporated in an instance of a domain specific-architecture.

1 The SDRR design team

An SDRR design team includes a number of specialists. The specialists need to coordinate their efforts but are able to work independently and concurrently on their assigned tasks. Although the roles are described as those of individuals, a single person might play several roles on a small project, or a single role might be shared by several persons on a large one. The members of the team are:

- The design manager, who is a *domain expert* and has overall responsibility for the software design, its validation and documentation.
- A domain-specific language designer, who should have some knowledge of the applications domain but who primarily knows the principles of language design.
- The semantics expert, who understands formal semantics of programming languages and is expert at programming in the algebraic design language, ADL.

- The verification expert, trained in mathematical logic, who uses formal proof techniques to verify critical properties of a design.
- The implementation designer, who is expert in use of the implementation language and in choosing efficient data representations.
- The transformations expert, who understands strategies for algorithm improvement.

2 The SDRR design concept

SDRR is a method for the design of flexible and powerful software component generators. With a component generator, software components themselves are not the basis for design reuse—the design encapsulated in the generator is the reusable artifact. When a subsequent application or version of a design is needed, design modifications are made to the specification that is input to the generator, and a new software component is generated automatically. This allows each design change to be made at a level of abstraction at which details of the software irrelevant to the change are not seen. It also allows the applications designer and the software maintainer to use a design language that is expressive of the domain of application, rather than to encode the design in a wide-spectrum programming language.

Component generators achieve their greatest advantage for the design of families of software modules that are needed in many particular instances. If a software component is anticipated to be a one-off instance, dissimilar to any existing design, to be used once and never modified, then developing a component generator to produce it has little advantage. However, this is almost never the case. Most software modules have family resemblances to other, related modules and will undergo use and modification over an extended life cycle that requires their design to be maintained and updated. SDRR is intended to produce software components that can be reliably and inexpensively maintained over an extended life cycle.

2.1 Steps in the design of a software component generator

The design of a component generator by SDRR proceeds in a series of steps that are carried out by the specialists of the design team, in cooperation with one another.

1. Perform domain analysis to determine the requirements of the intended application. This step is common to all software development. It is not unique to SDRR. It needs the attention of a domain expert.

2. Formulate a domain-specific design language (DSDL) in which to express the parameters, operations and constraints necessary to meet the requirements of the domain application. This is a task for the domain-specific language designer, with the collaboration of a domain expert.

3. Formalize a computational semantics of the DSDL in terms of ADL, the algebraic design language used in SDRR. This task requires the semantics expert, a computer scientist with advanced training in formal aspects of programming languages and software design.

4. Prove critical properties of the formal semantics. This is the task of the verification expert. This task does not imply the need for a comprehensive "proof of correctness" of every aspect of the DSDL semantics, but it does offer an opportunity for formal verification of those properties of a design that are deemed to be of critical importance. Some properties can be verified independently of one another, or "incrementally". Verification may be considered to be an off-line task, in that progress towards building an implementation does not have to await its completion.

5. Design an implementation. SDRR implementations are stereotyped. An implementation expert designs a set of implementation primitives that specify how the computational semantics of the DSDL are to be realized in terms of a target programming language. For DoD applications, the target programming language will ordinarily be Ada. An implementation design is usually retrieved from a library, rather than designed for a specific application. An implementation design is tailored to an application by an *environment specification* that is also provided by the implementation designer. The environment specification details the interfaces to target language libraries and to other software or hardware modules present in a system architecture.

6. Formulate tactics for performance improvement by program transformation. This is the task of the transformation expert. Performance improvement is obtained through the use of automated program transformations that are applied during the course of program generation. These transformations are mathematically based and are guaranteed to preserve the computational meaning of the ADL-specified semantics. The transformation expert designs a control scheme, or tactic, for application of these transformations to ensure their effectiveness. When an SDRR-designed program generator is applied to a DSDL specification, it automatically applies the necessary transformations by following the prescribed tactic.

3 Domain Specific Design Languages — DSDL's

A domain specific design language is intended to formally specify software designs. It is a formal language that is expressive over the abstractions of an application domain. A DSDL may be wholly or partially declarative or it may be a functional language with libraries of functions specialized to the application domain. Common examples of DSDLs are:

- Tex and Latex, text formatting languages.
- Mathematica, an extensible language for mathematical modeling.
- Schema description and query languages for databases.
- Layout languages for prettyprinting the text of computer programs.
- A message format description language for the message domain of military C^3 systems.

An advantage of using a DSDL is that a domain expert can express domain-specific concepts directly, rather than encoding them. This allows the domain expert to formalize the specification of a software solution immediately instead of communicating a specification informally to a software specialist who may be less familiar with the intended application.

3.1 Designing a DSDL

A DSDL is defined by a computer scientist in consultation with a domain expert. In the design of a DSDL, a dialogue is necessary between the two in order to settle three important issues:

- To clearly identify the principal conceptual abstractions of the domain. For example, in a language for formatting mathematics, the essential abstractions might include expressions, fractions, vectors, matrices, etc.

- To formally define a language of terms to represent these abstract concepts. A term language can be defined in terms of a syntactic phylum (syntactic category) for each conceptual entity. The formal definition of a syntactic phylum is done through the use of a context-free grammar. Each abstract grammar generates a context-free language. These languages give a means to express instances of the concepts and of relations among them.

- To interpret the relations among the principal conceptual entities. This interpretation is initially given by the domain expert in an informal manner, by describing the relations in natural language (English). The computational content of this description will later be elaborated by giving a formal semantics to the DSDL. For example, if the DSDL were a language for formatting mathematics, the relations among entities might consist of rules or constraints that govern the two-dimensional layout of their presentation within a rectangular window.

When designing a DSDL two important criteria should be kept in mind: (1) the DSDL must be intelligible to a domain expert, and (2) the formal semantics must allow a specification expressed in the DSDL to be translated into effective procedures that realize the specification. Typically, such a specification will provide static or dynamic constraints on an artifact of the application domain, or will specify its dynamic behavior.

Often, a graphical user interface (GUI) can be used to advantage to help an application designer to formulate an application design in the DSDL. With a well-designed GUI, the application designer does not need to "learn another language" in order to use the DSDL. The GUI takes the place of the "surface syntax" of the DSDL, providing instead the visual guidance of highlighting, windows and menus to guide the application designer to the desired structure of a formal specification. In the SDRR method, a GUI has very limited responsibility for checking data validity. It might, for instance, enforce a restriction on the number of characters in a fixed-length field of text but it would not be responsible for checking that words entered into such a field were valid or were spelled correctly.

3.2 Tool support for a DSDL

A DSDL has the syntactic structure of a context-free language. If it is given a surface syntax, then a parser generator tool such as *yacc* can be used to construct a translator from the surface syntax to its abstract syntax. If it is given a graphical interface, then a standard GUI design tool such as *Motif* can be used. In either case, the structure of a design specification is determined by the abstract syntax of the DSDL.

4 Formalizing the semantics of a DSDL

The formal semantics of a DSDL is defined in terms of an algebraic design language (ADL). This semantics gives the DSDL a computational interpretation in which the relations between the principal concepts of a design abstraction are formalized.

The first step in the formal specification of a semantics is to specify a datatype that corresponds to the abstract syntax of the DSDL. To each operator of the abstract syntax there will correspond a data constructor of the datatype. The semantics of a term constructed with a given data constructor will be composed from the semantics of the subterms given as arguments to the data constructor.

The control structure of an ADL program is specified through families of high level combinators. To each combinator there corresponds both a computation rule and a proof rule. The computation rules give an operational semantics to the ADL language and the proof rules give it a logical interpretation consistent with the computational one. Instances of the combinators are composed to form more complex function definitions in ADL. The laws obeyed by such functions are inferred by applying the proof rules of each constituent combinator.

Each operator of the abstract syntax of a DSDL is given a computational interpretation by a semantic function. The semantic function is well-typed in the type system of ADL, and is defined by cases on the data constructors of the ADL datatype derived from the abstract syntax of the DSDL. For each such case, the prescribed meaning of a DSDL fragment is specified by a computation programmed in ADL.

This programming technique uses the syntax of the DSDL to structure the specification of a computational solution. The resulting solution is compositional. Less attention is given initially to the efficiency of a solution than to the uniformity of its construction from its component parts. The goal is to specify a computation in such a way that it is amenable to formal reasoning, so that one can verify that it corresponds to the informally specified problem requirements. Algorithmic efficiency will be improved at a later stage by meaning-preserving program transformation of an ADL specification and by compilation into an efficient representation in an implementation language.

5 Verification of semantic properties

As noted previously, to every ADL combinator there corresponds an inductive (or coinductive) proof rule. The structure of the rule is dictated by the inductive definition of the particular datatype for which the combinator defines homomorphisms. Since proofs are constructed from induction schemes that correspond directly to the combinators, it is possible to derive from a combinator and a proposition for which a proof is sought, the verification conditions that must be discharged to complete the proof. By automating the derivation of proof obligations, we obtain a goal-driven proof assistant for ADL.

Certain combinators require termination conditions to be proved. Termination proofs require the specification of a domain predicate, which becomes a verification condition for the application of a combinator, and a well-founded ordering on the domain of the combinator, as restricted by the domain predicate. Termination proofs are independent of the proof of other properties of a combinator.

The construction of proofs affords opportunity for human error, just as does the specification and design of programs. Verification by proof adds reliability not only because it involves formal reasoning, but because this reasoning can be checked by a mechanical proof assistant. A proof assistant for ADL is an important (but as yet unimplemented) adjunct to the set of design support tools.

6 Transformational Improvement

When the semantics of a DSDL is fully elaborated in ADL it is algorithmically effective. A component design specified in the DSDL can be executed as a rapidly constructed prototype. However, without further work, it is likely to have poor performance in terms of execution time and space usage. The SDRR method encourages highly modular design of semantic functions in ADL. This produces a design that is easy to understand, to validate and to maintain, but engenders many more uses of function composition than might otherwise be necessary. Accordingly, control structures that might be shared are often duplicated, and intermediate data structures may be built and analyzed when they could have been avoided by careful programming.

To avoid paying performance penalties for modular design, SDRR employs extensive program transformation on the ADL specification. The transformations that are used are *meaning preserving*, which implies that they will never introduce errors that were not present in the original design. These transformations are, in fact, derived as instances of theorems in the algebra of ADL. There are transformations that support

- deforestation—elimination of intermediate data structures;
- fusion—consolidation of similar control structures;
- accumulator introduction—caching of values to avoid recomputation;
- recursion elimination, in favor of iterative control;
- introduction of state.

Transformations of an SDRR design are applied automatically. Interactive direction of transformation steps has proven difficult to do effectively. In applying the SDRR method, a human transformation expert supplies a tactic to control the automatic application of transformation steps. Transformations are directed by pattern-matching which triggers the invocation of embedded tactics.

7 Implementation Templates

An implementation is specified by a set of *implementation templates* and an interface specification. An interface specification documents the (typed) system interface that will be seen by the software component that is the object of the design. The functionality required of the interface can be specified informally or in terms of a first-order logic or software specification language.

The interface provided by the designed component includes the (typed) signature of its visible functions or procedures, together with the formal specification of the component as elaborated in the design.

Implementation templates are macro-like translation forms for the primitive access and construction functions of ADL datatypes. Implementation templates can be provided in a number of different implementation languages, although for DoD projects, Ada will be the preferred language for implementation. A set of implementation templates must contain generic templates for algebraic datatypes but it may also contain specialized templates for specific types that are commonly used. Through implementation templates, a designer can specify a hashed symbol table, for instance, as the implementation of an association list.

Implementation templates are typically quite small, of the order of a few hundred source lines, although a set of templates can grow if additional, specialized implementations are specified for particular datatypes. These templates are highly reusable, both because templates are copied many times during the translation of a single design from ADL to the target implementation language, and because a set of templates can be used in any number of specific applications.

8 Design reviews

SDRR design reviews are conducted by all the members of the group involved in a design activity. A review consists in critiquing different parts of the design to check for inconsistencies. The critical aspects to be reviewed are the DSDL design, the semantics of the DSDL as given in ADL, the specified implementation templates, and confirmation that the environment specifications have been met.

Reviewing a DSDL is critical since its specification is initially given with an informal English description. Therefore, there is the possibility that it may contain ambiguities and misunderstandings. Allowing several people to study it often leads to a better definition of the language.

Prior to reviewing the semantics of the DSDL given in ADL, preliminary validation is obtained by type-checking the semantics. This provides early notification of all typing errors. For an informal validation of the semantics, the reviewers read and discuss the semantics. Formal verification is carried out by constructing proofs, using the proof rules that accompany each combinator. For reliability, proofs should be machine checked.

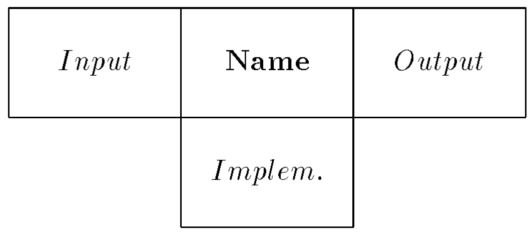
The design review committee also checks that the implementation templates meet all design constraints. Formal validation of the templates is done mechanically through exhaustive testing of each implementation function. After a set of implementation templates has been validated, it can be archived in a library of valid implementations.

Finally, it is also necessary to check that the environment specifications have been met. This is a necessary step, even though integration testing will be performed, since exhaustive integration testing is very hard.

Design review of transformational improvements is only needed to determine the effectiveness of the transformation tactics. It is not necessary to review the correctness of the semantics transformations, since it is proved that none of the transformations change the meaning of the ADL forms, but affect only the performance of an implementation.

9 Tool support for SDRR

This section summarizes the design support tools that underlie SDRR. The tools can be envisioned in terms of 'T' diagrams, such as



in which

Name is the name of the tool;

Input is the language of its input;

Output is the language of its output;

Implem. is the language in which the tool is implemented.

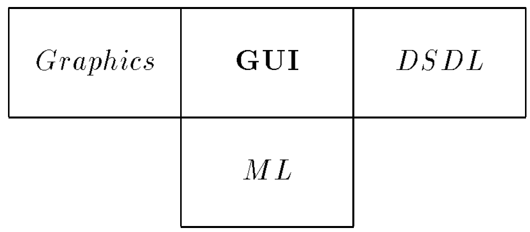
When the output language of tool A matches the input language of tool B, these tools can communicate directly with one another. Otherwise, a translation of representation is necessary to compose two tools.

The 'T' diagrams shown below for the tools *Schism* and *Astre* have been extended to show the interface modules that translate data representations. These interfaces allow these existing transformation tools to be seamlessly plugged into a composite sequence of SDRR transformation tools.

The tools are grouped into three sections, (1) those used to implement a DSDL, (2) those used to transform the ADL semantics of a DSDL into a simpler and more efficient program, and (3) those that translate the simplified ADL semantic representation into a program in the target implementation language. Tool group (2) will be standardized for the SDRR method and automatically invoked on each particular domain design. For each tool, three levels of capability are listed. These are to be achieved in the three stages of prototyping in the "evolving prototype" life cycle.

9.1 Design capture in a DSDL

9.1.1 The graphical user interface

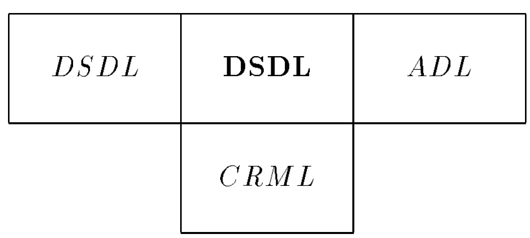


Purpose: To provide an interface for the design of applications in the prescribed domain.

Capabilities of scheduled prototypes:

1. Data entry—GUI embeds data into the phrase structure of the DSDL
2. Data editing—GUI supports editing of a specification.
3. GUI performs some data checking and error recovery.

9.1.2 The DSDL Compiler



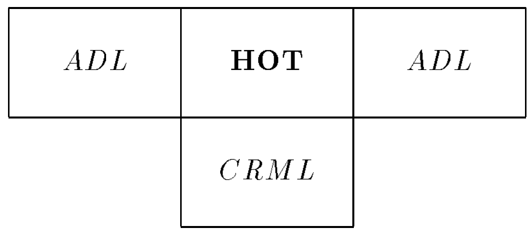
Purpose: To translate a domain-specific design specification into ADL.

Capabilities of scheduled prototypes:

1. DSDL is translated into core ADL, without monads.
2. Monad interpretations are added to the ADL formulation.
3. An ADL module capability is added.

9.2 Formal transformations for algorithm improvement

9.2.1 Higher Order Transformations

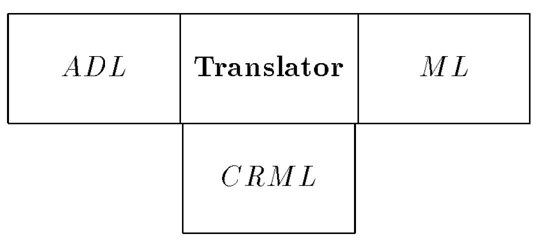


Purpose: To improve algorithmic efficiency by applying the algebra of ADL to rewrite combinator expressions.

Capabilities of scheduled prototypes:

1. Perform fusion for those ADL combinators based on initial algebras.
2. Extend techniques to non-initial algebras.
3. Extend techniques to bifunctors and combinators of coalgebras.

9.2.2 ADL Translator

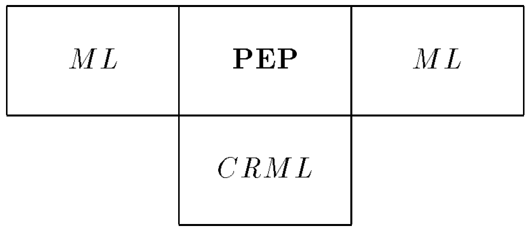


Purpose: Transform ADL programs into ML programs.

Capabilities of scheduled prototypes:

1. Restricted to "core" ADL without termination proofs.
2. Generate proof obligations necessary for termination.
3. Extend transformations to the ADL module system.

9.2.3 Partial evaluation preprocessor



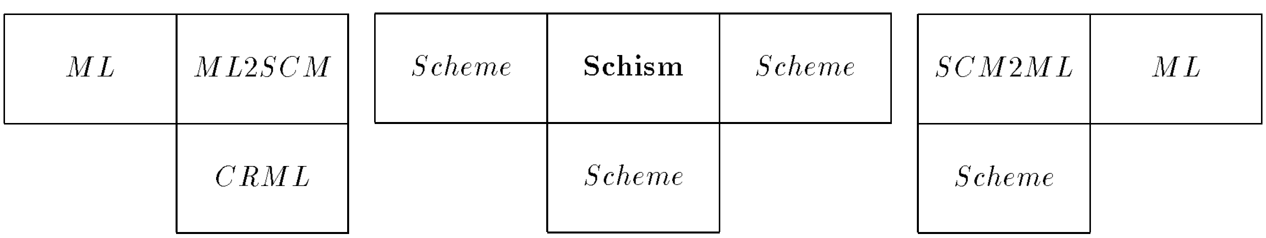
Purpose: Prepare ML programs for partial evaluation by:

- Replacing every function of multiple, individual arguments by an equivalent function of a single, tupled argument.
- Lifting all function definitions to top level.

Capabilities of scheduled prototypes:

1. Restricted to uncurrying functions in "core" ML.
2. Add lifting of function definitions to top level.
3. Extend to the ML module system.

9.2.4 Partial Evaluation

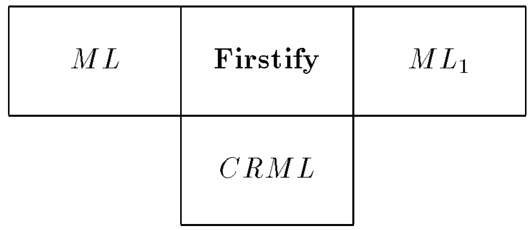


Purpose: Symbolically evaluate programs at compile time.

Capabilities of scheduled prototypes:

1. Provide translation tools between Scheme and ML.
2. Automatically generate the fold-unfold heuristics (via ADL).
3. Provide more extensive feedback to the user.

9.2.5 Firstify

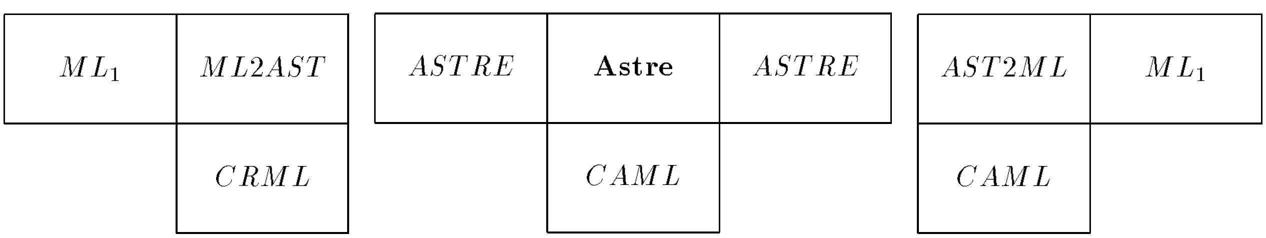


Purpose: Transform functional values into data structure representations so that the program can be transformed directly into an imperative language format.

Capabilities of scheduled prototypes:

1. Accepts only a list of ML declarations.
2. Extended to "core" ML.
3. Extended to the ML module system.

9.2.6 Astre



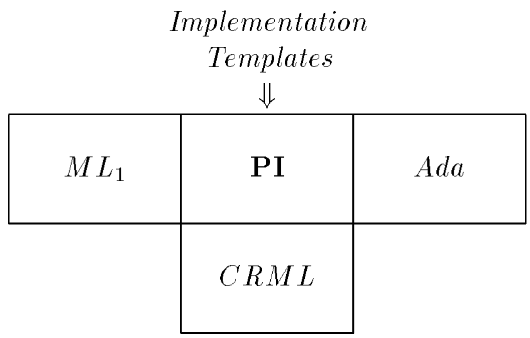
Purpose: Perform optimizations on first order programs.

Capabilities of scheduled prototypes:

1. Provide interactively-controlled program transformation.
2. Provide attributes inherited from the ADL representation (and supplied interactively at prototype level 1).
3. Fully automatic control of transformation tactics.

9.3 Translation to imperative language for implementation

9.3.1 Program Instantiation



Purpose: Translate first order ML programs into Ada.

Capabilities of scheduled prototypes:

1. Generate single-assignment Ada functions for ML expressions.

2. Generate imperative Ada using state variables and exceptions.

3. Generate Ada packages.

10 SDRR illustrated with a top-to-bottom example

Here will illustrate how the SDRR method is applied with a simple, yet instructive example. The application is a recognizer for strings of symbols that belong to a regular set. It is known that such a recognizer can be constructed on the model of a finite state automaton. The automaton makes a state transition on each successive symbol of an input string until either it determines that the string cannot be a member of the specified set or it has scanned the entire string and finds that it is a member of the set. A very simple example of such an automaton is one that would keep track of the parity of a string of binary digits and accept just those strings of even parity.

It is important to keep in mind is that such an automaton is not an intelligent being. Its capacity for memory is bounded—it has only a finite number of states. Thus no finite state automaton can recognize all those strings (over an alphabet of more than one symbol) that are palindromes, because any finite automaton could be presented with a palindrome that was long enough that the states of the automaton could not encode enough information from the first half of the string to determine whether the second half matched it in reverse.

Regular sets have been extensively studied and there exists a formal, mathematical language in which to describe any regular set. This language, called *regular expressions*, is an example of a domain-specific design language for the problem domain of recognizing regular sets. This problem is not entirely academic. Applications that rely on recognition of regular expressions include the lexical analysis phase of compilers, many communications protocols, and substring matching algorithms such as `grep` in UNIX systems, the `*.*` command of DOS, etc.

10.1 Domain analysis

A regular expression (RE) describes a set of strings. The syntax of RE's is expressed by an context-free grammar:

Syntax:

RE	$::=$	*symbol*	explicit symbol
	$\mid$	$RE \circ RE$	catenation of strings
	$\mid$	$RE + RE$	union of regular sets
	$\mid$	RE^*	zero or more repetitions
	$\mid$	(RE)	
	$\mid$	ϵ	the null string
	$\mid$	$\emptyset$	the empty set

10.2 Semantics of regular expressions

Every regular expression has an equivalent nondeterministic finite automaton (NFA) that will accept the set of strings specified by the RE. A computational realization that constructs an NFA from a given RE provides a semantics for the RE language.

Such a realization has been programmed as an ADL function specification. This function, called *translate_RE*, is specified by straightforward analysis-by-cases on the syntactic structure of an RE. The domain of this function is a datatype in which an NFA is represented by (1) the number of states that constitute it, (2) a list of triples that represents its state-transition relation, (3) a list of its initial states, (4) a list of its final states.

The final component of a string recognizer is a fixed function, *interpret_NFA*, specified in ADL. This function interprets the data structure produced by translating an RE, acting like the prescribed NFA to accept or reject a string that is presented to it as an argument.

10.3 Verifying the correctness of the realization

The fundamental result that verifies the semantics given to the RE language is a well-known theorem that asserts the equivalence of an NFA to the RE from which it is obtained. Informal reasoning has been used to verify that the two ADL functions, *translate_RE* and *interpret_NFA*, are consistent with the mathematical model.

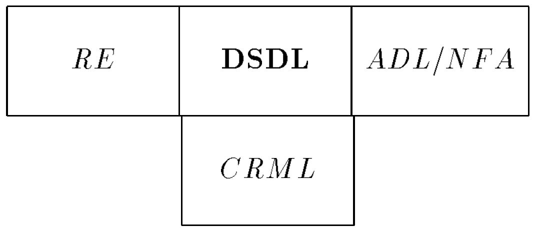
10.4 Using a pattern recognizer generator

- The user inputs a regular expression representing a set of strings for which a recognizer is wanted;
- *translate_RE* (the DSDL compiler) translates the RE a data structure representing the equivalent NFA;
- The NFA and its interpreter pass through the design automation system;
- The final product is a program that reads a string and accepts it if and only if it belongs to the set of strings described by the RE.

10.5 Tool support applied to the example

We continue the example by showing how each tool is applied to it.

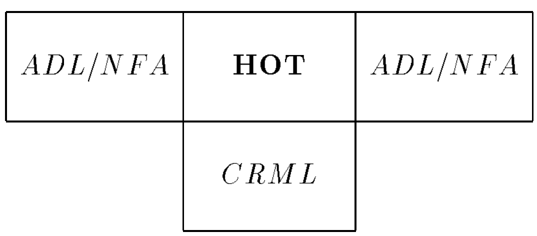
10.5.1 Using the DSDL Compiler



Purpose: To compile the NFA specified by a regular expression into ADL.

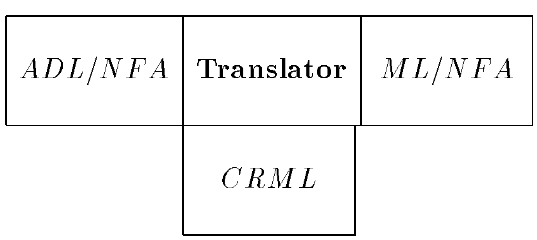
- Translate regular expressions into ADL data structures representing NFAs.
- An NFA interpreter takes an NFA specification and a string and simulates the actions of the NFA on the string.

10.5.2 Applying higher order transformations



Purpose: To improve the string recognition algorithm by applying the algebra of combinators to rewrite the compiled ADL representation of an NFA interpreter.

10.5.3 Applying the ADL Translator

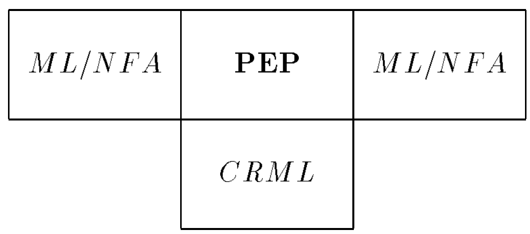


Purpose: To transform an ADL program into an ML program.

1. An NFA is translated into an ML data structure.

2. The ADL representation of an NFA interpreter is translated into an ML program that is equivalent in that it gives the ML data structure representation of an NFA the same interpretation as the ADL representation.

10.5.4 Applying the Partial Evaluation Preprocessor

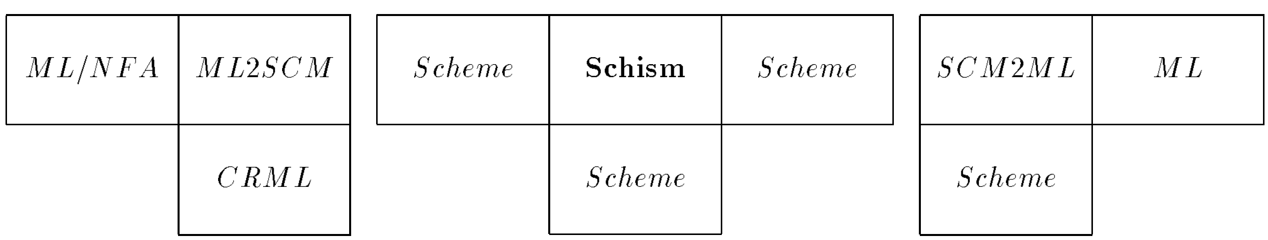


Purpose: Prepares ML programs for partial evaluation by:

- uncurrying functions in the interpreter;

- lifting all locally defined functions in the interpreter to top level.

10.5.5 Partial Evaluation

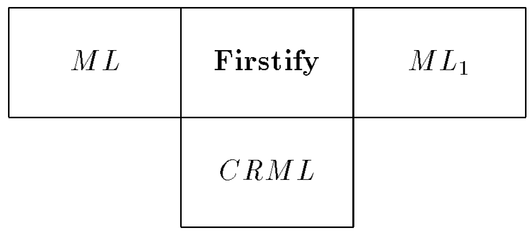


Purpose:

1. The interpreter is partially evaluated with respect to the particular NFA.

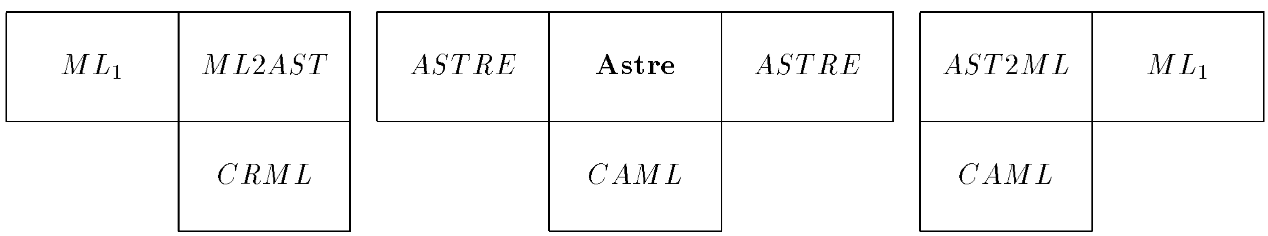
2. The result is a residual program that takes only a string as an argument.

10.5.6 Firstify



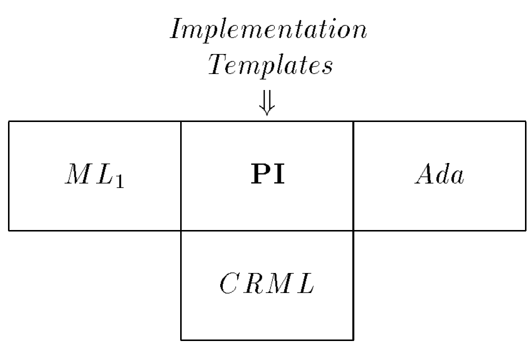
Purpose: Transform functional values into data structure representations so that the program can be transformed directly into an imperative language format.

10.5.7 Program rewriting with Astre



Purpose: Perform rewriting on first order program text to achieve algorithm improvement.

10.5.8 Generating a target program with the Program Instantiator



Purpose: To translate first order ML programs into Ada.

- An Ada implementation is specified by a set of Implementation Templates.
- PI translates a first order ML program into equivalent Ada program, using representation schemes specified in the Implementation Templates.

- When compiled, the Ada program will read a string and simulate the action of an NFA to either accept or reject it.

11 Summary

The SDRR method provides an alternative to conventional methods for the design and validation of software components. In conventional methods, CASE tools can support the activities of human analysts, designers, implementors and testers, but do not usually automate their activities. Not all of these activities have high intellectual content. In particular, implementation and testing are relatively routine activities that rely heavily on prior experience. These are prime candidates for automation. In particular, the implementation/testing cycle that is repetitively executed by the software engineer while developing a component seems a particularly good target for productivity improvement through automation.

In SDRR, humans concentrate on analysis, high-level design and specification, and upon validation by formal verification. Implementations are automatically generated. The formal structure of designs and implementation specifications allow critical functional properties of software to be verified by formal proof. The degree to which design automation tools are applied to the software generation process in SDRR is unparalleled.

The enabling technologies that underly the SDRR method are the algebraic design language and the meta-programming techniques that support its translation, and the program transformation and partial evaluation techniques that have been developed over the past 5-7 years.

A Algebraic Design Language

ADL is an abstract, high-level language with well defined mathematical properties. Its mathematical properties will support formal verification of properties in the design of the DSDL. Its high level of abstraction makes ADL suitable to explain the semantics of an arbitrary DSDL. ADL is sufficiently general to support specification of the semantics of a DSDL independent of an implementation. This allows for retargeting or reuse of software component designs with a variety of interfaces, and for optimizing the performance of implementations.

ADL is a very high level, typed functional language for designing software. In ADL, control is expressed through a family of type-parametric combinators. Certain combinators are parameterized with respect to datatypes, so that they can express the control associated with structural induction for any datatype. Additionally, ADL has coinductive types and there are coinductive combinators that express the control paradigms of iteration and search.

Control in ADL is completely specified through the use of its higher-order combinators, not through explicitly recursive function definitions or loops. ADL does not support unstructured recursion. A program whose termination properties cannot be verified does not have algebraic properties. Without algebraic properties, there are many program transformations and optimizations that cannot safely be performed.

Although control combinators can be expanded by being rewritten into recursion equations, these recursions are highly structured, and have special properties. One such property is that the recursion associated with an inductive combinator always terminates. The combinators admit inductive proof rules that provide structure for formal reasoning about properties of programs. The proof rules can be viewed as theorems about the algebra of the combinators. They also provide a basis for generic program transformation tactics.

ADL also has combinators that are not simply based upon primitive induction, but can realize more complex, transfinite induction schemes. When using these combinators, it is required to prove that the domain of each application satisfies a logical constraint ensuring termination of the computation. Only terminating computations are well-defined in ADL.

A.1 Programming with algebras

There are two main approaches to expressing software designs algebraically:

- **Abstract data types** (ADT's)

  ADT's specify the theory of a signature algebra as a system of equations. Typically, these equations refer to terms in particular datatypes and the algebraic theory is executable by a rewrite semantics. An executable algebraic theory combines specification with the design of an implementation. The external view of an ADT hides both its implementation and its theory, revealing only its signature.

  The module system of ADL allows the importation of a concrete algebra without importing its abstract signature. Such an algebra is, in effect, an ADT. This mechanism is used, for instance to import an arithmetic algebra. The axioms of such an algebra extend the logic expressed by the proof rules of ADL.

- **Structure algebras**

  These are more abstract than ADT's, but they enjoy general properties useful for reasoning about programs. Structure algebras arise in the theory of universal algebras. The homomorphisms of these parameterized signature algebras are of particular interest.

  A structure algebra corresponds to a type constructor, parameterized with respect to a datatype. A great many program control structures can be characterized as homomorphisms of structure algebras. Among these are all the *reduce* functions for freely-generated algebraic datatypes, as well as more complex functions for non-initial algebras.

  A dual notion is that of a structure co-algebra. Many co-algebra homomorphisms correspond to the iterative control structures of conventional programming languages. However, their use in ADL is constrained by proof obligations that must be discharged to assure that iterations always terminate.

The control combinators of ADL are all based upon the homomorphisms of structure algebras and co-algebras. It is in this way that unbounded recursion is avoided. Instead of defining functions with recursion equations, operators are specified as structure-algebra homomorphisms. Hence, the algebraic properties of the programs are known immediately. These properties justify a variety of program transformations as consequences of the equational theory of an algebra.

A.2 Programming with monads

Two common techniques are used to compose software designs in ADL. The first is algebraic composition, which follows from the fact that algebraic programming is based upon the concept of multi-sorted signature algebras, parametric in a carrier set. When a signature algebra is instantiated with a particular carrier set that may also have algebraic structure, that structure is inherited and a composite algebra is formed. The other technique is semantic composition, using monads as the underlying structuring concept.

Monads provide a framework for structuring programming language semantics. Monads are algebraic structures that provide an abstract formalization of many programming concepts. It is through the introduction of monads that we are able to add more detail to the semantic domain. For example, state variables, I/O, exceptions, continuations, backtracking and concurrency can be added by interpreting a structure algebra in the appropriate monad. Furthermore, the desired semantic constructions can be incorporated incrementally into a design. The ability to incorporate state variables, exceptions and continuations into the semantics explicitly guides the final step of design; the translation from a purely functional, high-level design language to a lower-level implementation language with conventional, imperative features.

Monads have been advocated as a program structuring concept promoting reusability. By introducing monad definitions into ADL, we obtain a mechanism for generating composite combinators. We have successfully used monad composition as the basis for a new technique of design refinement.

A.3 Design by semantic refinement

Design refinement begins by specifying the names and types of semantic functions that realize the informally specified relations among conceptual entities described in a DSDL. As initially specified, these functions may not be effective. They may lack detailed, algorithmic definition. However, the control structure required for these functions can be specified in general terms, leading to the first refinement of high-level combinator definition. Refinement of the initial definition is provided by detailing state components and additional control refinements, such as exceptions.

Each control combinator is a higher-order function requiring a set of basic action functions as arguments. The type signature of the control combinator determines the types of the action functions. Each action function specifies the action of the combinator for a particular case or constraint of the data to which the combinator is applied. In this refinement approach, the action functions needed by a combinator can be defined independently of one another, as separable design tasks. Their designs may involve further steps of combinator specification and action-function refinement.

Selected action functions can be identified as policy parameters of the design. A policy parameter is a design parameter that specializes behavior to a particular application. A different policy parameter can be substituted to achieve a different specialization for a related application. Policy parameters are explicitly abstracted, creating derived combinators that incorporate committed design decisions but expose, through the policy parameters, design choices subject to change. In this way, the scope of variability of a design at the level of semantics is made manifest. This has important consequences for the maintenance of a software design.