Abstraction-based Certification of Temporal Properties of Software Modules

Songtao Xia
B.S., Xidian University, 1992
M.S., Southeast University, 1997

A dissertation submitted to the faculty of the
OGI School of Science & Engineering
at Oregon Health & Science University
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

February 2005

The dissertation "Abstraction-based Certification of Temporal Properties of Software Modules" by Songtao Xia has been examined and approved by the following Examination Committee:

James Hook
Associate Professor
Thesis Research Adviser

Mark Jones
Associate Professor

Richard Fairley
Professor

Andrew Tolmach
Associate Professor
Portland State University

Thomas Ball
Senior Researcher
Microsoft Research

Dedication

To my family.

Acknowledgments

It has been my pleasure to work with my advisor, Jim Hook. During the years his input and guidance has reshaped me as a computer scientist profoundly. I also thank our group, Pacsoft, for being such a friendly and helpful environment. And the OGI school, where I have witnessed many changes during my seven years' stay, but where there are always smiling faces.

The members of my thesis committee silently competed against each other in giving me more instructive comments and suggestions. They also caught many silly errors I had made.

Last but not the least, I thank my family. Guangzhi, my wife and life companion, has given me full support to allow me to graduate with our adorable twin boys, Eric and Victor.

Contents

List of Tables

List of Figures

Abstract

Abstraction-based Certification of Temporal Properties of Software Modules

Songtao Xia

Supervising Professor: James Hook

When an untrusted program (termed a module in the dissertation) is executed on a host system, the correctness of the integrated system, in particular, the security, is of concern. Many security or correctness properties can be expressed in temporal logic. Certification of temporal properties provides a piece of evidence (termed a certificate in the dissertation) that helps the host system reverify a property of concern. Examples of the applications of such certification include extensions of an operating system kernel with API safety guarantees. Previous solutions, where certificates are primarily proofs, suffer from the problem that the sizes of the certificate do not scale as the programs become larger and the properties become more complicated.

We propose Abstraction-carrying code (ACC) as an alternative certification method to the traditional proof based approaches. ACC uses an abstract interpretation of the module as a certificate. Using abstraction methods from the software model checking community, we can compute an abstract model that demonstrates that a program satisfies a temporal property; the size of such an abstract model tends to scale well when the complexity of problem increases. A module will carry this abstraction as a certificate; a client receiving the code and the certificate will first validate the abstraction and then run a model checker to verify the temporal property.

We report our efforts in building the ACC Evaluation and Prototype Toolkit (ACCEPT). Two versions of ACCEPT, ACCEPT/While and ACCEPT/C, target a simple While language and C, respectively. Novel aspects of our work include:

1. the use of a Boolean program (BP), a useful abstraction of a program, as a certificate,

2. the simultaneous compilation of the source program and the BP to generate a BP for compiled code,

3. the encoding of the BP as index types, a powerful type system that can specify and check strong program properties, and

4. the semantics-based validation of the BP through index type checking.

We report our experience of applying ACCEPT to various programs, including Linux and NT drivers. Our investigation shows that ACC tools generate certificates that scale better compared to those generated by other techniques of similar expressive power and that the time spent on model checking is acceptable for supporting many applications.

Chapter 1

Introduction

Software materializes and extends human intelligence; it provides a convenient and effective way to specialize and replicate our expertise, two activities vital to the construction of complex automated systems. With software serving as their core control and communication components, various kinds of automated systems expand human beings' efforts in exploring, shaping and reshaping the environment and ourselves.

Unfortunately, the process of replicating human expertise, which in this sense includes the construction and dissemination of computer software, is not flawless. In particular, even if a program is correctly written and an electronic copy is reliably made, running such a replica in a new environment is not always successful. The well-publicized failure of the European Space Agency (ESA)'s Ariane 5 rocket was caused by a failure within a fragment of program (a calibration function) reused from the Ariane 4 software [10]. Ironically, this calibration function should not have been run in the first place. In reality, the function was called with a parameter that was impossible for the Ariane 4 hardware but meaningful for Ariane 5 hardware. This started a chain reaction that, besides derailing a multi-million-dollar project, exposed serious software engineering issues.

The code for Ariane 4 is a top-quality program developed by parties whom the Ariane 5 team trusted. In a lot of cases, we need to integrate into our system a program developed by someone whom we do not trust. We may download an applet to run on our cell phones; Linux users may install public-domain device drivers; future air traffic control systems may reconfigure to adapt to new auto-pilot systems. Can we guarantee the correctness, or at least the security, of the newly-integrated system? This problem is more pervasive since the rapid growth of the Internet. Programs are distributed at an ever-increasing speed. The protection of systems in this situation is an important and challenging problem.

We further distinguish situations where a whole system is integrated before deployment from situations where the integration takes place after the system has been deployed and while it is actively running. Normally, integrating a new program into a running system is a harder problem because, in this case, a system often offers less computational resources for any security assurance activity. For example, in the future air control system application, the auto-pilot system interface must be integrated into the air traffic control system before the deadline of handling the data from the auto-pilot system, which happens at real time. In the meantime, the air traffic control system must function normally, responding to other requests. A full scale test, or validation of the autopilot interface may not be possible.

Next, the problems investigated by my dissertation are explained with a discussion of the general family of approaches being adopted. This discussion leads to the statement of the thesis in Section 1.1. Section 1.2 uses an example to demonstrate the approach we have proposed. Section 1.4 overviews the rest of the dissertation. Section 1.3 highlights the contributions.

In the rest of the dissertation, we refer to the system into which a new program is to be integrated as a *client* and the program of concern as a *module*. The system in which the module is developed is called a *server*. The agent who integrates the module into the client system is called the *user*, while the agent who develops the module is called the *developer*.

1.1 Thesis Statement

The topic of investigation is the abstraction-based certification of temporal properties of software modules. In this section, we explain both the approach (abstraction-based certification) and problem domain (to ensure temporal properties of software modules). Then we state the thesis of this dissertation.

The broad problem domain that we are addressing concerns the software security (and maybe correctness) problems caused by integrating into a running system a program that may be flawed, or that comes from a party whom we may not necessarily trust, or that is executed in a different environment to the one in which it was developed. An important category of solutions to these problems are language-based security approaches, a general overview of which is given in Section 1.1.1. Then we explain temporal properties in Section 1.1.2, where potential applications for ensuring temporal properties for a module are discussed. After that, extending the language-based security approaches to ensure temporal properties is briefly discussed in Section 1.1.3. These approaches often suffer from a scalability problem caused by state explosion, a phenomenon associated with verifying temporal properties. To solve this problem, we propose abstraction-carrying code (ACC), which is described in Section 1.1.4. Finally, we are able to state the thesis of the dissertation.

1.1.1 Approaches: Language-based Security

Various approaches have been proposed to address the problem of how to guarantee the behavior of a module. To directly address the issue of trust, mechanisms based on proof of identity have been studied and adopted in practice. Digital signatures based on cryptography allow a developer to convince a user of the source of the module. However, if the developer is not trusted at all by the user, the identity-based approach will not work. Furthermore, even if the developer is trusted, the program may still be flawed.

Language-based approaches rely on analysis of the instructions in the module to make sure that the module's behavior is acceptable. This category of approaches does not require the trust between a user and a developer; the only thing a user needs to trust is the program that analyzes the module. Such an analyzer is based on the semantics of the language and the logic in which the properties are expressed. So the correctness of the analyzer can be formally reasoned about. Furthermore, because these logics and semantics are relatively independent of the security requirements, the analyzer based on them can be developed over time and is considered less error-prone. Any flaw in the module, or any malicious or inadvertent tampering of the module, will hopefully be caught by the analysis. Thus language-based approaches are more suitable as a solution to the problem studied in this dissertation. Traditionally, language-based approaches are first applied to security problems. However, some of the problems studied in this dissertation concern general correctness properties.

A language-based security assurance method can be *static* or *dynamic*. A static approach makes a decision on whether the code is secure before the program is executed while a dynamic approach makes the decision at runtime. Many practical systems take a mixture of static and dynamic approaches. Static approaches have the benefit of avoiding any runtime overhead of the integrated system; they are often based on program analysis techniques. The program analysis may be accelerated with help from the server. Thus a static method can be *assisted* or *unassisted*, depending on the existence of a piece of *evidence* provided by the server. The process through which such a piece of evidence is generated and presented is called *certification*. This piece of evidence is called a *certificate*.

Take the security of Java applets as an example. An applet is an intermediate program (called bytecode) compiled from a Java program. The user may download and execute this applet at a site (the client) different than where this applet is built (the server). The loading and execution of this applet is managed by a virtual machine (a Java Virtual Machine, or JVM) on the client side. A combination of static and dynamic methods ensures that this applet cannot compromise the security of the client machine. Static approaches check the properties by analyzing the bytecode program without running it. Dynamic approaches make decisions such as whether a certain request for resources should be granted by monitoring the runtime states of the JVM.

The security of Java bytecode is based primarily on the unassisted paradigm. That is, generally, no information from a server, other than the instructions in the applet itself, is used to determine whether running the code is secure. There are a few exceptions that can be regarded as primitive forms of evidence-based certification. First, simple type information is retained by a server in the applet class file. This type information contributes to the type correctness of instructions at the client side. The client must check the type correctness to ensure the protection of security mechanism itself. Second, visibility of a class or interface member, such as whether a member function is private or public, is present in the applet. This piece of information will be verified by a client and is necessary for the correctness of the verifier, the program responsible of analyzing the applet to see whether it is safe [42, 25].

Evidence-based certification achieves security assurance in an "untrusted but cooperative" setting. This works particularly well when the information for the security decision making has to be gathered from the source code (a module is often presented to a client in compiled form) and/or takes large amounts of computational resources to calculate. A server, which often has access to the source code of the module and relatively rich resources, may compute this information for a client. A certificate will carry a certain representation of this information, which the client can use to make an informed but independent decision.

	Assisted Certification	Unassisted Certification
Server Side	Know Properties Prove Properties Generate Certificate	Nothing
Client Side	Validate Certificate Verify Properties	Analyze Program

Table 1.1: Comparison between Assisted and Unassisted Certification

Naturally, the correctness of an evidence-based mechanism should not depend on any assumption of the authenticity or validity of the information carried by a certificate because it is provided by the untrusted party. Like the procedure in a court of law, a piece of evidence should be verified independently before it can be admitted. Thus, a working, evidence-based mechanism should provide a way of verifying the evidence supplied by an untrusted party. Ideally, if the untrusted developer has cheated, either the trick is detected, or no bad consequence other than the program not being run (or other requests of resources not being granted) will occur.

The process through which a client checks the security property with the help of the certificate is called *reverification*. The process through which the validity of the certificate itself is checked is called *certificate validation*.

Table 1.1 compares evidence-based certification and traditional, purely unassisted certification. It shows that in the unassisted paradigm, the checking of properties is exclusively the client's job. Assisted certification (and the associated reverification) balances the workload of verifying a property between the client and the server. The computation that can be performed off-line is outsourced to the server. The direct benefit is that the client spends less effort in proving a property. An indirect benefit is that verification is decomposed into several activities, which can be performed separately, yet coordinatively, by the server and the client. Not only is such a system easier to implement, the client side components can also be made relatively simple. The price to pay is that more network bandwidth is consumed because additional information is exchanged between the client and the server.

For the evidence-based mechanisms, one of the classic application domains is the safe extension of an operating system kernel, which was adopted as the running example in Necula and

Lee's seminal paper [53]. In that paper, the authors proposed Proof-carrying Code (PCC), the first of its kind in evidence-based certification. The problem studied may be intuitively understood by considering the following scenario: In Linux, when a user requests a module, such as a new device driver, to be dynamically loaded into the kernel, he or she invokes the install module command: `insmod`. This command checks a set of safety properties, including that the module refers to the names in the kernel space correctly. However, the command cannot guarantee that a module is type safe. In the module one may be able to add two registers holding addresses. If we extend the function of the `insmod` command with PCC, the designer of the module will be able to convince the client of type safety.

In PCC, a server presents a proof of a safety property to a client. The client can verify this proof independently to make sure that the module satisfies the safety property. So far, PCC systems have focused on type-safety and memory safety. A program is type safe if every operation has the correct number and type of operands. A program is memory safe if every operation accesses only the part of the memory that the initiator of the operation has the right to access. These properties are very useful, but are not powerful enough to address many other aspects of program security.

1.1.2 Temporal Properties

Often, to express the security requirements of a software system, we need to address the order of events that occur during execution. For example, we may be concerned that an untrusted program uses the local APIs correctly: does a program always write to a file only after this program opens the file? Properties of this kind inevitably involve reasoning about the order of events. Although these properties, generally refereed to as temporal properties, are expressible in first-order logic, translating a property in plain English into such a logic formula is tedious. Systematical embedding (of these properties) in first order logic requires explicit introduction of the notions of time and order. What is worse, temporal properties expressed in such a manner are not suitable for mechanical verification of these temporal properties.

Temporal logics provide notions suitable for expressing and reasoning about the order of events. They are also well-supported by mechanical verifiers known as model checkers. We can use temporal logic to specify a wide range of important properties. Typical examples of security properties that are expressible in temporal logic include:

```
spin_init_lock ();
...
if (req)
  spin_lock();

...
if (req)
  spin_unlock ();
```

Figure 1.1: Example of a Linux Network Device Driver

- Reachability properties: that a particular location in a program is reachable during execution. We are often interested in a negative form of the property, for example, that a label representing that an error condition occurs should not be reached.
- Safety properties: that, under certain conditions, a bad event will never occur. For example, we never write to a file before we open it. Safety properties can often be translated into reachability properties by program translation.
- Liveness properties: that, under certain conditions, an event will eventually happen. For example, for a producer/consumer program, we may expect a consumer will eventually consume an item once the item has been produced by a producer.

As examples of potential applications of the certification of temporal properties, we list a few possible application domains. Of these applications, we will focus on the correct use of an API.

Correct Use of APIs in Device Drivers

A simple example is shown on Figure 1.1. A Linux network device driver is using spinlocks. We expect this driver to unlock a lock only after it acquires the lock, among other things.

In Linux, we may install a device driver by using the insmod command. Linux will perform a set of safety checks to make sure the device driver behaves properly. None of these checks address API safety. If we do not use locks correctly, the device driver will hang.

The sources of complexity of this problem are: First, the device driver is normally developed by someone else who we may not trust, in whom we do not have sufficient trust. Even if we do, there is no guarantee the party wrote a perfect program. Second, often the device driver is presented as a compiled C program. We have to certify binary code and have to address assembly code patterns compiled from unrestricted C programs.

Dynamic Reconfiguration of Aviation Systems

While a lot of application developers enjoy the convenience of off-the-shelf software components, their counterparts in the safety-critical domain have to follow rigid development procedures, such as RTCA/0170B [60] for air-borne systems, that help guarantee the quality of the software. The dynamic reconfiguration of a running system is considered too risky (and also too expensive) to do on a regular basis.

Driven by the need of the industry, we frequently have to reconfigure an existing system by introducing or upgrading components. For example, airlines want to be able to configure avionics systems after the aircraft have been delivered. They want to insert modules or applications of their own design to customize the system. A complicated system, such as an air traffic management system, may require distributed computation involving different forms of collaboration between deployed software components. With large amounts of software spread all over the air-ground network, frequent updates and configuration changes are likely. Some of these security issues can be resolved if we are able to ensure certain temporal properties of the modules being updated.

Applets Executed on Consumer or Embedded Systems

Java applets are executed in consumer and embedded systems such as mobile phones, PDAs, TV set-top boxes, in-vehicle telematics systems, and a broad range of other embedded devices. The computational environments of these systems vary while there is a common need for verification of the applet with limited resources. For example, Java 2 Platform, Micro Edition (J2ME) is developed for these systems. J2ME maintains a certain level of security. But application-specific security is still primarily through a security manager, which is a runtime monitor. A scenario is that, for additional functionality, such a system may install a supporting library. Whether an applet developed to use these supporting library uses this API correctly can only be monitored by the supporting library. Runtime overhead is incurred.

The JVM security for some of the systems may be maintained through an assisted approach.

In particular we are interested in extending the JVM designed for High-end PDAs, TV set-top boxes, or most embedded devices. If a JVM can support verification of an applet's use of an API that may be unknown to the JVM when it is installed, we can have both extended functionality and security. To do so, a library should be able to register its specification with the JVM and an applet should be able to carry a claim of the properties and a certificate supporting these claim.

1.1.3 Certifying Temporal Property

In the language-based security community, there are efforts to extend the power of PCC to temporal properties, such as Bernard and Lee's TPCC system, and Namjoshi's work on certifying model checkers. These approaches [8, 34, 48, 49, 69] use a proof as a certificate, resembling PCC. For example, Namjoshi studied how to construct a proof of a temporal property from the result of a model checker [48, 49].

Although such methods that can successfully compute a proof, these proof-based certification approaches suffer from an inherent problem: a temporal property is often concerned with the order of events on all possible paths during a program's execution. In Chapter 3 and Chapter 6, we will show that the proof of a temporal property normally requires the enumeration of a huge state space derived both from a large data space (variables) and from a complicated control space (branches and recursion). Thus, when the size of the problem increases, the size of a proof tends to increase drastically.

1.1.4 Abstraction-carrying Code

To reduce the size of a certificate, we seek inspiration from the process of verifying a temporal property. Model checking is the primary technique to automatically verify a temporal property. Model checking algorithm enumerate all possible (sequences of) states of a program to determine whether a temporal property holds. A piece of software can have a large, sometimes infinite, number of states. The most important method to attack this state space problem is to divide the state space into equivalence classes so that elements in the same equivalence class have the same impact on the properties of concern. This approach is known as abstraction. A more precise definition of abstraction will be provided in Section 3.4. For model checking a program, abstraction is almost always necessary. A data point in the state space of the abstract model corresponds to an equivalence class of points in the state space of the original program and this enables the model checking algorithms to be applied to a smaller problem. The following distinct features are expected from an effective abstraction method.

- An abstract model must preserve the property. That is, if a property is verified on the abstract model, the property must hold for the original program;
- The abstract model must contain substantially fewer states. An exhaustive search of all the states by a model checker is expected to be feasible and usually a lot faster.

If an abstract model satisfies these requirements, it is also a good candidate as a certificate for temporal properties. The first requirement guarantees the soundness: we can verify the temporal property on an abstract model to verify the property for the module. The second requirement guarantees that the size of the certificate will scale. It helps limit the duration of time spent on reverifying a temporal property. This discussion inspires the Abstraction-carrying code approach, that is, the topic of this dissertation.

Thesis statement:

In this dissertation, we study the problems: Can we find a scalable certification method, as an alternative to proof-based certification, to allow a client to verify temporal properties of a module in a convenient and flexible way? Are we able to leverage the latest advance in software model checking? We propose the ACC framework and show that ACC is a viable solution to temporal property certification. In addition, we demonstrate that index typed intermediate languages help us in building a sound ACC system conveniently.

We present the experience with our prototype systems to demonstrate the effectiveness of this approach, the compactness of the certificate being generated and the overall practicality of this ACC framework.

The idea of ACC can be briefly described as follows. A server will compute an abstract model of the module and send this model to the client. The client will first revalidate the model and then model check the temporal property on the model.

The next section contains an introductory example to ACC.

```
    lockStatus =0;

LOOP:
    FSMLock ();
    nPacketOld = nPacket;

    if (request !=0 && request-> status !=0) {
        FSMUnlock ();
        nPacket++;
        // other code
    }

    if (nPacketOld == nPacket) goto LOOP;

    FSMUnlock ();
```

Figure 1.2: Source program

1.2 An Example of ACC

This section uses a running example to demonstrate the problem that ACC attempts to solve, the expressive power of ACC, and the specific design choices adopted by ACC. To understand this example, readers are not required to have previous knowledge of temporal logic, software model checking, predicate abstraction or index types. The presentation here is intuitive. Background information on these subjects will be provided in the subsequent chapters. The example is based on the ACC Evaluation Prototype Toolkit for C (ACCEPT/C), which is described in detail in Chapter 6.

1.2.1 Problem

In Figure 1.2, a C program from the SLAM project [4] is used as an example. The program is a code fragment from a device driver. This driver uses locks to protect a critical section. If a user is to integrate this driver in her system, one of the safety properties she expects from the driver is that it uses the locks correctly. That is, the driver cannot lock a lock that is already locked, or unlock a lock that has not been locked. Often, this driver is presented to a user in a binary form, particularly when the code is written by a third party. The problem ACC attempts to solve is for the developer of the driver to provide a piece of evidence that guarantees temporal properties of the code. In this case, the property of concern is the correct use of locks.

1.2.2 Solution Overview

In this subsection, we describe the activities performed by the server and the client.

The server (driver developer) does the following:

- acquire knowledge of the correct use of the locks and specify it as temporal properties.
- Apply abstraction-based software model checking techniques to verify the property on the source code; if the verification is successful, an abstract model will be computed.
- Compile the source code into an intermediate program. Transform the abstract model during compilation to produce a valid abstraction of the compiled code on which model checking the temporal property is still successful.
- Encode the abstract model in a form that the end user (the client) can easily validate, i.e., decide whether the abstract model is a valid abstraction of the compiled code. The results are annotations in the low level (intermediate or assembly) program that can be verified using the semantics of the low level language.
- Ship the code and the encoded abstract model to the client.

The client's task is to reverify the temporal properties, with the help from the additional information the driver carries. She has to:

- revalidate the abstract model to prevent the server from cheating; this is accomplished by the application of the semantics of the intermediate program to verify the encoded annotations; and
- model check the temporal property to conclude that the driver is safe in terms of lock usage.

The next subsections illustrate these activities using the example.

1.2.3 Properties

The correct use of locks can be formulated as a temporal property. A temporal property is a formula in temporal logic that describe the temporal sequence of events. Modalities in a temporal logic may be interpreted as adverbs such as "always", "eventually" or "until". They provide a powerful language to describe properties of software systems. For example, the property that a lock can be locked only when it is unlocked can be written as:

$$\Box(\texttt{lockRequest} \Rightarrow (\texttt{lockStatus} == \texttt{UNLOCKED}))$$

where $\Box$ means "Always". The Symbol $\Rightarrow$ is implication. The specification requires that there is a legitimate `lockRequest`, only when the status of the lock is unlocked. The proposition `lockRequest` is true when there is a function call to `lock ()`.

Model checking is a procedure to decide whether a temporal property holds for a model. A model can be the program itself or an abstraction of the program that omits irrelevant information or operations. The model checking process enumerates the state space of the model. Namely, for every possible initial state, the model checker will compute every possible sequence of states by running the program, either explicitly or symbolically. For a common imperative language, a state of a program involves the value assignment to variables as well the program locations that are reached. For example, the value of `lockStatus` is relevant because it is part of the temporal property of concern. Because there can be variables that can take on a large number of, or infinite, possible values, and control structures that involve choice (such as if statements), the state space of the original driver can be huge.

In practice, this problem is attacked by applying abstraction techniques to reduce the size of the state space.

1.2.4 Predicate Abstraction

Abstraction is often a necessary step in model checking software systems; it simplifies a program by considering a smaller state space that is sufficient for verifying the property of concern. Predicate abstraction is a particularly successful abstraction technique that characterizes the system state by considering the truth values of a few predicates. For example, in the above model checking example, it is sufficient to consider the following predicates:

```
lockStatus == LOCKED
lockStatus == UNLOCKED
nPacket == nPacketOld
```

We only consider the effect the program has on these predicates and control decisions shall be made using information limited to the truth values of them. For example, the correct invocation of a `lock` () function call will require the second predicate to be true, and, if so, it will change the value of the first predicate to true and the value of the second predicate to false.

The predicate `nPacket == nPacketOld` is particularly interesting: in the loop body, the lock is released when and only when `nPacket` is changed in the `if` statement. Therefore, if the program exits the loop, then we know that `nPacket` and `nPacketOld` are equal and that the lock must be locked. It is necessary to unlock the lock after the loop. If the abstraction is overzealous and fails to keep track of the relation between this predicate and the status of the lock, then model checking cannot tell whether the lock is held when the loop exits. It may falsely conclude that the lock can be unlocked twice.

A Boolean program (BP) represents these predicates as Boolean variables and restricts the forms of operations allowed on these Boolean variables. A BP may capture just enough information for us to model check the program. Two important features of a BP are: 1) for certain properties, including those characterizing the use of locks, if the properties hold for the BP, then the properties must hold for the original program; and 2) a BP is a small abstract model on which model checking is usually fast. The BP computed for the example is listed in Figure 1.3. Model checking this BP will determine whether a distinguished label ERROR can be reached. Because the label is reached when a wrong sequence of lock operations occurs, that it is not reached proves that the BP is free from the erroneous sequence of lock operations. Because the property that a label will not be reached belongs to the kind of properties preserved by Boolean abstraction, we are assured that the original program is free of such a bad sequence.

The BP is computed by a local translation of the statements in the source code. A BP statement is computed to capture the effects of a concrete C statement over a set of Boolean variables. Each of these Boolean variables corresponds to a predicate in the C program. For example, the Boolean statement translated from the concrete statement `nPacket++` is:

```
b_3 = if b_3 then 0 else *;
```

```
  /* b1: lockStatus = 0
     b2: lockStatus = 1
     b3: nPacket = nPacketOld
  */
  (b1, b2) = (1,0); /* inlining FSMLockInit();
                       simultaneous assignment.
                       source statement lockStatus =0 sets b1 to true (1)
                       and b2 to false (0).
                     */
LOOP:
  b3 = 1;           /* source statement: nPacketOld = nPacket; */
  if (b2) goto ERROR;
          else (b1,b2) = (0,1);
                    /* inlining FSMLock () ;
                       source:
                         if (lockStatus ==1) goto ERROR;
                                 else lockStatus =1;
                    */

  if (*){           /* variable request is not observed. Taking either
                       branch will not affect the property of concern. */
    if (b1) goto ERROR;
            else (b1,b2) = (1,0);
                    /* inlining FSMUnlock (). */
    b3 = choose(0, b3);
                    /* source: nPacket ++
                       if nPacket== nPacketOld beforehand, then they are
                       not equal afterwards. */
  }

  if (*) {
     assume (!b3);   /* test if nPacket== nPacketOld. */
                    /* take a non-deterministic choice, if control reaches here,
                       assume the condition b3 is true. If b3 is not true, the
                       model checker will not proceed. */
     goto LOOP;
  }
  else {
     assume (b3);
     goto NEXT;
  }

NEXT:
   if (b1) goto ERROR
          else (b1,b2) = (1,0);
                    /* inlining FSMUnlock (). */
```

Figure 1.3: Boolean program for a device driver

where b_3 is a Boolean variable for `nPacket == nPacketOld` and symbol * is a special value. When tested by a model checker, this * can take either *false* or *true*. Later we shall see that this * is in fact a top value in a lattice. In other words, it represents a non-deterministic choice made by the model checker. This assignment characterizes the effect this concrete statement has on the predicate. Namely, if `nPacket == nPacketOld` before the statement, then the predicate will not hold afterward.

An `if` statement in the source is translated into an `if` statement in the BP. In computing the condition for a branch to be taken, certain approximations are used. This may result in a situation where the conditions granting the then and the else branches are not negations of each other. To address this issue, in a BP, the condition after `if` is often set to *. In the branches, there are `assume` statements that characterize the condition that must hold when control reaches that point. In the BP in Figure 1.3, we use the test of the loop condition as an example of this structure, although in this case the conditions in the `assume` statements are negations of each other. Details of how the approximation is computed are presented in Chapter 3.

Readers who are not familiar with model checking can view it as the symbolic execution of the program. Details of the syntax of BPs and the model checking process are introduced in Chapter 3.

In ACC, a server may deliver a BP together with module. There is one remaining problem: The BP in Figure 1.3 is computed for the source program; the program that the server eventually sends out to a client is binary. The driver developer has to provide a BP of the compiled code to the user. For reasons to be discussed in Section 5.2, in ACC, computing a BP for the compiled program is done through certifying compilation: the compiler will transform the source program and the BP for the source program in parallel; the result of the compilation is a target program and a BP for the target program.

1.2.5 Certifying Compilation

In ACC, a server uses abstraction-preserving compilation to generate a BP for compiled code. The idea is to compile the program while maintaining the correspondence between the compiled code and the BP. Thus the BP for a source program is used as the BP for the target program. This approach may result in a target program that is not aggressively optimized; a separate optimization process may be necessary for applications that expect target programs of better quality.

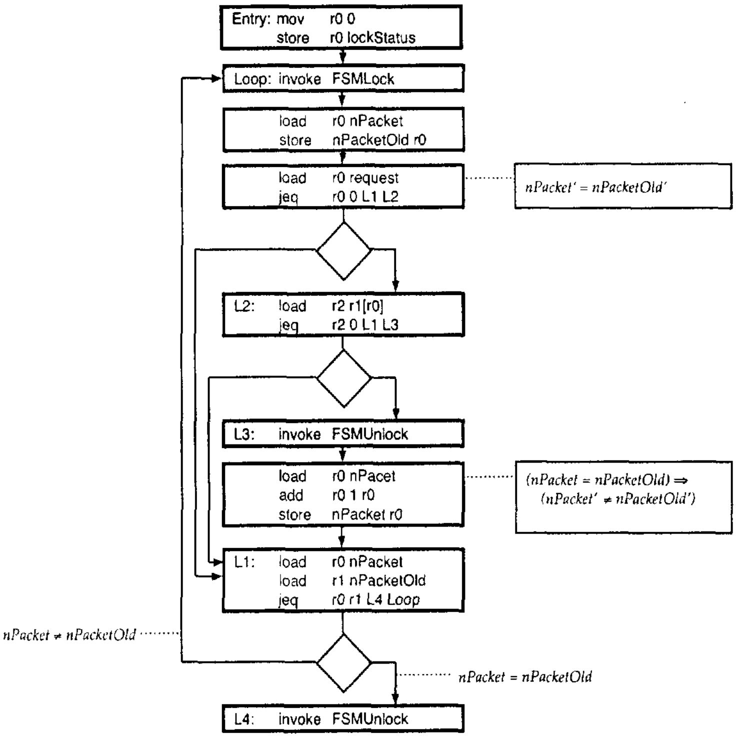


Figure 1.4: Compiled program with annotations

For example, an assignment in the source corresponds to a simultaneous assignment in the BP. The BP assignment characterizes the effect the source assignment has on the predicates of concern. Such an effect can be viewed as pre and post-conditions. In particular, the effect can be captured by a Hoare triple of the form $\{P\}\texttt{stmt}\{Q\}$, which asserts that if P is true before the statement `stmt`, then Q must be true after executing the statement. Thus, the compiler may compile the source program into the target program while preserving these triples. This is a conservative approach that limits the types of optimizations that can be performed by the compiler. A compiled fragment of the program is shown in Figure 1.4, organized in a flow diagram form.

1.2.6 Abstraction Validation

As mentioned, the result of the compilation is a BP and compiled code. This time, the BP characterizes the effect of the compiled code on the predicates; the control decisions made by the BP are based solely on the information contained in the predicates. The BP is a certificate of the temporal property for the target program. It is subject to model revalidation: the BP has to be checked to see if it is a valid abstraction of the module according to the semantics of the language in which the module is compiled to.

In ACC, the BP is encoded in a set of type annotations and the model is validated through an extended type checking process that precisely characterizes the algebraic semantics of the intermediate language. This is accomplished through a so-called index type system, introduced in the later chapters. Index types can relate pre- and post- conditions of a program segment in a convenient way and are suitable for the encoding of Boolean programs.

We do not introduce the syntax of index types here, which can be found in Section 3.5.1. In Figure 1.4, we can see two kinds of annotations. The first kind of annotation is associated with a block of instructions. For example, the ones to the right in Figure 1.4 are such annotations. They are logic formulas that represent the same computation effects as the corresponding Boolean assignments. The index type encodings are to the effect such logic formulas. In the annotations, primed variables refer to the variables in the state after the block is executed; the unprimed variables refer to the ones in the state before the block is executed. Take the annotation to the lower right as an example. The logic formula indicates that if $\texttt{nPacket} = \texttt{nPacketOld}$ before the block, then the predicate is not true after the block. This is the same as Hoare triple $\{\texttt{nPacket} = \texttt{nPacketOld}\}I\{\texttt{nPacket} \leq \texttt{nPacketOld}\}$, from which we can recover the BP.

The client will validate these annotations to decide if the encoded BP is a valid abstraction of the compiled code. The validation can be understood as the validation of Hoare triples as described above.

1.2.7 Property Reverification

After the BP is validated, a model checker can verify the temporal property on the BP. Because BP preserves a class of temporal properties, including the most common ones in the software verification domain, checking the BP guarantees that the property holds for the compiled program.

A BP may contain function calls and is modeled as a push-down automaton. Existing tools model check such an automaton. For example, in ACCEPT/C, we apply *moped* [63] to model check the BPs.

1.3 Contributions

We discuss two major contributions of the dissertation. The first contribution is that we propose the concept of abstraction-carrying code, implement two prototype systems (ACCEPT/While and ACCEPT/C), and investigate the practicality of the ACC approach. The second contribution is at the implementation level, where we use an expressive type system to encode a Boolean program.

The first contribution distinguishes our research from previous, proof-based certification methods of temporal properties and from the recent development of model-carrying code [64, 65].

Previously, a certificate of a temporal property of a module is primarily a proof, which suffers from the scalability problem. ACC is a different approach where a certificate is simply an abstract transition system. As will be shown by later chapters, ACC is a viable alternative approach that scales better in certificate size but is inferior to proof-based approaches in some other aspects. Our experiments will show the penalty caused by the weakness of the ACC method is acceptable for some applications. Therefore, ACC enriches the solution space of the certification of temporal properties.

Model-carrying code (MCC) [64, 65] uses a model as a certificate for security properties. MCC and ACC are similar in expressive power and concept. A model in MCC is a transition system and the properties being certified by MCC in practice are temporal. But MCC and ACC are different in application domains and certification methods: MCC certifies a standalone system, while ACC certifies a module. In MCC the certification is through runtime learning [65], while ACC uses software model checking techniques. The models generated by MCC are not always statically verifiable and runtime monitoring by a client is sometimes necessary, while the reverification in ACC is purely static.

The second contribution is the introduction of an index typed assembly language as the intermediate language. The validation of a certificate is through the type checking of the type system of the intermediate language. This is an innovative application of an index typed assembly language. Index types have been previously studied for functional programming languages. It provides a programmer with strong expressive power in a type system. The use of decision procedures allows index type checking to be automatic. Index types have been applied to the certification of some hard memory safety properties in a typed assembly language [72]. Our research explores the connection between abstract interpretation and index types. This connection allows us to specify and verify an abstraction relation between compiled code and a Boolean program.

1.4 Organization

The rest of the dissertation is organized as follows.

Chapter 2 compares assisted and unassisted security assurance of a reconfigurable module. For the unassisted paradigm, we use the Java Virtual Machine as an example. For the assisted paradigm, we introduce Proof-carrying code (PCC) framework in the context of safe extension of an operating system kernel.

Chapter 3 reviews technical results used throughout the dissertation. We present the syntax of Linear Time Logic (LTL) and the model checking process to decide the truth value of an LTL formula with respect to a state-transition system. Such model checking processes rely on the enumeration of the state space of the transition system. Abstract interpretation allows us to reduce the number of states being considered. The general theory of abstract interpretation, as well as the automatic predicate abstraction techniques used in software model checking is presented.

Chapter 3 also studies the connection between index types and abstract interpretation. The general idea of index types is introduced. We introduce the type system of SDTAL and show how this type system can be used to represent a BP in an easily verifiable way. The soundness of the index type system of SDTAL is proved.

Chapter 4 discusses the general framework of ACC. Features expected from a certification system are summarized.

Chapters 5 and 6 present two prototype systems: ACCEPT/While and ACCEPT/C. For ACCEPT/While, we relate the semantics of a While language and explain how compilation is done to preserve the BP computed for the source language. The soundness of ACCEPT/While is proved based on the soundness of the SDTAL. For ACCEPT/C, we focus on how existing software model checking tools can be integrated to build an ACC system. For both systems, experimental results are reported.

Chapter 7 discusses related work and concludes.

Chapter 2

Background: Language-based Security

This chapter describes language-based security. It contains two parts, introducing examples of the unassisted and assisted paradigms, respectively. First, we review how the JVM security works in Section 2.1, which sets up a typical scene where language-based security approaches can be applied. The JVM security mechanism is, in large part, unassisted. The application of assisted approaches in JVM security is either less significant or simple. Then, we introduce the principles of PCC in Section 2.2. Unlike most of the security mechanisms used by JVM, PCC is assisted in that a server must provide information to help the client prove security properties. Finally, some criteria of extending PCC to certify temporal properties are discussed.

2.1 Security Mechanisms for Java Applets

A Java Applet is a compiled Java program that can be embedded in a web page. When this web page is opened, the applet may run on the client machine. It enables a web browser to execute a remote program locally. Applets also cause serious security problems: applets are programs written by untrusted parties. Unfortunately, running an applet within a browser often gives a user a false sense of safety. The popularity of applets often contributes to the vulnerability they have caused.

The Java security system has been widely documented and studied [42, 28, 26, 13, 70, 67, 59, 57, 54, 55]. The account below focuses on the types of decisions the security system has to make, the information base on which these decisions are made, and the mechanisms for acquiring the information.

In Section 2.1.1, I summarize the security policies of the JVM. Then, in Section 2.1.2, we demonstrate how the JVM uses a combination of static and dynamic checks to help enforce these security policies.

2.1.1 Security Policies

Running an applet exposes a client to the risk of attack. A malicious program may modify the client's system, peek into the client's private data, or degrade the performance of the client machine. To protect a client from these kinds of attacks, the designers of the Java applet security system must draw a clear line between what an untrusted applet can do and what it cannot do.

For an applet, there is a forbidden list of operations [25]. For example, an untrusted applet is prohibited from reading, writing, renaming or deleting files on the client file system. An untrusted applet can neither create a network connection to any computer other than the host from which it originated nor listen for or accept network connections on any port on the client system.

These rules are known as security policies, enforced by the so-called sandbox model, introduced in the next subsection.

2.1.2 The Sandbox Model

The sandbox model can be viewed as a chain of protection mechanisms; each link of the chain works at a different level of abstraction. The programs that implement the three major mechanisms are called security managers, class loaders and a verifier. A security manager works at the policy level, making decision of whether or not to grant a request from an applet; a class loader and a verifier make security-related decisions at different levels of program constructs. They prevent a bad applet from shortcutting the security manager.

In Figure 2.1, we illustrate the components of a sandbox model. This figure is a reproduction of the one in the classic literature[42].

A security manager decides whether the applet has enough privilege to access the resources it requests. A standard security manager, for example, may block the request from an untrusted applet for a local file or an Internet connection. However, the client can program the security manager in a way that is flexible for certain applets.

The impact of the security manager is transparent to an applet developer. The requests for important resources are through Java APIs. The library code of these APIs will report the requests

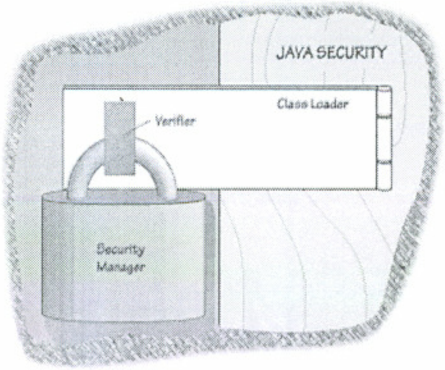


Figure 2.1: Java Sandbox Model

to the security manager for approval. The likely events in sequence are:

- An applet uses the Java API to attempt a potentially dangerous operation.
- The API code asks the security manager whether this operation is legitimate.
- If the manager intends to deny the operation, a `SecurityExpection` will be thrown back to the API, which propagates to the requesting applet.
- If no exception is thrown, the API call returns smoothly and the applet accomplishes its operation.

For this scheme to work, care must be taken to prevent an applet from circumventing the security manager. For example, an applet should not be allowed to use an API other than the one used by the client JVM, because this (unlawful) API may deliberately grant every operation without consulting a security manager. Class loaders and a class verifier are designed to provide this protection.

When an applet is loaded, an applet loader, which is a class loader, is invoked. This loader prevents the external applet class from overriding the Java API. For example, the applet is not allowed to load a replacement of java.lang.SecurityManager class.

The class file verifier checks the integrity of the mobile code loaded by a class loader. If the verifier finds anything suspicious, an exception will be raised and the attempt to load the class will fail. Although there is no particular way for a verifier to tell if a program in binary form is generated by a compiler or is created by a malicious hacker, the verifier prevents the applet from cracking the security system by enforcing a set of rules. Among other things, the class verifier will check if Java's static semantics are followed. For example, a class cannot be derived from a final class.

One important component of the class verifier is the byte code verifier, which ensures the following:

- Instruction level type safety: That the operands needed by an operation are of the right number and type.
- Program counter safety: That the program counter cannot be set to an address that is not the beginning of an instruction, and that the program counter cannot point to somewhere beyond the boundary of the current method.
- Correct initialization of an object: That an instance of a class can only be used after an initialization method of the corresponding class has been invoked.
- Visibility: That a field or member function that is not supposed to be visible in Java should not be visible at the bytecode level. For example, a field of an object can only be accessed (or a method can only be invoked) if this field (method) is visible to the class where access is requested.
- Stack safety: That no operand is pushed onto a full stack and that there is no attempt to pop from an empty stack.
- Local variable (register) initialization: That a register cannot be loaded from unless there is a previous store, or this register corresponds to an argument of this method (which means that it is initialized when the method is invoked).

As mentioned, these checks, performed over various levels of abstraction, are intended to prevent a malicious applet from bypassing the security manager. For example, if the type safety at the instruction level is compromised, then an attacker may be able to cast an arbitrary integer as an address and gain access to the frame stack, from which point a worm-like attack may be mounted. At the Java class level, if the JVM is confused about the type of an object, an attacker may refer to an instance of a security manager through a pointer that seemingly points to an object of a harmless class. To underline this point, a team at Princeton has written a tool that can exploit any type confusion involving an object type (types that are derived from the Java Object class) to gain full access to the client's machine [18].

Some of the checks performed by the bytecode verifier are simple, while some require non-trivial analysis of the byte code program. For example, the integrity of the stack is enforced by a stricter rule that says when control repeatedly reaches the same program location, the operand stack must be of the same length and contain the same type of operands. This is a loop invariant that is validated eventually by a built-in theorem prover [25]. The verifier checks such conditions by analyzing the byte code itself; the check is not assisted by any other data artifact generated by the compiler.

There are also some properties that are not easily determined by static verification. For example, the index of an array access should not be outside the array bounds. It is impossible to validate such properties entirely at load time. For these kinds of conditions, the JVM uses run-time checks, which may degrade the efficiency of running an applet.

Overall, the security of the JVM relies on techniques from several levels of abstractions. Languages technologies, such as type systems for an intermediate language, dataflow analysis and run-time monitoring, play an important role. Security policies on file systems, networks and other resources rely on these techniques directly or indirectly.

The dataflow analysis performed by the byte code verifier is of concern here. For example, because dataflow analysis is typically expensive, and more so when the properties are more complicated, the number and category of properties being statically checked is limited. This either leads to a stricter, but easy to check, set of rules, or means that we may delay the check until run-time.

However, in the assisted verification paradigm, some of the problems will naturally disappear. We may be able to statically verify complex properties without spending much time and/or degrading runtime performance.

2.2 Proof-carrying Code

About the same time as the advent of the Java technology, a group in CMU were working on the Fox project [9]. The goal of the Fox project was to investigate the application of rigorous type theory, and functional programming languages (in particular, Standard ML), to the safe and dependable construction of systems software. The Fox project was successful in many respects, including typed intermediate languages and the functional construction of network servers; the most relevant to this dissertation is Necula's work on Proof-carrying Code (PCC).

The first influential paper on PCC appeared in the Conference of Operating System Design and Implementation (OSDI), 1996, and was entitled "Safe Kernel Extension without Runtime Checking" [53]. The paper studies this problem: if we want to extend an operating system at runtime by installing a kernel module – for example a device driver – how can we make sure that this newly added part will not compromise the behavior of the rest of the system? In general, the driver might be supplied in binary form, and it might come from an untrusted source. How can we get the assurance (of safety) without degrading the performance of the operating system?

The setting of safe extension of an OS kernel is different than that of Java bytecode security. To begin with, the potential security risk caused by a device driver is greater than that caused by an applet because the driver often has access to kernel data structures. Thus the protection domain is more extensive. More importantly, an operating system kernel requires prompt response to events, and the use of a runtime monitor may have unacceptable performance overheads. We cannot introduce another layer of API to separate the device driver and kernel data structures. On the other hand, the developers of a device driver share common interests with kernel developers. They normally understand the security concerns of kernel developers and they are willing to perform extra verification tasks to satisfy kernel developers. This cooperative attitude is essential to the success of the PCC system. However, the security of a PCC system does not derive from trusting device driver developers: any information submitted by them must still be subject to reverification.

These distinct aspects of the problem influence the design of a PCC system: the security check is assisted by the server. The server can give hints, which themselves are independently verified, to help the client to draw conclusions regarding security policies. With the help of such hints, the client (the end user) can check more powerful properties than without. These checks are primarily

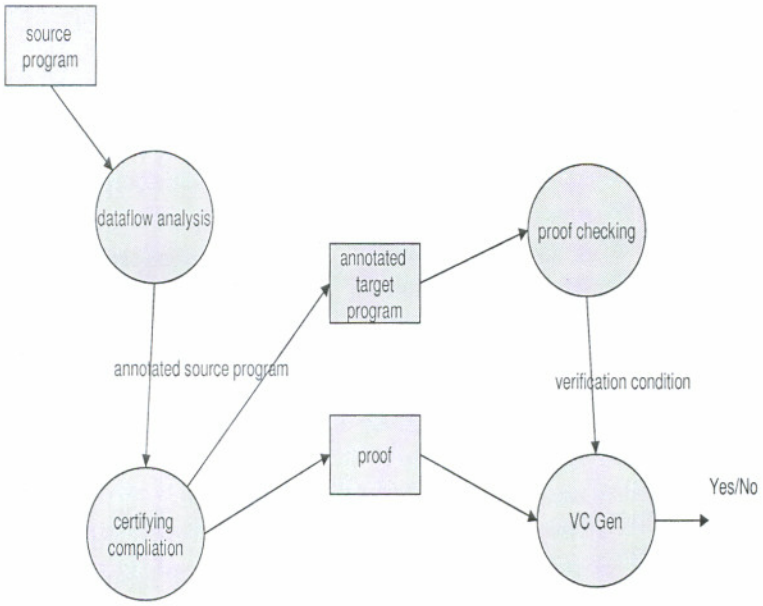


Figure 2.2: dataflow in a PCC system: top-level

static. In PCC, the certificate prepared by the server is a proof of the statement that the program satisfies a safety property. The module will carry this proof when it is acquired by a client. Hence the name proof-carrying code. This proof will be checked by the client when the module is loaded.

2.2.1 PCC Overview

A high level description of a PCC system is: When a server compiles a module, it will first prove that this piece of code is safe. When the server ships the code to a client, the server will attach the proof to the code to be used as a certificate of the safety property. A client will verify the proof to check the safety property, which is known to both parties at the beginning. This process is illustrated by the top-level dataflow diagram in Figure 2.2.

PCC systems [40, 50, 51, 52, 1, 45] have focused on safety properties such as type safety (that programs may not add two addresses, for example) or memory safety (that programs may not refer to a memory location that is not allocated to them). A server analyzes the program, proves the safety property and compiles the source program into an intermediate/assembly program while preserving the safety properties. More over, the server computes the safety proof in a form that is

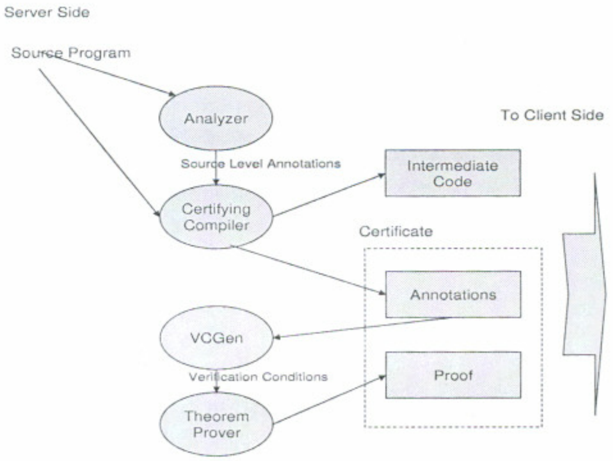


Figure 2.3: dataflow in a PCC system: Server Side

independently checkable by the client.

For some harder properties, it may be necessary for the server to annotate the compiled programs with dataflow facts. Forms of such dataflow facts include pre- and post- conditions of a function, and loop invariants. To generate such annotations automatically, a server may perform complicated and time-consuming analysis. However, with these annotations, proving the safety properties will be simpler for the client. In most PCC systems, these dataflow annotations are also attached to the module as part of the certificate so that they can be used by the client. Of course, because these annotations are from the server, the client ought to check the validity of these annotations independently, but it does not need to generate them from scratch.

Both the generation of the proof by the server and the reverification of the proof by the client are based on the semantics of the compiled program, with the help of the dataflow annotations. More specifically, a program, called VCGen (Verification Condition GENarator), is shared by the server and the client. The tasks of a VCGen are to interpret the security requirements to verify the annotations, and to verify the security properties. To achieve these goals, a VCGen generates a set of logic formulas (the verification conditions, or VCs) by scanning the intermediate program and dataflow annotations attached. These propositions must hold for the safety property to hold. These VCs are treated differently between the server and the client. The server calls a theorem prover to prove the VCs. The resultant proof is the proof that is attached to the intermediate program. The

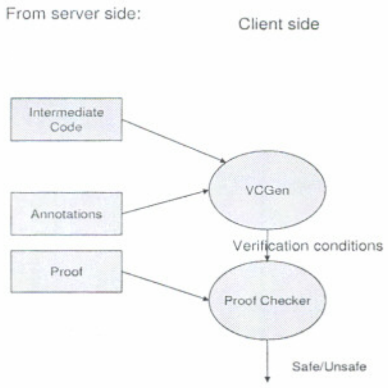


Figure 2.4: Refined dataflow in a PCC system: Client Side

client invokes a proof checker to verify the proof.

In a PCC system, the server always performs harder tasks and leaves the easier reverification task for the client. In particular, computing the dataflow annotations often requires fixed point computations while checking them is relatively simple. Also, checking a proof is much quicker than generating a proof. The proof-checking process does not involve the many backtracking steps that the proof discovery process typically requires. This is a fitting response to the problem requirements: The server's computation is done off line and can take a much longer time or involve human intervention, while the client's computation happens at load time and is not expected to be time-consuming.

Figures 2.3 and 2.4 illustrates the dataflow in the server side and the client side, respectively.

For the PCC system to be safe, the annotations and the proofs of VCs both need to be independently verifiable. The annotations are verified using the semantics of the intermediate language into which the module is compiled and the semantics of the logic in which the annotations are written. The typical way in which the proofs are encoded in PCC guarantees the effectiveness of the proof-checker, that is, a proof that passes the proof-checker is a valid proof of the VCs.

Thus, barring programming errors introduced in implementing a few core components, a PCC system is provably impenetrable with respect to the safety properties of concern. This set of components must be trusted by the client; they are known as the *trust base*. In a PCC system, the trust base comprises the VCGen and the proof-checker. Both are small and well-understood programs. Notably, the trust base does not contain a theorem prover, which is usually a large program that performs complicated computations. As mentioned, in PCC, such computations are performed on the server side.

The programs used by a server do not have to be correct to protect the client. In particular, if a malicious user produces a malicious code and forges a proof, then one of the following two situations must happen[50].

- The client finds that the proof does not match the generated VCs and rejects the code. There are two possibilities: either the proof is not valid or it does not prove the VCs.
- The client finds that the proof is valid for the VCs. Then the execution of the code is simply safe (at least it is compliant with policy).

In either case, this malicious code will not affect the safety of the system. However, it is possible that, for a module that is indeed safe to run, the client may not be able to prove its safety, or may come up with an incorrect proof.

2.2.2 Discussion of PCC

To facilitate extension of PCC to address temporal properties, we use the following criteria to analyze the PCC approach. These criteria can be considered as dimensions in a solution space for program certification, and different ways of extending PCC may be assessed using these criteria.

- *Expressive Power*

  Currently, most PCC systems are concerned with type safety and memory safety, although there are efforts to extend the PCC approach to other properties. Additional expressive power may require stronger decision procedures, and, for some properties, an automated certification process may not exist. Also, stronger properties could result in a larger certificate size and a longer reverification time. TPCC, for example, generates large proofs for simple properties.

- *Efficiency*

Efficiency may involve several factors: the size of the certificate and the reverification time are the primary ones. In PCC, the size of the certificate is related to the nature of the property.

- *Automation*

  PCC implements a fully automated certifying compiler. The level of automation is determined by the existence of an automated analysis method to decide a safety property. It is relatively easy to extend such an analysis into a certification method.

- *Size of the Trust Base*

  PCC's trust base includes a VCGen and a proof-checker. Both are small programs that are well formulated and understood. The ACC approach we describe above will introduce new programs into the trust base.

- *Modularity*

  The VCGen in PCC hard codes the safety policy. If a new policy is adopted, then we have to write a new VCGen. Therefore, such an implementation does not scale well to other properties.

By attaching a proof, a server enables the client to verify the proof and thus save time in proving the VCs all by itself. But, in most cases, the automated theorem provers are fast. For example, cooperative decision procedures tend to use the simplex method to solve integer equalities, the algorithm performance of which is, in worst case, exponential and, in average, linear. And nowadays there are fast SAT solvers[47, 77], which are incooperated in decision procedures. This fact suggests another design choice: the server may not attach the proofs for VCs and instead leave the client to invoke the automated theorem prover to discharge the VCs. This exposes the client to two risk factors: First, reverification could be time consuming and not fit for many applications. Second, a theorem prover instead of a proof checker is now included in the trust base; a proof checker is likely to be much simpler and less error-prone to implement than a theorem prover. Therefore, this design choice is less dependable than the one used by the original PCC implementation.

But this alternative has its own merits too. Most importantly, there is no need for a mobile program to carry proofs. Therefore, the certificate contains only the dataflow annotations, and the size of the certificate is decreased. For complicated properties, when the size of the proof is huge and when network bandwidth is limited, this design choice could be sensible.

This discussion relates to a common misunderstanding of PCC: people tend to think that the efficiency of PCC is based on the asymmetry between searching for a proof and checking it. Although this is true, one should not confuse this asymmetry with the time difference between proving VCs and checking VCs. The computation of the dataflow annotations by the server plays a significant role, especially for more complicated properties. Consider type safety; as long as the type information is retained in the target code, type checking is unlikely to consume significant computational resources. For memory safety properties, the search for a loop invariant takes several iterations of dataflow analysis. Once the loop invariant is computed, the number of VCs is linear in the size of the program (though it may be exponential in the size of the VC). Therefore, attaching proofs for VCs, depending on applications, may not be a defining feature of program certification. An interesting perspective is to view the certifying compiler as a big theorem prover. The annotations that it generates can be viewed as proof sketches of the overall safety property.

This discussion may inspire alternative ways of implementing a PCC-like certification/reverification system. In particular, we may trade reverification-time and the size of the trust base for a reduced size of certificate. One approach that belongs to this category is Hongwei Xi's index type system, where the asymmetry only exists at the annotation level: a server performs dataflow analysis of the code and annotates it with the results. The server, however, does not attach a verifiable proof of the fact that the result is valid, as would be the case with a PCC system. A client, in turn, will have to reprove the dataflow results by invoking a theorem prover, partially embedded in the index type checker. Still, if the the theorem prover is fast enough for most of the input, as is the case with many applications of index types, then the approach is practical. The gain is that the time spent by the server on dataflow analysis, which may require several iterations, is avoided on the client side.

The idea of abstraction-carrying code derives from this trade-off. With ACC, we trade the size of the trust base and reverification time for increased expressive power and a decreased certificate size. Thus, ACC is useful in applications where temporal properties are key to the safety of the mobile program, especially where network bandwidth is limited. We will introduce the technical preliminaries of ACC in the next chapter.

Chapter 3

Preliminaries

Temporal properties of a program characterize the order of events that are observed during the execution of a program. They provide a convenient way of describing many important and practically useful properties. We define Linear Time Logic (LTL), a form of temporal logic in Section 3.2. Syntax and semantics of these logics, as well as the transition models over which the semantics is defined, are presented.

Model Checking [39, 21, 38] is an automatic method of checking a computational model to determine whether it satisfies a temporal logic formula. I cover the basics of model checking in Section 3.3, with an emphasis on software model checking. Software model checking is practically impossible without first abstracting the program into an abstract model. This approach, based on abstract interpretation, is introduced in Section 3.4. A particularly useful abstraction method, predicate abstraction, including a variant Boolean abstraction, is introduced in Section 3.4.2.

Index type systems extend traditional type systems with additional expressive power that is capable of specifying and verifying complicated program properties. The way this extension is constructed is fundamentally connected to abstract interpretation. In ACC, we will explore this connection to express an abstract model using index types. Index types are introduced in Section 3.5.1.

Part of the advantage of using index types is that the soundness of the whole ACC system can be built on the soundness of the index type systems. An index typed intermediate language, Simple Dependently Typed Assembly Language (SDTAL), is introduced in Section 3.6.2. Its syntax, semantics, type checking rules and soundness results are presented.

3.1 Temporal Properties

We first describe labeled transition systems and then define the syntax and semantics of LTL.

We are interested in observing a program's behavior, the modeling of which involves at least two elements. First, a mechanism needs to be designed to take snapshots of the system. For this, we choose a logic to describe the relations among the variables of a machine state. Second, we need a mechanism to describe the behavior of the system, especially the relative order in which events of concern happen. If an extension of first order logic allows us to describe a snapshot, then a temporal logic formula allows us to describe the behavior of program executions.

A (labeled) state transition system captures both aspects of the program's behavior. Below, we model a program as a state transition system, on which we will define the syntax and semantics of temporal logics.

3.1.1 State Transition System

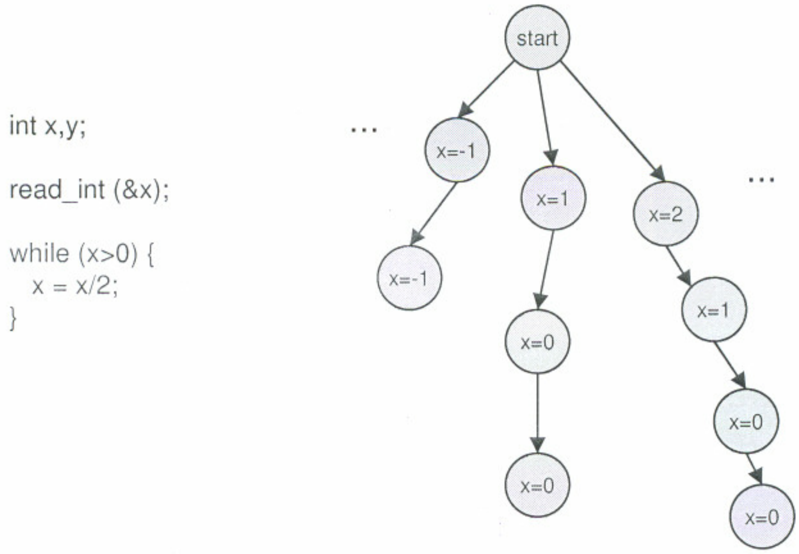
A transition system is a tuple $T = (S, R, S_0)$, where S is a set of states, $R \subseteq S \times S$, and S_0 is the set of starting states. A path π of T is a (potentially infinite) sequence of states $s_0, \ldots, s_n, \ldots$ such that $s_0 \in S$ and $\forall i.R(s_i, s_{i+1})$.

3.1.2 Runs and Trees

For a labeled transition system, given a description of the input state, we can compute a sequence of successor states. Such a sequence is called a *run*. Formally, a run is a sequence $s_0, s_1, ..., s_n, ...$ where every s_i is a state of the program and $s_i \rightarrow s_{i+1}$, for every s_i. A run can be finite or infinite.

We use π to represent a sequence of states. We may want to talk about an element in π, or a suffix of it. If π is $s_0, ...$, we use $\pi(i)$ to refer to s_i, and use π^i for the suffix starting from and including s_i.

Consider the set Π of all runs of a program. Sometimes it is convenient to organize Π as a tree. Assuming all $\pi(0)$ are sets of machine states, from a state, we may have different successors. For concurrent programs, the different successors may be caused by the different scheduling decisions. For sequential programs, one state may have different successors due to a conditional test in the program. For example, in Figure 3.1, the program to the left is represented by a tree structure to



```
int x,y;

read_int (&x);

while (x>0) {
  x = x/2;
}
```

Figure 3.1: A Tree Structure of Runs

the right. This tree structure of runs will be referred to as the model on which the semantics of temporal properties are defined.

3.1.3 Büchi Automata

A finite state transition system can be viewed as a finite state automaton. To do so, we simply add a set of final states. A finite state automaton can be viewed as a language acceptor. A finite sequence of transitions is accepted by the automaton iff this sequence of transitions ends at a final state. When modeling a program, the input language is the set of labels of the transitions, that is, statements or conditionals in the program. That the finite state automaton accepts a string is the same as that the execution of the program reaches a final (accepting) state.

To model reactive systems, which normally do not terminate, a finite state automaton needs to accept infinitely long sequences of transitions, known as ω- *sequences*. This acceptance condition is called ω- *acceptance*.

For a sequence X = $x_0 x_1 ... x_n, ...$, let X^ω be the set of symbols that appear infinitely often in X. A finite state automaton $(\texttt{Prog}, S, \rightarrow, s_0, F)$ accepts X if $X^\omega \cap F \neq \phi$. `Prog` are the statements or conditionals in the program used as labels on the transitions.

A finite state automaton with an ω- acceptance condition is known as a Büchi automaton [14]. A nice property of Büchi automata is that we can translate a temporal logic formula in LTL (Linear Time Logic) to a Büchi Automaton [14]. Thus, both the specification of the system and the specification of the property are represented in the same automata form. This leads to an elegant model checking algorithm for LTL formulas. We will return to this issue in later sections.

3.2 Linear Time Logic

Temporal logic specifies properties relevant to the temporal orders. Examples are:

- Event A keeps happening until event B happens.
- Eventually event A will happen.
- Event A may never happen.

For the verification of a software system, properties that are true on all runs are particularly important because these are the only ones that are practically verifiable. Below we present Linear Time Logic (LTL) [14].

Historically there are two families of temporal logic systems, based on tree models and on linear model, respectivelly. For example, Computational Tree Logic (CTL) is based on a tree model, while LTL is based on a linear model. LTL is sufficient to specify common properties of realistic systems.

The syntax of next-operator-free linear temporal logic (LTL) is:

$$\phi ::= \mathit{true} \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \Diamond\phi \mid \Box\phi$$

where p is a predicate the truth value of which can be decided at a particular state. An LTL formula can be evaluated with respect to a transition system. An LTL formula ϕ is true with respect to a transition system T (written as $T \models \phi$) iff it is true with respect to all the paths of T.

Given a method to decide an atomic proposition p with respect to a state (written as $s \models p$), the truth value of an LTL formula with respect to a path π (written as $\pi \models \phi$) can be defined as follows: (1) $\models$ *true*, (2) $s_0, \ldots \models p$ iff $s_0 \models p$, (3) $\pi \models \neg\phi$ iff $\pi \not\models \phi$, (4) $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$

and $\pi \models \phi_2$, (5) $\pi \models \Diamond\phi$ iff $\exists i.\pi^i \models \phi$, and (6) $\pi \models \Box\phi$ iff $\forall i.\pi^i \models \phi$. Intuitively, formula $\Box\phi$ means that ϕ is always true and formula $\Diamond\phi$ means that ϕ will eventually be true.

To provide an interpretation of a state, we use a labeling function $L : S \rightarrow 2^{AP}$ to associate a state with a set of atomic propositions (APs). For the purpose of model checking a program, we represent a state as a conjunction of the equations created by the valuation of all program variables (augmented by some artificial variables such as a program counter). Using this representation, $s \models p$ iff s implies p, where the latter s is the conjunction of the equations that characterize the state. When a state is represented this way, we call the transition relation R the (concrete) post operator.

The semantics of a program provides a direct but implicit definition of the transition relation R. Knowing the semantics is sufficient for us to model check a program, where finding all possible paths is often accomplished through a search algorithm. The search starts with states in S_0 and applies the post operator to find the next set of states. With a concrete representation of a state, the search algorithm has to consider numerous possibilities; this contributes to the state explosion problem with software model checking.

3.2.1 Expressive Power of LTL

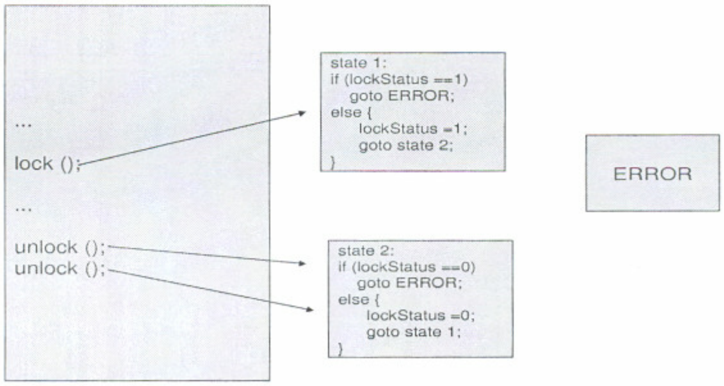
With LTL, we can express a lot of interesting properties of a system, some of which are discussed below.

Reachability

According to Berard et al. [3], reachability refers to properties that a particular location in the program will be reached. Often, the negation of a reachability property is of interest. For example, we may want to state that an error label in a C program will never be reached. This property is expressed by the LTL formula: $\Box(\mathtt{pc} \neq l)$.

Practically, we may choose to monitor a program's execution to translate a temporal property into a reachability problem. For example, the temporal logic for the correct use of a lock can be transformed into a reachability problem following these steps.

- Identify events. In the example, the initialization of a lock, the request of a lock, and the



```
...
lock ();
...
unlock ();
unlock ();
```

```
state 1:
if (lockStatus ==1)
   goto ERROR;
else {
     lockStatus =1;
     goto state 2;
}
```

```
state 2:
if (lockStatus ==0)
   goto ERROR;
else {
     lockStatus =0;
     goto state 1;
}
```

Figure 3.2: Reachability Problem for Correct Use of Locks

release of a lock are events. In C programs, events are often identified as the invocation of the corresponding functions.

- Respond to events by maintaining an auxiliary state variable. In our example, we may use a variable `lock_status` to hold the history of operations on the lock, namely to reflect that a lock is locked or unlocked. Note this variable is used for verification purposes only and has nothing to do with any runtime activities.

- Transfer control to an error label if the erroneous condition is satisfied. For example, if we try to acquire a lock when we already have it or try to release a lock which we do not possess,then an error label should be reached.

Figure 3.2 illustates how the correct use of locks is transformed into a reachability problem. The rectangle to the left represents a program, which to use locks through calling certain functions. The function calls are monitored. The logic of the monitor automaton is illustrated by the C-style code in the two rectangles in the middle. When error conditions are detected, an ERROR label will be reached.

Safety Properties

Safety properties refer to properties that state that an erroneous condition will never occur. In terms of temporal logic, this is $\Box(\neg p)$, where p is the error condition. Therefore, a reachability property is a special case of a safety property. On the other hand, the monitoring/instrumentation approach can be extended to safety properties, provided that the events are observable.

Specifically, the condition that violates the safety requirement can be expressed as a predicate over a machine state. In this case, safety properties can be expressed as a negation form of the reachability property. Thus they can be expressed in form of: $\Box(\neg p)$. Thus safety properties are more general than reachability properties.

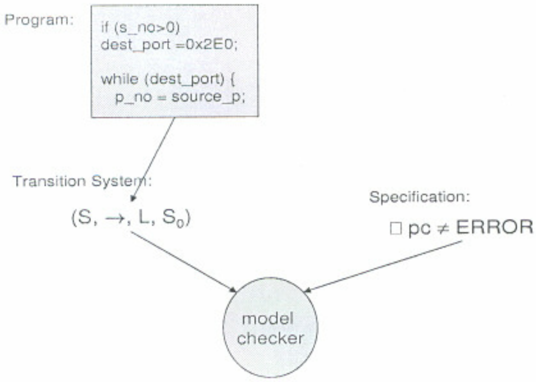
Liveness Properties

Safety properties assert that bad things will not happen. Sometimes, we expect the dual: good things must happen. Liveness properties state that a favorable condition will eventually happen in the future. In LTL, this is easily expressed by the $\Diamond$ operator.

For concurrent programs, liveness properties are typically verified with fairness assumptions. For example, a liveness property can easily be violated if a thread is never scheduled to run, even if the thread is not waiting on any condition and is eligible to run. Normally we assume at least a so-called weak fairness for a reasonable scheduler. In a concurrent system, a thread will be scheduled if the condition for it to be eligible to run holds and does not change.

3.3 Model Checking

To verify a temporal property on a model, for example, a labeled transition system, a model checker is invoked. Figure 3.3 illustrates the relationships between implementations, specifications, temporal properties and computational models. Between an implementation of a software system and its specification, two steps have to be taken before model checking can be applied. First, the specification is normally written in a language that is easier for a software designer to use, not always in temporal logic, so translation is necessary. For example, Object Constraint Language (OCL) elements or state diagrams in a UML design may be translated into temporal logic [37, 66]. Second, the implementation is in a programming language, which must be abstracted into the computational model accepted by a model checker. In some cases, model checkers will accept a compiled program as the input model [31], while, in general, model checkers expect a special input language close to the computational models introduced below.



```
if (s_no>0)
dest_port =0x2E0;

while (dest_port) {
  p_no = source_p;
```

Figure 3.3: Model Checking Process

Model checking with Büchi automata can be transformed into the test of emptiness of the language accepted by the product automaton (of the specification automaton and the property automaton). More specifically, for a system M and a property P, both expressed as Büchi automata, to determine if $M \models P$, it suffices to verify that $M \cap \neg P$ cannot accept any input.

Tools exist to translate LTL formulas into Büchi automata. For example, Spin comes with a built-in translator [36]. A faster algorithm is implemented in the tool ltl2ba [27].

3.3.1 Checking Emptiness of Synchronous Products

When we have a specification as a Büchi automaton A_P, and the LTL specification as another one A_ϕ, we can compute a new automaton that accepts exactly $L(P) \cap L(\neg\phi)$. This construction is called the synchronous product of two automata.

Essentially, each transition of the product automaton is a transition from the specification automaton and a transition from the property automaton. The final state of the product automaton is cleverly chosen to make sure that both the final state of the specification automaton and the final state of the property automaton are visited infinitely often [14]. A special case that often happens is that every state of the specification automaton is a final state, in that case, we only need to make sure the final state of the property automaton is met infinitely often.

Formally, a synchronous product of two Büchi automata $(\Sigma, Q_1, \delta_1, Q_1^0, F_1)$ and $(\Sigma, Q_2, \delta_2, Q_2^0, F_2)$ is a Büchi automaton $(\Sigma, Q_1 \times Q_2 \times \{0, 1, 2\}, \delta, Q_1^0 \times Q_2^0 \times \{0\}, Q_1 \times Q_2 \times \{2\}$.

We have $((r_i, q_j, x), a, (r_m, q_n, y)) \in \delta$ [14] if and only if:

- $(r_i, a, r_m) \in \delta_1$ and $(q_j, a, q_n) \in \delta_2$
- if $x = 0$ and $r_m \in F_1$, then $y = 1$
- if $x = 1$ and $q_n \in F_2$, then $y = 2$
- if $x = 2$ and $y = 0$

Algorithms based on either depth first or breadth first search can be applied to check whether the ω-acceptance condition of the product automaton is met. The absence of such a condition implies that the original LTL formula holds. In Spin [36], an embedded search algorithm is used for this testing.

The next subsection introduces a similar searching algorithm that can be applied directly on a labeled transition system to test reachability properties.

3.3.2 Checking Reachability

In the previous section, we showed that the absence of an erroneous event sequence can be transformed into a reachability problem. The instrumentation is a special case of the product construction. Namely, when an event occurs in the program, the event triggers a state transition in the monitoring automaton. Otherwise, a state transition in the program will be paired with a skip in the monitoring automaton. The only accepting state is the state that is labeled with an error. Therefore, the construction of a product automaton is important in the verification of reachability properties as well. Next we explain how we model check an instrumented program for reachability properties.

The algorithm for checking whether a label is reachable in a program is listed in Figure 3.4 [36]. We represent a program as a labeled transition system. The algorithm assumes a state representation without a stack. That is, we do not handle recursion. Extension of this algorithm to a pushdown system is introduced in Chapter 6. The representation is thus a valuation of program variables and a program counter. For source programs, the value of the program counter can be line numbers or labels, if unique labels are used throughout the program.

```
let S′ = S0;

while S′ is not empty do
  begin
    pick s in S′
    let pc = pc of s
    if pc = ERROR then return error_trace
    let S1 = post s J(pc)
    S′ = union (S′, S1)
  end
```

Figure 3.4: Explicit state model checking for reachability

For a labeled transition system, we define a post operator, which computes the set of next states given a state and a labeled transition. The post operator can be viewed as an interpreter of the program. J(pc) returns the label at pc. The post operator is also defined over the test of the if statement. The model checking algorithm is thus a search algorithm that starts from all possible initial states. Both depth first or breadth first search will be able to find out all the possible runs. The program terminates when the pc is valued as ERROR, or the search terminates after traversing all possibilities. When the ERROR label is reached, we may generate an error trace, showing the steps taken. This algorithm is called that of the explicit state model checking.

This algorithm applies to cases when the state is either abstract or concrete. How we represent a state abstractly is introduced in the next section.

3.4 Abstract Interpretation

This section reviews abstract interpretation, which is arguably the most important technique that reduces the complexity of the problem in software model checking. First we introduce the basic concepts of abstract interpretation (Section 3.4.1). Then we present an important abstract interpretation technique called predicate abstraction (Section 3.4.2). This technique is widely applied in software model checking.

In general, software model checking algorithms do not terminate. Part of the reason is that the state space of a piece of software is usually infinite. The state space involves the data space and the control space. Data space, comprised by variables, is huge if not infinite, which is computationally

```
int foo() {
   int x, int y;

   y = input_integer ();

   while (!(y %2))
      y = y / 2;

   if (y % 2) x = 5;
}
```

Figure 3.5: A Simple C Program

hard for a model checker to enumerate. In addition, recursion may occur in the control flow, resulting in an infinite control space, represented by a stack.

Consider a simple C program in Figure 3.5. It has two integer variables, x and y. If this program is running on a 64-bit machine, then data state space is as large as 2^{128}. However, if we know the property to check is $\Diamond(x = 5)$. Then we may try to consider only the parity of y. Thus the state space will be much smaller. Intuitively, we need to consider only four values, namely, one bit representing whether x is 5 and another representing whether y is even or odd, respectively (to be precise, we also need to consider the case when we do not have enough information, as shown below). Reducing the numerous concrete situations into a handful of cases is a problem handled by the abstract interpretation technique in program analysis [15].

3.4.1 Abstract Interpretation

According to Cousot and Cousot, abstract interpretation is a theory for approximating the semantics of discrete dynamic systems, for example, computations of programming languages [15]. The standard semantics are defined over the set of concrete values for program variables. Abstract interpretation provides a non-standard semantics that operates on a set of abstract values. The standard semantics for a conventional imperative statement can be defined as a function from concrete states to concrete states. The non-standard semantics is defined as a function from abstract states to abstract states.

The primary advantage of abstraction is that analysis will consider fewer possible values. A standard semantics may involve many possible values for program variables; an analysis, such as an explicit state model checking, if based on the standard semantics, works on this large state space. Using abstraction, we can work on a smaller set of abstract values. For example, we may replace the normal integer operations with abstract operations that only characterize the parity of the integers.

Formally, consider a concrete domain L and an abstract domain L_a. A concretization function γ maps a value in L_a to a value in L. An abstraction function α maps a value in L to a value in L_a. The following condition must be true.

$$\forall x \in L, x \leq \gamma(\alpha(x)) \tag{3.1}$$

In the application domain that we are interested in, we may regard the elements in L and L_a as sets of values, using the set inclusion as the $\leq$ relation. In this case, we can interpret the condition above as that the size of the image of L in L_a is less than or equal to the size of L. For example, in parity abstraction, we divide the integer values into two subsets, even numbers and odd numbers. The abstract domain should also include the empty set and the superset of the sets of even and odd numbers. The result is a lattice L_s containing 4 elements.

If the standard semantics (of a statement, or an unary expression) is a function f from L to L, then the corresponding non-standard semantics is a function f' from the lattice L_a to L_a. The following condition must be true for f' to be a valid abstraction of f.

$$\forall x \in L_a, f \circ \gamma(x) \leq \gamma \circ f'(x) \tag{3.2}$$

Consider function $f = \lambda x.(x + 1)$ and parity abstraction: the function f' changes the parity of the x. Equation (3.2) guarantees that the non-standard semantics should be consistent with the standard semantics.

Consider a `mod` operation between an odd number and an even number, there is no general way to tell the parity of the result. In fact, the result can be either odd or even. Such uncertainty derives from the approximation of the concrete value using abstract values. That is why the top element in the lattice L_s should be returned.

The non-standard semantics for adding two integers (in the parity abstraction) is presented in

$+$	Even	Odd
Even	Even	Odd
Odd	Odd	Even

Figure 3.6: Parity Abstraction: Semantics of Addition

a tablular form in Figure 3.6. The operation is defined over the values in the lattice.

An abstract interpretation can be formalized as a pair of functions (α, γ). We require the functions to be monotonic functions, and an adjunct (Galois connection):

$$x \leq \gamma \cdot \alpha(x) \quad y \leq \alpha \cdot \gamma(y)$$

From the concrete semantics we can construct a transition system over concrete values. From the abstraction we are able to define a transition system over abstract states. To define such an abstraction, we have to provide 1) a set of abstract values, 2) a function from concrete values to abstract values, and 3) a function from concrete state-transitions to abstract state transitions. The soundness of the abstraction (e.g., Equation (3.2)) will require that the successor of the concrete state will be abstracted into the successor of the corresponding abstract state.

While abstraction techniques such as the parity abstraction allow us to consider a smaller state space, they primarily reduce the state space of one variable. The large state space that we have to consider during verification often derives from the product of multiple variables. Thus, sometimes we expect to consider the abstract state space induced by relations among the variables, which reduces the state space as a whole. In software model checking, a technique called predicate abstraction is often successful.

3.4.2 Predicate Abstraction

Predicate abstraction [62] uses a lattice induced by a set of predicates over the variables in the program and the logic operators $\neg$, $\vee$, and $\wedge$, as the abstract state space. We can automatically compute an abstract post operator given a concrete one.

Formally, the concrete transition system T_c is (S, R_c, S_0), where R_c is implicitly defined by a program `prog`. Given a set of predicates $\Phi = \{p_1, p_2, \ldots, p_n\}$ over the program variables in `prog`, a predicate abstraction of P induces an abstract transition system $T_a = (S_a, R_a, S_{0a})$.

An abstract state is represented by a propositional formula over $B = \{b_1, b_2, \ldots, b_n\}$, abstract Boolean variables corresponding to $p_1, p_2, \ldots, p_n$, respectively. The abstraction function α maps a concrete state into an abstract state such that $\forall s \in S, s \models \alpha(s)[\Phi/B]$, where $P[\Phi/B]$ substitutes every b_i in P with corresponding p_i. This substitution defines a monotonic function γ that maps an abstract state into a set of concrete states. The abstract transition relation must satisfy: $\forall(m_1, m_2) \in R_c, \exists s_1, s_2 \in S_a$, such that: (1) $R_a(s_1, s_2)$, and (2) $m_1 \models \gamma(s_1)$ and $m_2 \models \gamma(s_2)$.

LTL formulas are preserved under predicate abstraction. Preservation is defined as $(T_a \models \phi) \Rightarrow (T_c \models \gamma(\phi))$, where symbol $\Rightarrow$ is implication. Clarke *et al.*'s textbook [14] proves ACTL (a superset of LTL) is preserved by abstraction. The properties certified by ACC systems are equivalent to LTL formulas. Thus, the significance of this result is that the ACC system is sound: if a temporal property holds for the abstraction, then it holds for the original program.

For a program written in an imperative language that contains assignments, `if` statements and `goto`'s, we can represent the result of Boolean abstraction as a Boolean program (BP) [4]. The translation can be done statement-by-statement.

A concrete assignment is translated into a simultaneous Boolean assignment to variables in B. To decide the value to assign to b_i, we compute the weakest precondition of every p_i, with respect to the statement `stmt`: $q = \mathrm{WP}(\texttt{stmt}, p_i)$. Then we find a formula over B that best approximates q: $F(q)$, that is, for any other formula r over B, $\gamma(r) \Rightarrow q$ implies $\gamma(r) \Rightarrow \gamma(F(q))$. This computation computes what condition in the previous abstract state will make the bit b_i to be true in the next abstract state. Repeat the process for $\neg p_i$ and all other p_i's. The simultaneous Boolean assignment is of the form:

$$\ldots, b_i, \ldots = \ldots, \texttt{choose}(F(\mathrm{WP}(\texttt{stmt}, p_i)), F(\mathrm{WP}(\texttt{stmt}, \neg p_i))), \ldots$$

The function $\texttt{choose}(e_1, e_2)$ is 1 (*true*) if e_1 is 1, 0 (*false*) if e_2 is 1, and `top` otherwise ($\texttt{choose}(1, 1)$ is in practice not possible). For example, if a C statement is `nPacket++`, and the predicate of concern (call it p_3) is `nPacket` = `nPacketOld`, then the corresponding Boolean program will include the assignment $b_3 = \texttt{choose}(0, b_3)$. This assignment reflects the fact that if the variables `nPacketOld` and `nPacket` are equal before the statement, then they will not be equal afterwards. The assignment never guarantees that the equality of the variables will be positively established. Since the algorithm is looking for the best approximation over B that establishes the truth of p_3, this assignment implies that the precondition necessary to establish p_3 (i.e. $\texttt{nPacket} + 1 = \texttt{nPacketOld}$) cannot be approximated by a propositional combination of Φ.

Information from the control logic of a program is represented by `assume` statements that introduce facts implied by the control behavior. An `assume`(P) statement makes P true in the next state. For example, a concrete `if` statement: `if (e) then` $\texttt{Stmt}_1$ `else` $\texttt{Stmt}_2$ is translated into (`st1` and `st2` are the translation of the two branches, respectively):

```
if  (*) then {assume(not F(not e)); st1}
         else {assume(not F(e)); st2}
```

Formula $(\neg F(\neg e))$ represents the strongest condition over B the concretization of which is implied by e. Symbol * denotes `top` and can be understood as non-deterministic choice.

We can verify that the Boolean programs defined in this way satisfy the soundness requirements of an abstract interpretation, in particular, Equation (3.2). In fact, we only need to ensure that a BP is a valid predicate abstraction. The third condition in the predicate abstraction ensures predicate abstraction satisfies the soundness Equation (3.2), when $\leq$ is interpreted as $\Rightarrow$.

Theorem 3.4.1 *For a concrete statement* `stmt` *and its BP translation* `bstmt`*, and any concrete state* s_1*, if after* `stmt` *is executed, the concrete state becomes* s_2*, then for any* s_{a1} *that* $s_1 \Rightarrow \gamma(s_{a1})$*, if the abstract state after* `bstmt` *is executed is* s_{a2}*, then* $s_2 \Rightarrow \gamma(s_{a2})$.

Proof: Consider the cases when the statement is an assignment and when it is an `if` statement.

- For an assignment, we consider every Boolean variable separately. For any b_i in s_{a2}, if it is *true*, then $s_1 \Rightarrow F(\texttt{WP}(\texttt{stmt}, p_i))$, then $s_2 \Rightarrow p_i$. Similar for the case when b_i is *false*in s_{a2}. For the case when b_i is * in s_{a2}, $s_2 \Rightarrow$ *true* trivially holds.

- For an `if`(e) statement, $s_2 = s_1$. Then the Boolean program will execute an `assume`(P) statement. $s_{a2} = s_{a1} \wedge P$. Without loss of generality, consider the then clause. In this case, $s_1 \Rightarrow e$, since $P = \neg F(\neg e)$, which is implied by e. So $s_2 \Rightarrow P$.

Other control forms of Boolean statements include explicit goto statements and function calls and returns. In ACCEPT/While, we use a model checker that cannot handle recursions, so the programs studied there do not contain recursions. In ACCEPT/C, the model checker a client uses can model check programs with restricted forms of recursive function calls. The programs we studied there may contain recursions. But because the model checking of programs with recursive calls is not decidable in general, the BPs computed there are mostly not recursive.

The next section discusses the issues relevant to revalidating an abstract model computed through predicate abstraction. The discussion is based on BP, but the arguments apply to predicate abstraction in general.

3.5 Reverifiable BP

In ACC, the essential information to be communicated from server to client is the abstract interpretation represented by the BP. The key operations on the client side are to verify the abstract interpretation and to verify the asserted property of the model. Type systems are a natural way to express the abstract interpretation in a mechanically verifiable manner.

There is a close relationship between type systems and abstract interpretations. The proof of soundness of a type system typically establishes that the domain of types is an abstraction of the domain of values. That is, the soundness of a type system guarantees that the type of a computation predicted by the type system is the type of the value that will be yielded by execution of the program.

Traditional type systems that focus on structural types and do not express any form of "dependence" correspond to a fixed abstract interpretation—the values are mapped to the domain of types. To support ACC it is necessary to specify the abstract domain. Index type systems, presented in detail in Section 3.5.1, provide a mechanism to tune the abstraction. In an index type system structural types can be refined by index propositions. These propositions effectively enrich the abstract domain when types are regarded as values in an abstract domain.

For example, to achieve the parity abstraction, the integers are refined by an index characterizing the parity of the values. All types involving integers must also be refined. A function that adds its integer arguments can then be assigned a type indicating that it maps arguments with different parities to odd numbers.

Index type systems are typically sufficiently expressive to capture all refinements that are discovered by the automatic predicate abstraction algorithms. For this reason ACC uses an index type system to verifiably represent the abstract interpretation used to calculate the BP.

The rest of the section introduces index types, its relation with abstract interpretation, and an index type system for SDTAL.

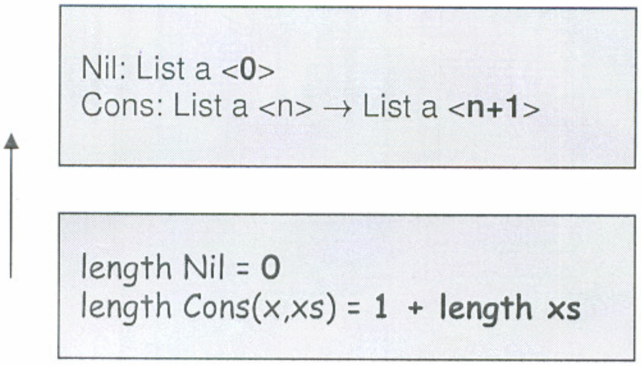
3.5.1 Background: Index Types

Index types are a variant of dependent types [41, 2, 30] introduced independently by Xi and Pfenning [71, 73] and by Zenger[76, 75]. Like dependent types, index types allow for a more precise characterization of the relationships between values than are traditionally expected of type systems. Unlike dependent types, index types are a refinement of another type system, which is refereed to as the base type system. A base type system is the host type system used in a programming language, for example, the type system for ML. The set of (untyped) programs typed by an index type system is a subset of the set of programs typed by the base type system.

Index types were originally introduced for functional programming languages. In an index type system, a data type can be refined by an index structure. For example, the type of list in a traditional functional language (refereed to as the host language) may be refined by an index that characterizes the length of the list.

```
data List a = Nil | Cons a (List a)
refine List a by int
where Nil  : List a <0>
      Cons : a (List a <n>) -> List a <n+1>
```

In the declaration above, the type (`List` a) in the host language is refined through the `refine` clause. The result is a cluster of types of the form `List` a $\langle e \rangle$, where e is an integer expression refereed to as an index expression. The `where` clause defines the index expressions that refine the types for values. The definition provides a way to construct index types. For example, the value `Nil` is of type `List` a $\langle 0 \rangle$. If a list is constructed through the application of `Cons`, then the type for this list is refined by an index expression $n + 1$, where n is the index that refines the type of the second argument of `Cons`. This definition is consistent with the meaning of the length of a list. Figure 3.7 illustrates the correspondence between a function `length` and the above refinement.



```
Nil: List a <0>
Cons: List a <n> → List a <n+1>
```

```
length Nil = 0
length Cons(x,xs) = 1 + length xs
```

Figure 3.7: Relation between a length function in the host language and the indices for a list type.

Index expressions are expressions over *index variables*. The index variables, for example n in the type declaration for `Cons`, are not program variables (as in dependent types), but can characterize relationships between run-time values.

After specifying how to refine a type through index expressions, we may specify index types for functions. These type declarations will be checked against the implementation to catch subtle bugs that are not usually caught by a type system. For example, the additive property of length under concatenation is expressed by the type:

```
concat: List a <m> -> List a <n> -> List a <m+n>
```

Index type checking uses the rules defined in the `where` clause to infer a type for an expression and to check the inferred type against any type annotations. For example, an erroneous implementation of `concat` is given below,

```
concat x y= y
```

This is type-correct in most type systems because the result is indeed a list. But an index type checker will assume that x is of type List a $\langle m \rangle$ and that y is of type List a $\langle n \rangle$, and hence infer that the result y is thus of type List a $\langle n \rangle$, which does not agree with the specification: List a $\langle m + n \rangle$.

Technically, this comparison is through the checking of subtyping and will introduce VCs to be discharged by theorem provers. In the above example, type List a $\langle n \rangle$ will be tested to see whether it is a subtype of type List a $\langle m + n \rangle$. Following a set of subtyping rules, VCs are generated. In the example above, a theorem prover has to test whether VC $\forall m.\forall n.m + n = n$ is true in an empty context. Because this condition is not true, type checking fails. Typically the constraints are logical formulas in a weak theory of arithmetic.

In summary, the extension of a host type system to index types includes four steps. First, an existing type system is refined. Second, a set of rules are described to infer types for program expressions. Third, the intended program properties are specified as index types. Fourth, a set of subtyping rules are given to facilitate the type checking. Type checking thus applies the rules specified in Step 2 to generate types and compares the inferred type with the type specified in Step 3; the comparison is the application of rules in Step 4. Below, we introduce common techniques used in each of the steps.

Refinement

A common refinement in index type systems is to construct the *singleton type* by refining a data type by itself.

```
data Nat = Z | S Nat
refine Nat by nat
where Z : Nat <0>
      S : Nat <m> -> Nat <m+1>
```

In this case the type system propagates symbolic information about the exact values of variables. In theory, this refinement technique applies to any algebraic data type. So far in practice, singleton types are often applied to integers.

Other refinement methods are concerned with other characteristics of types. For example, we may use integer indices with an intended meaning for the length of a list, the height of a tree, or the bound of a region. Again, in practice, indices are primarily integers.

The relation between values, for example between the result of a function and its arguments, is characterized by binding index expressions with their types. First, *index variables* are introduced. In a function type that does not involve high order function types, the index variables that are introduced in the types for arguments are considered universally quantified while those first appear in the result are existential. Second, arithmetic expressions over the index variables implicitly describe the relation between values. For example, the successor constructor S above has type Nat $\langle m \rangle \rightarrow$ Nat $\langle m + 1 \rangle$, which specifies that the result is larger than the argument by 1. The forms of index expressions are often limited to those allowed in a simple arithmetic theory that has a decision procedure, for example, Presburger arithmetic.

The relations between values can be made explicit by introducing an index variable in the place of an index expression and allowing *index propositions* to constrain complex types such as function types or tuple types. For example, the constructor S above can have type:

```
        S: Nat <m> -> Nat <n> | n = m+1
```

Here $n = m+1$ is the index proposition that constrains the function type before the separator |. An indexed type tuple representing a machine state is particularly useful when applying index type techniques to procedural languages, including intermediate or assembly languages.

There is a connection between index types and predicate abstraction. The index type for a machine state can be a tuple type constrained by index propositions. An index type that corresponds to an abstract state s_a in predicate abstraction is the case where the index propositions are the predicates that are true in the concrete state $\gamma(s_a)$. For example, for a program that involves variables x, y and z and a predicate abstraction over predicate $x = y$. The two possible index types are:

```
(int <x>, int <y>, int <z>) | x=y
```

and

```
(int <x>, int <y>, int <z>) | x!=y
```

Type Inference

To discuss type inference in index type checking, we may distinguish two different categories of applications: applications where application programmers are allowed to specify application-specific refinements, and applications where system-wide language-specific refinements are enough for the specification of properties.

In some applications, it is desirable to allow programmers to choose an intended meaning of the indices: they may specify refinement through a set of index type construction rules. The construction rules focus on the computation of index expressions. For an algebraic type in a functional language, the computation rules are specified for each constructor of the algebraic type, as shown in the list example above. Then a programmer can also provide an application-specific property, such as the one for `concat` above. The inference of types, as illustrated above, involves constructing (and simplifying) index expressions using the rules in refining the `list` type.

In other cases, the properties to be checked are universal for all the programs written in the host language; the refinement of types is system-wide; there is no need for a programmer to specify how a base type is refined. A programmer, or in most cases, an automatic tool, simply uses an existing indexed type system to specify and verify properties of a program. In this case, the primary way to infer a type is through system-wide index expression construction rules. This manipulation is normally a lifting of program analysis, which manipulates values in a program, to the index world. The designer of the index type system has to provide type inference rules that reflect the semantics of the host language. This approach has been applied in our early work in ensuring safe array access in a Java bytecode-like language. Later in this dissertation, we will see how the same approach is used to validate predicate abstractions.

As discussed earlier, to design type inference rules for a language, a common approach is first to study the logic in which the analysis is performed, and then to lift this logic into the index domain. In validating predicate abstractions, the logic of the analysis is the program logic that allows the specification and computation of pre- and post- conditions, which will be introduced in Section 3.6.1. In the list length example, the function `length` can be used by a simpler analyzer. The type inference rules that compute indices that represent the length of the list are the definition of the `length` function lifted into the index world.

VC Generation

After an index type is inferred, the type checker may compare this inferred type with the specified type to decide type correctness. A common way this is implemented is through a set of subtyping rules. Such rules typically generate VCs to decide whether the inferred type is a subtype of the specified type. An common rule for subtyping is as follows:

$$\frac{(P_1 \Rightarrow P_2) \quad \tau_1 \sqsubseteq \tau_2}{(\tau_1|P_1) \sqsubseteq (\tau_2|P_2)} \text{ (tc-subtype)}$$

Here the proposition $P_1 \Rightarrow P_2$ is a verification condition.

Below, we demonstrate the construction of an index type system for an intermediate language SDTAL. First we present a logic over SDTAL terms that is sufficient to express the computation of

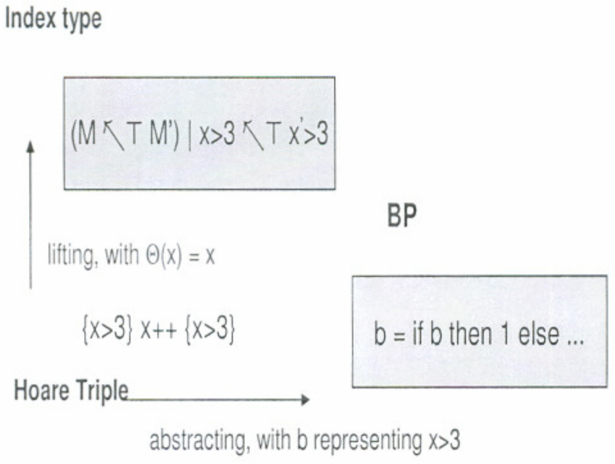


Figure 3.8: Relationship between LAP, Boolean expressions and index propositions

a BP. Then we lift this logic into the index domain. This way we construct the index type system of SDTAL's.

3.6 Index Types in SDTAL

The logic presented in this chapter is the logic used in specifying the abstraction bases in predicate abstraction. We call this the Logic for Atomic Propositions (LAP).

3.6.1 LAP: A Logic

Figure 3.8 illustrates the relation between Boolean expressions, the program logic LAP, and index propositions in SDTAL. LAP is the programming logic that specifies the predicates in predicate abstraction. As discussed, abstraction and concretization functions α and γ, described in Section 3.4, map between a LAP expression and a Boolean expression. For example, a LAP predicate $x > (y + 3)$ will be used to compute a BP. This predicate corresponds to a variable b_1 in the resultant BP. The variables mentioned, namely x and y, are program variables in the concrete program. A proposition in LAP may be lifted into an index domain to form an index proposition, which is over index variables. To define the lifting, we have to define a mapping between program variables and index variables. The convention in this dissertation refers to this mapping as θ.

$$\begin{array}{lll l}
\text{expressions} & e & ::= & x \mid c \mid e_1 + e_2 \mid e_1 - e_2 \ldots \\
\text{variables} & x & & \\
\text{propositions} & p & ::= & e_1 = e_2 \mid e_1 \leq e_2 \ldots \\
 & & & \neg p \mid p_1 \vee p_2 \mid p_1 \wedge p_2 \ldots
\end{array}$$

Figure 3.9: Syntax of LAP

Both the syntax and evaluation of the LAP are that of a common program logic. The syntax of LAP is listed in Figure 3.9. A LAP proposition is evaluated in a way similar to that defined in Section C.4.

3.6.2 SDTAL, an Index Typed Assembly Language

The application of index types to target languages was explored by Xi in separate collaborations with Xia and Harper [74, 72]. Xi and Xia explore the elimination of unnecessary array bounds checks in a setting based on the JVM.

The SDTAL language is a typed intermediate language that has a type system very similar to those mentioned above. Fig. 3.10 presents the syntax of SDTAL, which will be the target language for ACCEPT/While.

SDTAL has a simple instruction set that is sufficient to compile a simple imperative language. There are three kinds of instructions.

- Arithmetic instructions: operations take operands from registers or constants. The registers are $r_0, r_1, ..., r_n$. A variable has to be loaded into a register before an operation such as addition is performed. SDTAL has integer operands only.

- Load and store: these are the only instructions that access the memory, where variables are stored.

- Transition instructions: there is an unconditional `goto` instruction and several conditional branch instructions. Conditioanl branch instructions take two registers, compare them (but keep the contents intact) and transfer the program pointer according to the result of comparison.

labels	l		
registers	$R_1, \ldots, R_n$		
index variables	x, y		
index expressions	e	$::=$	$x \mid c \mid \texttt{ic} \mid e_1 + e_2 \mid e_1 - e_2 \ldots$
index propositions	P	$::=$	$e_1 = e_2 \mid e_1 \leq e_2 \ldots$
			$\neg P \mid P_1 \vee P_2 \mid \ldots$
type constants	$\texttt{tc}$	$::=$	$\texttt{int} \mid \ldots$
types	τ	$::=$	$\texttt{tc}(x) \mid (\tau_1, \tau_2)$
		$::=$	$\mid \tau_1 \rightarrow \tau_2 \mid (\tau \mid P) \ldots$
instructions	$\texttt{ins}$	$::=$	$\texttt{add}\, R_1\ R_2\ R_3 \mid \texttt{sub}\, R_1\ R_2\ R_3 \ldots$
			$\texttt{load}\,\texttt{var}\, R \mid \texttt{store}\, R\, \texttt{var}$
			$\mid \texttt{jcond}\, l_1\ l_2\ R_1\ R_2 \mid \texttt{mov}\, R_1\ R_2$
ins sequence	I	$::=$	$\texttt{ins}^*$

Figure 3.10: Syntax of SDTAL

Type State

The constraint logic of the index system is built in SDTAL. Constraint logic propositions are the index propositions used to constrain a type in an index type system. The syntax of index expressions and propositions are very similar to that in the LAP.

It is necessary to associate variables and registers with index variables. To reduce the validation of BPs to extended type checking, the type system is constructed to characterize the side effects of the program execution. The effects of interest are the memory effects of `load` and `store` operations on observable variables. To characterize these, a type system also characterizes the relevant registers. The notion of *type state*, introduced by Xi and Xia, represents the registers and memory as a large tuple type. Each component type in this tuple is refined to be a singleton type. Syntactically, we write an *unconstrained type state* M as this tuple of singleton types, each of which is refined by an index variable. A variable x or a register r_i is of a type refined by $\theta(x)$ (or $\theta(r_i)$).

An index proposition over these index variables thus characterizes the relations among program variables or registers. An index type state is written as $(M|P)$, where P is an index proposition.

Indexed Function Types

Conceptually, instructions are functions from one type state to another. Every block in the target language is given a function type characterizing its effects on the type state. For example, $((M|P \to M')|Q)$ is such a type, where P is an index proposition over the index variables in a tuple M, while Q is an index proposition over the index variables in two tuples M and M'. Unconstrained types M and M' are different only in the index variable names. By convention, we write M' such that, if x is the index variable that refines a component type in M, then x' is an index variable that refines the corresponding type in M'.

To check such function types, it is necessary to compute index types that capture post-conditions. This computation is the lifting of the computation of the post-condition in the standard programming logic to a computation in the index domain. The post condition of $P(x)$ with respect to the instruction add r_0 r_1 r_2 is:

$$Q = \exists x.P[x/r_2] \wedge (r_2 = r_1 + r_0)$$

Lifting Q to the index domain, an index type for the add instruction is $(((M|P) \to M)|Q[\theta])$, where θ maps a register or a variable to the index variable that refines its type; $Q[\theta]$ is the application of this mapping. Section 3.6.4 gives a detailed account of the index type checking in SDTAL.

The concepts of type state and indexed function types were designed for the verification of memory safety. In ACC, we show these concepts can be conveniently applied to encode and verify BPs. Reusing the type checker in this way is a software engineering benefit. Furthermore, that a function type encodes BP statements makes the validation modular: the sizes of the index type expressions are bounded because only a limited number of predicates are affected.

3.6.3 Dynamic Semantics

The dynamic semantics of SDTAL is given as a small-step structured operational semantics, a fragment of the definition is presented in Fig. 3.11. The semantics defines the transition relation, $\to$, which induces a transition system. The state of the machine is given by the control memory (J), the instruction counter (ic), and the state of registers and memory (Mem). The model of the memory assumes a finite number of registers (n) and an unbounded number of variables. For example, Mem(i) represents the i-th register and Mem(x) represents the value of x. Updates to

$$\frac{J(ic) = \mathtt{add}\, r_a\, r_b\, r_i}{(ic, \mathtt{Mem}) \rightarrow (ic + 1, \mathtt{Mem}[i \mapsto (\mathtt{Mem}(a) + \mathtt{Mem}(b))])}\ \text{(eval-add)}$$

$$\frac{J(ic) = \mathtt{sub}\, r_a\, r_b\, r_i}{(ic, \mathtt{Mem}) \rightarrow (ic + 1, \mathtt{Mem}[i \mapsto (\mathtt{Mem}(a) - \mathtt{Mem}(b))])}\ \text{(eval-sub)}$$

$$\frac{J(ic) = \mathtt{mul}\, r_a\, r_b\, r_i}{(ic, \mathtt{Mem}) \rightarrow (ic + 1, \mathtt{Mem}[i \mapsto (\mathtt{Mem}(a) * \mathtt{Mem}(b))])}\ \text{(eval-mul)}$$

$$\frac{J(ic) = \mathtt{store}\, r_i\, \mathtt{var}}{(ic, \mathtt{Mem}) \rightarrow (ic + 1, \mathtt{Mem}[\mathtt{var} \mapsto \mathtt{Mem}(i)])}\ \text{(eval-store)}$$

$$\frac{J(ic) = \mathtt{load}\, \mathtt{var}\, r_i}{(ic, \mathtt{Mem}) \rightarrow (ic + 1, \mathtt{Mem}[i \mapsto \mathtt{Mem}(\mathtt{var})])}\ \text{(eval-load)}$$

$$\frac{J(ic) = \mathtt{mov}\, r_i\, r_j}{(ic, \mathtt{Mem}) \rightarrow (ic + 1, \mathtt{Mem}[j \mapsto \mathtt{Mem}(i)])}\ \text{(eval-mov)}$$

$$\frac{J(ic) = \mathtt{jmp}\, l}{(ic, \mathtt{Mem}) \rightarrow (l, \mathtt{Mem})}\ \text{(eval-jump)}$$

$$\frac{J(ic) = \mathtt{jeq}\, l_1\, l_2\, r_i\, r_j \quad \mathtt{Mem}(i) = \mathtt{Mem}(j)}{(ic, \mathtt{Mem}) \rightarrow (l_1, \mathtt{Mem})}\ \text{(eval-jeq-yes)}$$

$$\frac{J(ic) = \mathtt{jeq}\, l_1\, l_2\, r_i\, r_j \quad \mathtt{Mem}(i) \neq \mathtt{Mem}(j)}{(ic, \mathtt{Mem}) \rightarrow (l_2, \mathtt{Mem})}\ \text{(eval-jeq-no)}$$

Figure 3.11: Evaluation Rules of SDTAL

variables and registers are written $\mathtt{Mem}[u \mapsto v]$, where u is a register or variable and v is the new value. If Q is a proposition over registers and program variables, $\mathtt{Mem}[Q]$ indicates that Q is true under the variable and register assignment Mem. The instruction at location l is indicated by $J(l)$.

The dynamic semantics is presented as a set of evaluation rules in Fig. 3.11. A typical rule is the (eval-add) rule, the antecedent of which indicates that the current instruction is an addition. The consequent is of the form $(ic, \mathtt{Mem}) \rightarrow (ic', \mathtt{Mem}')$, representing one step of execution. This rule demands that the ic will be incremented by 1 and that the target register is updated to hold the sum. The semantics of conditional transitions are illustrated by two rules for the `jeq` instruction. We define $\rightarrow^*$ as the reflexive transitive closure of the $\rightarrow$ relation.

3.6.4 Type Checking

The goal of the type checker is to check the type annotations. As mentioned, annotations can be either a type state assertion or a function type. Accordingly, the type checker can be decomposed into two aspects. Checking a type state is presented in the subsection entitled "Type State Annotations" below. To type check a function type, the type checker will first compute an inferred type characterizing the semantics of intermediate instructions (described in the subsection entitled "Inferred Types" below) and then test to see if the inferred type is a subtype of the annotated type (described in the subsection entitled "Subtyping").

Inferred Types

In a general setting (not just for state-transformer types), an index type for an instruction sequence I is written as I: $((M|Q_1) \rightarrow N)|Q_2$. If I is the body of an integer incrementation function, M is an unconstrained type for the arguments (for example $\texttt{int}(i)$), and N is an unconstrained type for the return value (for example $\texttt{int}(\texttt{ret})$). Index propositions P and Q specify the pre- and postconditions (for example that $\texttt{ret} > i$). The type inference procedure is to apply the semantics of the program to compute a postcondition (in our example, that $\texttt{ret} = i + 1$).

For a sequence of instructions $I : j_0, j_1, ..., j_n$, we refer to the corresponding unconstrained type states as $M^0, M^1, ..., M^{n+1}$. They are different from each other only in the name of index variables. To infer the function type, we compute an index proposition over (M^0, M^i) and call it a *transient proposition*. We refer to the transient propositions as $Q^1, ..., Q^{n+1}$.

Relation $\twoheadrightarrow$ is defined between Q^i and Q^{i+1}, possibly superscripted by the instruction. We assume I is free of transition instructions. This assumption is valid for the validation of BP because such validation happens within basic blocks. In the general case, every path is checked, which relies on loop invariants. Loop invariants are treated as type state assertions.

For every non-transition instruction, we define a characteristic assignment of the form $y = e(X)$. For example, the characteristic assignment for $\texttt{store}\ r_1\ \texttt{var}$ is $\texttt{var} = v_1$, where $\texttt{var}$ and v_1 are the index variables for $\texttt{var}$ and r_1, respectively.

$$\frac{Q(X^0, X^{i+1}) = P(X^0, X^i)[\theta] \wedge \texttt{same}(X - \{y\}) \wedge y_{i+1} = e(X_i)}{P(X^0, X^i) \twoheadrightarrow^{j_i} Q(X^0, X^{i+1})}\ (post)$$

Where θ is a substitution defined as follows: if y is in X_0, then θ is empty, otherwise it is $y_i \mapsto \texttt{newvar}$. Meta function $\texttt{same}(X)$ generates a set of equations between x^i and x^{i+1}. In the implementation, careful (and involved) handling of the variables is required to avoid the trivial equations introduced by $\texttt{same}$.

For example, for the instruction sequence $\texttt{load}\, r_0\ x$; $\texttt{add}\, r_0\ 1\ r_0$; $\texttt{store}\, r_0\ x$, and a precondition $true$, we have Q_0 as *true*, Q_1 as $(v_0^1 = x^0 \wedge x^0 = x^1)$, Q_2 as: $(v_0^2 = v_0^1 + 1 \wedge v_0^1 = x^0 \wedge x^0 = x^1 \wedge x^2 = x^1)$, and Q_3 as: $(v_0^2 = v_0^1 + 1 \wedge v_0^1 = x^0 \wedge x^0 = x^1 \wedge x^2 = x^1 \wedge x^3 = v_0^2)$. The final transient proposition is used to construct a function type for the sequence as: $(M^0 \rightarrow M^3)|Q_3$.

Subtyping

For types $\sigma_1 = (\tau_1|P_1)$ and $\sigma_2 = (\tau_2|P_2)$, we test (recursively) if τ_1 is a subtype of τ_2 and if P_1 implies P_2 to determine whether σ_1 is a subtype of σ_2. This general case is as follows:

$$\frac{(P_1 \Rightarrow P_2) \quad \tau_1 \sqsubseteq \tau_2}{(\tau_1|P_1) \sqsubseteq (\tau_2|P_2)} \text{ (tc-subtype)}$$

For function types σ_1 and σ_2, $\sigma_1 \sqsubseteq \sigma_2$ if σ_1's domain is a super-type of that of σ_2, and σ_1's range is a subtype of that of σ_2, as in the rule below, which can be intuitively interpreted as the precondition strengthening and the postcondition weakening.

$$\frac{\tau_3 \sqsubseteq \tau_1 \quad \tau_2 \sqsubseteq \tau_4}{(\tau_1 \rightarrow \tau_2) \sqsubseteq (\tau_3 \rightarrow \tau_4)} \text{ (subtype-func)}$$

In the running example, the infered type is compared to the annotated type: $(M \rightarrow M')|x' = x + 1$, we generate the verification condition: $(v_0^2 = v_0^1 + 1 \wedge v_0^1 = x^0 \wedge x^0 = x^1 \wedge x^2 = x^1 \wedge x^3 = v_0^2) \Rightarrow x_3 = x_0 + 1$.

Type State Annotations

SDTAL programs may have type assertions at a label that specifies the property of the machine state when control reaches the label. To simplify the presentation, we restrict the locations that can have an type state assertion to the beginning labels of blocks. For example, a typical block B in SDTAL is:

$$(l_B : (M|P),\ I_1 : ((M \rightarrow M')|Q); \texttt{jeq}\, l_1\ l_2\ v_1\ v_2)$$

In this block, P is a condition that must be established before control is transferred to the starting label l_B, Q is a postcondition consistent with the inferred postcondition of the body I_1, and l is a label the precondition of which, τ_l, must be established by $Q \wedge (r_1 = r_2)$. In addition, the continuation of this block, which must be a block, must have a precondition implied by $Q \wedge (r_1 \neq$

r_2). This is expressed in the rule below, which checks a block ending with a jeq. Meta function $\mathcal{L}$ returns the assertion at a given label.

$$\frac{\vdash I_1 : \tau \quad P \twoheadrightarrow Q' \quad (M|Q' \wedge x_1 = x_2) \sqsubseteq \mathcal{L}(l_1) \quad (M|Q' \wedge x_1 \neq x_2) \sqsubseteq \mathcal{L}(l_2)}{\vdash B = (l_B : (M|P), I_1 : (\tau = ((M \rightarrow M')|Q)); \texttt{jeq}\, r_1\, r_2\, l_1\, l_2)} \text{ (jeq)}$$

Given a program $\texttt{Prog}_D$, if for every block K_i, $\vdash K_i$, then the program is well-typed.

An analysis of the type checking rules reveals that the complexity of type checking, in terms of theorem prover calls, is linear in the size of the program. The worst case complexity of each VC is exponential in the number of implications being checked. However, in reality, the form of the VC is in general simple, because when applying index types to represent a BP, the fact that the program has been model checked sets a computational bound for the type checking.

3.6.5 Soundness

In this subsection we state the soundness of the typing rules of SDTAL with respect to the $\rightarrow$ transition semantics. The soundness theorem states that if a program is well typed, and if the machine moves from one state to another by executing a block, then these two states satisfy the specified post type state; and the computed post state also satisfies the type state at the entry of the a target block.

Specifically, (s, s') is a product of two machine states. They as a whole can be viewed as a mapping from registers and variables to integer values. θ is a substitution mapping integer variables to the corresponding index variables. Therefore, $(s, s')[\theta]$ is a mapping from index variables (from both states) to integer values. For example, if $s(r_0) = 0$ and $s'(r_0) = 1$, then $(s, s')[\theta](x_0) = 0$ and $(s, s')[\theta](x'_0) = 1$. $(s, s')[\theta] \models P$ if P is evaluated to be true in the integer domain, given the mapping of $(s, s')[\theta]$. For example, in the example above, we have: $(s, s')[\theta] \models x'_0 = x_0 + 1$.

This local soundness theorem validates a state transformer type.

Theorem 3.6.1 *For an instruction sequence I: $((M|P) \rightarrow M')|Q$ in a well typed SDTAL program, and for every machine run such that $s \rightarrow^I s'$, if $s[\theta] \models P$, then $(s, s')[\theta] \models Q$.*

This global soundness theorem validates the type state asserted at various program locations.

Theorem 3.6.2 *Let* $\texttt{Prog}_D$ *be a well typed program composed of blocks* K_i*'s. Let the label of* K_i *be* l_i*, the type state at the label* l_i *is* $M|P_i$*, and the state transformer type of body of* K_i *be* $(M \rightarrow M')|Q_i$*. For any initial machine state* s_0*, if* $s_0 \rightarrow^* s'$*, with* $s' \models ic = l_j$*, then either the machine stops, or* $s'[\theta] \models P_j$*.*

First, we prove the local soundness theorem.

Recall that the welltypedness requires that $\vdash I : \tau$, where τ is $((M|P) \rightarrow M')|Q$. Also recall that this judgment is done through two-steps, first there must exists a P_1 such that $P \twoheadrightarrow P_1$ and $P_1 \Rightarrow Q$. So to prove the above theorem we can prove the following lemma:

Theorem 3.6.3 *For an instruction* `instr`*, and for every machine run such that* $s_1 \rightarrow^{\texttt{instr}} s_2$*, if* $(s, s_1)[\theta] \models P$*, and* $P \twoheadrightarrow P_1$*, then* $(s, s_2)[\theta] \models P_1$*.*

Proof: Induction on the form of the instruction `instr`.

- `instr` is `add` $r_i\ r_j\ r_k$. In this case, P_1 is $\exists x_k.P(V^2/V^1) \wedge x_k^2 = x_i^2 + x_j^2)$, where V^1 is the index variables superscripted by 1. Take into account the evaluation rules: s_2 is different than s_1 only in r_k. Thus $(s, s_2) \models (r_k^2 = r_i^2 + r_j^2)$, according to the evaluation rules. More directly, we have $(s, s_2)[\theta] \models \exists x_k.P[V_2/V_1]$.

- The cases of instructions `sub` and `mul` are similar to that of `add`'s.

- `instr` is `load` r_i `var`. In this case, P_1 is $\exists \texttt{nv}.P(\texttt{nv}/\texttt{var}) \wedge \texttt{same}(V - \{\texttt{var}\}) \wedge \texttt{var} = v_i$. According to the evaluations rules, $\texttt{same}(V - \{\texttt{var}\})$ is true; so is $\texttt{var} = v_i$. Because $P(V)$ is true over s and s', so $\exists \texttt{nv}.P[\texttt{nv}/\texttt{var}]$ also holds over s and s'.

- `instr` is `store` r_i `var`. This case is similar to that of loads.

The global soundness can be proved by induction on the number of steps. The induction step is proven through the definition of well-typedness: that the target machine type at $i + 1$ step is implied by the disjunction of all machine state at i-th step.

Chapter 4

Abstraction-carrying Code

This chapter presents the general framework of ACC and discusses the pros and cons of this approach.

4.1 Abstraction Carrying Code: Framework

Figure 4.1 illustrates the relations between data artifacts in a general ACC framework. A source program is abstracted into an abstract model, with a reduced state space. Model checking on the abstract model is successful. Then the server compiles the source program into an intermediate program; the source program and the abstract model are transformed together so that the abstract model is a valid abstraction of the compiled code and the temporal property is still model checked. The abstract model is then encoded and attached to the compiled program. A client validates the abstract model and then model checks the temporal property.

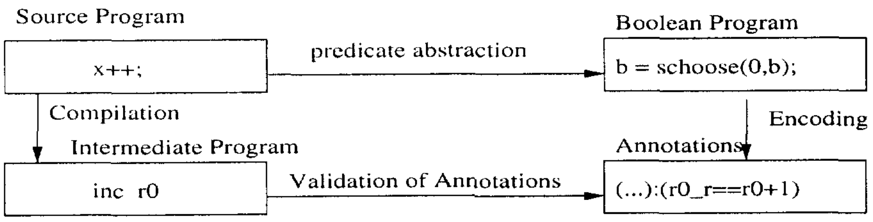


Figure 4.1: Relationship Among Data Artifacts in an ACC System

4.2 Abstraction Carrying Code: Design Choices

The framework described above leaves much room for design choices. We divide an ACC system into the following parts:

- An abstractor of the mobile program;
- A certifying compiler;
- An encoder that encodes the abstraction;
- A validation of the abstraction;
- A decoder of the abstraction; and
- A model checker of the abstraction.

This section discusses possible design choices for each of the components above. Such choices are faced by an implementor of an ACC system. Because the validation of the abstraction is strongly tied with the way an abstraction is represented (encoded), I will discuss these two components together. Throughout the discussion, I will refer to the example from Section 1.2 to demonstrate the concepts.

4.2.1 Abstraction

Various abstraction techniques exist to reduce the size of the state space, some reducing the control state space and some the data space. There are different levels of automation as well. For example, abstract interpretation techniques can automatically compute an abstract model. But the decision on which abstraction method to apply for a particular verification task requires human guidance. Predicate abstraction, combined with counter-example driven predicate discovery, can automatically discover and perform the abstraction necessary to model check a property. However, whether this approach works for concurrent programs is still under investigation [56]. A combination of automated software model checking and theorem proving can verify a selected class of properties, such as race-condition freedom for certain, structured concurrent programs. In general, we can conclude that automated predicate abstraction is our best approach, and that human selected abstraction techniques are also helpful for general temporal properties of concurrent programs.

Even for a predicate abstraction, there are different levels of precision in computing and representing the abstract transition relation. Each of these levels has different computational resources needs. For example, computing a precise predicate abstraction takes more computational resources than a simplified abstraction, where the precision is compromised for efficiency. Research has shown that BPs are effective in model checking API safety properties for industrial strength programs [4]. The example presented in Section 1.2 is an example of the application of the counter-example driven technique to discover predicates in Boolean abstraction.

4.2.2 Compilation

A certifying compiler refers to a compiler that also generates a certificate for a property. It is implied that the property has been verified for the source program and that compilation preserves the property.

The idea of property preservation during compilation can be viewed as a design choice because we may choose to certify a compiled program directly. It has been generally agreed that many properties are much more easily verified on the source program. For example, PCC takes the certifying compilation approach because type safety is obvious and data flow analysis is less complicated for source programs. Similar wisdom applies to the verification of temporal properties: more predicates are involved if we attempt to model check a compiled program through predicate abstraction. For example, consider `nPacket++` in the example. if we compile it into the following instruction sequence:

```
mov   nPacket r0
inc   r0
mov   r0 nPacket
```

We need additional information about r0 to keep track of the relationship between `nPacket`and `nPacketOld`, which is vital to the correctness of the example.

In theory, the certifying compiler needs to preserve the properties of concern, which in the example is the correct use of the locks. In ACC, because we generate an abstract model as a certificate, the compilation focuses on the preservation of temporal properties on the abstract model. In the simplest case, the compiler will try to retain the same abstract model for the compiled code. That is, the model for the source will still be a valid abstraction of the target program. The challenge for the compiler is to preserve the abstract model. Surprisingly, the techniques studied by Necula in certifying compilation apply to the certifying compiler in ACC. The reason is that Necula's certifying compilation attempts to preserve dataflow annotations such as the pre- and post-conditions or loop invariants. The same is needed in ACC at a finer granularity: the BPs are essentially specifications of pre- and post- conditions. These pre- and post- conditions do not consider global dataflow facts. A BP statement that translates a source statement will be associated with a block of instructions in the compiled program.

This simple compiler may perform optimizations. Some of the optimizations may transform the BPs in parallel in transforming the intermediate code. As studied by Necula, a slightly limited class of optimizations can be performed. Many of his optimization techniques can be used in our system. Furthermore, when the efficiency of the compiled code is of concern, a client may choose to optimize the intermediate code after reverification of temporal properties, although this would require a trusted optimizer.

The example in Section 1.2 is not optimized.

4.2.3 Encoding Abstraction/Model Re-validation

As mentioned, the abstraction can be viewed as localized pre and post conditions (Hoare triples): revalidation can focus on the block that the BP statement is associated with. For example, the Block I in the example is associated with Hoare triple: $\{\texttt{nPacket} = \texttt{nPacketOld}\}I\{\texttt{nPacket} \leq \texttt{nPacketOld}\}$, Encoding the abstraction is to encode the Hoare triples. The simplest way is to write annotations in terms of the safety logic. In reality, we may also consider other verification tasks that co-exist with the validation of the abstract model. In particular, as will be introduced in Chapter 8, memory safety plays an important role in the correctness of Boolean abstraction. Thus we need a general approach that can reason about pre- and post- conditions as well as memory safety that also involves the bound of a piece of allocated memory. For that reason and other engineering considerations, ACC uses a type system called the index type system. The model revalidation is through the type checking of the index type system.

4.2.4 Property Reverification

Once the BPs are validated, we may use a model checker to verify the temporal property. There are again multiple possibilities. First, we may use a model checker for BP. For example, moped [63] and bebop [5] are designed specifically for BPs and run very fast. The problem with this approach is that these tools lack the support for concurrent programs. Thus we may consider the second choice, which is to translate the BP to an input language of a general purpose model checker. For example, in ACCEPT/While we translate the BP plus some concurrent primitives to Promela [36] programs. In this approach, we have to ignore recursion.

It is worth mentioning that model checking is not the only way a property can be reverified. An alternative is for the server to use a proof-generating model checker to generate a proof. Then the client may simply check the proof to reverify the property. Because the proof is on the abstract model, it may not be as large as a proof for the original program.

4.3 Pros and Cons

Evidence-based software certification is a compelling idea that we expect to contribute to a dramatic increase in the safety and robustness of software in use by society. Abstraction-carrying code is a novel point in the design space of evidence-based techniques. There are several important dimensions of this design space, including the expressive power of the properties to be certified, the size of the trusted code base necessary to validate certificates, the cost of certificate construction, and the cost of certificate validation. Our investigation focused on a relatively expressive class of certificates inspired by specifications that are useful in industrial practice. In the context of this expressive class of certificates, we have explored the costs of certificate construction and validation.

The decomposition of the problem of verification of temporal properties of intermediate representations of mobile code into certificate construction, model validation and model checking contribute to the scalability. By using an abstraction as a certificate, both certification and model validation are localized to fractions of the program (a function or a statement). The global computation is performed by the model checker. Because model checking an abstract model is much faster than checking a source program, the client-side operations are expected to be orders of magnitude faster than the server side operations. Thus we expect the client-side operations to remain acceptably fast for any problem on which server-side verification is tractable.

A further implication of the modularity of ACC is the separation between a property of concern and the VCGen. In ACC, a VCGen verifies Hoare triples. Writing such a VCGen requires only the knowledge of the semantics of the intermediate language. No knowledge of the safety policy is needed. In contrast, in the original PCC systems, the VCGen hard-codes the safety policy. A direct consequence is that once the policy changes, a system such as Touchstone needs to rewrite its VCGen, while there is no such need in ACC. Although with ACC, the abstract models depend on the properties being certified, we may hope that the size of a model that serves as a certificate for multiple properties will not increase sharply with the number of properties.

ACC pays a price, most notably in the reverification (model checking) time and an enlarged trust base on a client's side. A model checker is a non-trivial program. In ACC it is a part of the trust base. In contrast, PCC's trust base involves only a VCGen and a proof-checker.

Chapter 5

An Implementation of ACC: ACCEPT/While

The ACCEPT/While system is a complete implementation of an ACC system in a restricted context. This implementation builds on a simple procedural language with concurrent primitives that are similar to those of Java. The experience with ACCEPT/While facilitates the investigation of the following sets of questions:

- A general question: Is the basic ACC framework feasible?
- Implementation questions: How are certificates constructed for a compiled program? How may a certificate be represented and validated? Is the validation sound? Is the whole ACCEPT/While system sound? Is it complete? Is certificate construction and validation tractable?
- Performance questions: For concurrent programs written in the While-language, are certificates compact enough? Is certificate validation time reasonable in practice? Can routine optimizations be performed in an ACC-based certifying compiler?

Among these questions, ACCEPT/While focuses on the implementation issues. Theoretical development combined with system construction, including the adoption of the existing type system from SDTAL, leads to answers to the implementation questions listed above. Although a small number of While-programs were written in an attempt to answer the performance questions, the experiments reported in this chapter are performed at a smaller scale and primarily serve as a proof-of-concept. The investigation of performance is addressed more seriously in the next chapter when we extend the ACC framework to operate on C programs, such as public domain programs and Linux device drivers.

In the rest of this chapter, I will introduce the While language and a set of concurrent primitives as part of its standard library (Section 5.1). Then I explain the compilation process, which maintains the validity of the abstract model on the compiled program (Section 5.2). After that, I will show how annotations are added into the compiled code and how index type checking algorithms validate the encoded Boolean program. Readers may refer back to Section 3.6 for details of the SDTAL type system. Several properties of the ACCEPT/While system are studied (Section 5.3) , including the soundness of BP validation and the soundness of the whole system. The optimization techniques are discussed in Section 5.4. After that, we estimate the complexity of some key computation in the system in Section 5.5. Finally, experience with the ACCEPT/While system, including its implementation and applications, is reported in Section 5.6.

5.1 The While Language

Figure 5.1 gives the syntax of our While language. The While language supports concurrency. There can be multiple threads. The threads run in parallel with each other. There are no assumptions about how fast a thread runs. The behavior of the system is the behavior of the asynchronous product of all the threads.

A While program contains integer variables, which are either global or local to a thread. There are common control flow structures, such as if statements and while statements. For simplicity, ACCEPT/While does not include function calls, which allows us to write concurrent programs that do not involve recursive function calls.

The concurrent primitives supported by the While language are similar to their namesakes in the Java API, which implements a monitor semantics.

5.1.1 Semantics

We present the semantics of a While program as a labeled state transition system. The labels in the transition system are single step operations. We first define what constitutes an atomic step in the While language. The operational semantics of the state transition system is in two parts. One part resembles a conventional semantics of a simple imperative language, modifying a value assignment of the program variables. The second aspect of the semantics defines how the control

```
program       ::=   vardecl* thread+
vardecl       ::=   type varname ;
type          ::=   int | bool
threads       ::=   [sync] varname (vardecl*) { vardecl* stmt* }
stmt          ::=   assignment; | primitives
                    | if (pred) stmt else stmt
                    | stmt stmt
                    | label: stmt
                    | goto label
                    | while stmt
primitives    ::=   | wait
                    | notify
assignment    ::=   var = expr
expr          ::=   var
                    | expr baop expr
                    | uaop expr
pred          ::=   expr cop expr
                    | ubop pred
                    | pred bbop pred
baop          ::=   + | − | ∗ | % | /
uaop          ::=   −
cop           ::=   <|>|≤|≥|==
ubop          ::=   !
bbop          ::=   and | or
```

Figure 5.1: Syntax of the While Language

$$\begin{aligned}
\text{eval env x} &= \text{lookup(env, x)} \\
\text{eval env } (e_1 + e_2) &= \text{eval env } e_1 + \text{eval env } e_2 \\
\text{eval env } (e_1 \leq e_2) &= \text{if (eval env } e_1) \leq \text{(eval env } e_2) \\
& \text{ then 1 else 0}
\end{aligned}$$

Figure 5.2: Selected Evaluation rules for expressions in While

of CPU is transferred among concurrent threads.

Atomic Steps

An atomic step is a sequence of operations that are performed without being interrupted. The atomic steps in the While language are closer in granularity to those of a modeling language than to those of a programming language. For a modeling language used in verification, what constitutes an atomic step is reflected by the semantics executed by the search algorithm of the model checker. Because the model checker is performed at the source level, one step performed by the model checker executes one atomic statement, such as an assignment. For a programming language, the atomic steps are determined by the hardware on which the program is complied. A typical atomic step is one assembly instruction.

Each automatic step is a function from a machine state to another. A machine state s is defined as a tuple $(V, \mathtt{pc}, \mathtt{Qs})$, where V is a value assignment to variables, `pc` is a pair that encodes the thread number and the relative program location, and `Qs` is a data structure used in the scheduling, which will be explained later in this section. We further assume that all variables are global for the convenience of presentation.

We first define the evaluation rules for expressions. A selected set of rules is given in Figure 5.2. As is common for a While language, these rules define a recursive function `eval` that evaluates an expression given an environment. Meta-function $\mathtt{lookup}(\mathtt{env}, x)$ returns the value of x in `env`.

In the While language, we define the atomic steps as:

- An assignment, which includes reading the operands, computing the right hand side expression and updating the target.

- A test of conditions and associated transition of control, as appears in an if or while statement.

- A goto statement, which includes updating the pc component.
- Concurrent primitives, the operation details of which are explained below.

Concurrency Control

At run-time, several threads may co-exist. A thread may be runnable or waiting for a condition. The Qs component in a machine state records the status of a thread. The following operations are allowed to test Qs.

- Request the set of all runnable threads. A scheduler may choose to run one of these threads.
- Put a thread into waiting mode. For simplicity, we assume there is only one queue in the system.
- Release all the threads waiting in the queue and make them runnable.

When the concurrent primitives are not used, the semantics of the composition of multiple threads is defined as an atomic step that picks one runnable thread to occupy the CPU. The value of pc is set to the current program location of that thread.

The semantics of the monitor primitives are defined as atomic steps that operate on the Qs component. The concurrency primitives implement a monitor semantics. The code inside the monitor is labeled by a keyword sync. Other primitives control the transfer of the CPU ownership. For example, wait() causes the current thread to wait until another thread invokes the notify() function. notify () wakes up all threads that are waiting on the monitor.

Labeled Transition System

Formally, a thread is interpreted as a labeled transition system $(S, \rightarrow, s_0, L)$. The set of states S contains all program locations. There is a transition between two states if it is possible for control to transfer from the program location corresponding to the first state to the one corresponding to the second state. A transition is labeled with the atomic step that enables the transition.

The semantics of the whole program is the asynchronous product of the semantics of each thread. This semantics is similar to that of Promela programs. In fact, from the model checking point of view, the While language is a simplified version of the Promela language, although the level of abstraction in terms of concurrency control is different. Also, this similarity in semantics allows us to argue about the preservation of temporal properties.

Asynchronous Product

Formally, the asynchronous product of labeled transition systems $(S_i, \rightarrow_i, s_{0i}, L_i)$, where i ranges from 1 to n, is a new labeled transition system $(S, \rightarrow, s_0, L)$ where:

- The set of states S is the Cartesian product $S_1 \times S_2 ... \times S_n$;
- The set of transition labels L is the union of L_i's: $L_1 \cup L_2 \ldots \cup L_n$;
- The labeled transition relation $\rightarrow$ is of the form $((s_1, ..., s_n), l, (s'_1, ..., s'_n))$, where there exists an i between 1 and n, such that $(s_i, l, s'_i) \in \rightarrow_i$ and for every j other than i, $s_j = s'_j$.
- The initial state s_0 is the product $(s_{01}, ..., s_{0n})$.

Path Enumerator

Given a scheduler, we can run a While program. Due to the randomness introduced by the scheduler, we can have different runs, that is, sequences of machine states. To model check the corresponding transition system, we have to execute the program symbolically or explicitly to compute all possible runs of the system. The runs generated by this evaluator are sequences of the states (of the transition system) labeled by a representation of the machine state (a triple $(V, \mathtt{pc}, \mathtt{Qs})$). As mentioned in Chapter 4, for both a concrete program and an abstract program, we can represent the machine state as a proposition. Thus the symbolic evaluator can be viewed as the semantic engine. The pseudo code for the controled execution is given in Figure 5.3.

All the threads in the system are initialized as ready. Then we keep track of which machine states are not handled in the variable `unhandled`. When we pick a thread to run from currently ready threads, we then put the rest of the ready threads into the unhandled set of machine states. When the current run reaches its end, we output the run. We use symbol . to denote the operations allowed by a class of objects, for example, scheduler.pick.

Because the labeled transition system for the whole program is a finite state system, a run will either end or get into a loop, so the search for a run will eventually stop. Then we retreat to explore

```
for all threads t
  Qs.add_to_runnables (t)
unhandled = empty set

while (true) {
  let ready_threads = Qs.get_runnables ()
  if current run ends or infinite loop identified
     { output run
       (V,pc,Qs) = pick one from unhandled
     }
  let t = scheduler.pick ready_threads
  unhandled = union(unhandled, representation(V, pc, Qs - {t}))
  update pc t.pc
  (new_V,new_pc,new_Qs) = eval_stmt(t.stmtsAt(pc), V, pc,
  Qs)
  run = run.add (new_V, new_pc, new_Qs)
}
```

Figure 5.3: Path Enumerator of a While program

other possibilities, as in a DFS or BFS search algorithm. Extending this argument, we can prove that the search algorithm finds all the runs in the system.

To model check an LTL formula ϕ, we apply the technique described in Section 3.3. Namely, an automaton is computed for $\neg\phi$ and then the synchronous product of this automaton with the automaton corresponding to the labeled transition system is computed. Model checking an LTL formula is equivalent to testing the emptiness of the language accepted by the synchronous product automaton. This is handled by the Spin model checker [36]. We need only to translate a While program into a Promela program and the specification into an LTL formula.

Next, we show an example program and a verification task that is assisted by predicate abstraction.

5.1.2 Predicate Abstraction

As part of the ACCEPT/While toolkit, we designed a predicate abstractor for the While language that accepts, as input, a While program and a set of predicates and outputs an abstract, finite state model. This design is similar to that of Bandera's[20]. Automatic discovery of predicates is not supported in ACCEPT/While.

```
int bound;
int buf1, buf2;

sync thread1 () {
while (1) {

    while (buf1 <=0) wait ();
    buf1 --;
    notify ();

    while (buf2 >= bound) wait ();
    buf2 ++;
    notify ();

}
}

// thread2 symmetric to thread1
```

Figure 5.4: A Bounded Buffer in the While Language

For example, we use this While language to write a bounded buffer program in Figure 5.4, which contains two threads operating on two buffers. There are a fixed number of items in these buffers. One thread takes items from one buffer and puts them into the other buffer. The other thread does the opposite. The two threads are synchronized: neither thread will remove an item from an empty buffer or put an item into a full buffer. Our goal is to certify a temporal property of this program. For example, assuming a fair scheduling algorithm, we may expect that an item will eventually be removed whenever a buffer is full. This property is called `FullToNFull`. Define $\texttt{Full}_i$ as $\texttt{buf}_i \geq \texttt{bound}$. We formulate this property as the LTL formula:

$$\Box(\text{Full} \rightarrow \Diamond\text{NonFull}) \qquad (\text{FullToNonFull}).$$

Similarly, we may have another property called EmptyToNonEmpty, which states that a buffer will contain at least one item once it is empty. From the properties we can find the relevant predicates that can help reduce the size of the problem.

$$\begin{array}{rll} \{\text{Empty}_i: & \texttt{buf}_i = 0, & \text{where } i = 1,2. \\ \text{Full}_i: & \texttt{buf}_i = \texttt{bound}, & \\ \text{NonFull}_i: & \texttt{buf}_i < \texttt{bound}\} & \end{array}$$

```
bool full1, empty1, full2, empty2;

sync thread1() {

   while (empty1) wait ();
   empty1, full1 = *, 0;

   notify ();

   while (full2) wait ();
   empty2, full2 = 0, *;
   notify ();
}
```

Figure 5.5: Abstracted Bounded Buffer

Using these predicates, we can compute an abstracted version of the program, which is illustrated in Figure 5.5. The syntax of such Boolean programs is similar to that of the While language. The differences are: first, all of the variables are Boolean variables; second, there are simultaneous assignments; and, finally, there is a top element from the abstract domain (*), which can be understood as a non-deterministic choice when tested.

Consider model checking the programs using an explicit state model checker. For the concrete program, the state space, containing integers, is larger, while the abstract version contains only Boolean (bit) variables.

The algorithm used by the abstractor is summarized as follows. The abstraction is done on a thread-by-thread basis. For each thread, we do not translate the concurrent primitives. The rest of the statements are handled using the algorithm outlined in Section 3.4.2. The translation function `translate`, which, when given a While statement, returns a Boolean statement, is recursively defined as follows:

- If the statement is an assignment of the form `x=e`, then the BP statement is either a `skip` statement or of the form:

  $$\ldots, b_i, \ldots = \ldots, \texttt{choose}(F(\text{WP}(\texttt{x=e}, p_i)), F(\text{WP}(\texttt{x=e}, \neg p_i))), \ldots$$

  where the weakest precondition $\text{WP}(x = e, p)$ is $p[e/x]$ and F is defined as described in Section 3.4.2.

- If the statement is an `if` statement of the form `if (e) then stmt1 else stmt2`, then the BP statement is of the form:

```
                    if  (*) then {assume(not F(not e)); st1}
                     else {assume(not F(e)); st2}
```

  Statements st1 and st2 are the translation of stmt1 and stmt2, respectively.

- If the statement is a goto statement, the BP statement is this statement itself.

- If the statement is a labeled statement `l:   stmt`, the BP statement is a labeled statement: `l:  translate(stmt).`

The path enumerator outlined in Figure 5.3 can be applied to computed runs for an abstract program. The result is a set of runs. Each state is represented by a pair $(P, \texttt{Qs})$, where P is a proposition about the machine state, including the program counter. This representation applies to both the concrete state and the abstract state. A state $(P_1, \texttt{Qs}_1)$ covers another state $(P_2, \texttt{Qs}_2)$, both of a run π, if, for every state s in π, $P_2 \Rightarrow P_1$ and $\texttt{Qs}_1 = \texttt{Qs}_2$. For two sets of labeled runs Π_1 and Π_2, Π_1 covers Π_2 if for every labeled run π in Π_2, there is a labeled run in Π_1 that is over the same sequence of states and covers π. The state generated for an abstract transition system covers that generated for a concrete transition system.

Proposition 5.1.1 *Suppose we abstract a labeled transition system using the predicate abstraction described above. Suppose we apply the search algorithm in Figure 5.3 to search the set of runs for both system. Then the set of runs generated for the abstract transition system covers the set of runs generated for the concrete transition system.*

The proof of this proposition is through a stronger lemma: we prove that any finite prefix of a run generated for the concrete system is covered by a prefix of the same length generated for the abstract system. The proof of the lemma is by an induction on the size of the prefixes. The induction step is to prove that, if every concrete prefix of length k is covered by an abstract prefix of length k, then every concrete prefix of length $k+1$ is covered by an abstract prefix of length $k+1$. For any prefix of length k, written as π_c^k, $\pi_c^k(k)$ is of form $(P_k, \texttt{Qs}_k)$. According to the induction hypothesis, π_c^k is covered by an abstract prefix π_a^k, the k-th element of which is

$(P'_k, \mathtt{Qs}'_k)$. Now from the definition of covering, $P_k \Rightarrow P'_k$, and $\mathtt{Qs}_k = \mathtt{Qs}'_k$. The next concrete state is $(P_{k+1}, \mathtt{Qs}_{k+1})$. The next abstract state is $(P'_{k+1}, \mathtt{Qs}'_{k+1})$. There are two possibilities :

- The move is through executing an ordinary statement. In this case, $P_k \rightarrow_c (P_{k+1})$. Because predicate abstraction is a sound abstract interpretation (Theorem 3.1),$\rightarrow_c (P_k) \Rightarrow \rightarrow_a (P'_k)$, which is the same as $P_{k+1} \Rightarrow P_{k+1}$. Here symbols $\rightarrow_c$ and $\rightarrow_a$ are the (prefix) next state operators for the concrete transition system and the abstract transition system, respectively. The $\mathtt{Qs}$ component is not changed when executing an ordinary statement.

- The move is through executing a concurrent primitive. In this case, only the $\mathtt{Qs}$ component may be changed. Case analysis of how this may change shows that there is at least one $\mathtt{Qs}_{k+1}$ such that $\mathtt{Qs}_{k+1} = \mathtt{Qs}'_{k+1}$. The P component is not changed.

The abstract program produced by predicate abstraction can be translated into a Promela program. In particular, a thread is translated into a proctype in Promela. We use Promela to implement the concurrent primitives. The simultaneous assignments are modeled by Promela's `atomic` structure.

The BP in Figure 5.5 is for the source program. To certify a compiled program, we must compute an abstract model for the compiled program. This is discussed in the next section.

5.2 Certifying Compilation: Abstraction and Compilation

This section describes the compilation methods that are used in ACCEPT/While. It is intended to answer the question: how do we compute an abstract model for a compiled program. The short answer is through certifying compilation, that is, we compile a verified program into the target program together with the abstract model. The result of this certifying compilation process is a compiled program together with one of its (the compiled program's) abstract models that preserves the temporal property of concern.

First, I explain in detail the reasons why the certifying compilation approach is chosen.

5.2.1 Abstracting Compiled Program

Consider the choice between certifying compilation and directly abstracting a compiled program. The latter is, in general, more involved than the former.

The chief source of complexity derives from the state space of a compiled program, which is a refinement of the state space of the source program: the state space of a compiled program involves temporary variables, registers, and stack locations. Although these components of a state space need not to be included in an abstract state to verify the property of concern, the current automatic predicate discovery techniques may not be able to find such a set of predicates for predicate abstraction. If we use counter-example driven Boolean abstraction to characterize the same effect in a Boolean program abstracting a compiled program, then more predicates are needed. As an example, for an assignment $x = 2 * x + 1$ and a predicate $\{x > 0\}$, we will have a Boolean assignment that states: if $x > 0$ beforehand, then it will be so afterwards. But a corresponding instruction sequence, {mul x 2 r_0; add r_0 1 x}, will need another predicate $r_0 > 0$ to capture the same information. Consequently, abstracting compiled code generates a larger set of predicates. Because computation of a Boolean program takes time exponential in the number of predicates, a larger predicate set means significantly longer abstraction time.

For some programs that can be properly abstracted and model checked at the source level, the effect of these newly introduced predicates will cause a corresponding compiled program not model checkable with the same amount of computational resources. And this is not easily remedied. Predicate abstraction can be performed with a scoped predicate set, that is, different sets of predicates are used in different part of the program. Although the new predicates tend to have a small scope, within its scope the total number of predicates that need to be considered is increased.

Another problem is with the complexity of the certification. For a compiled program, if we consider a finer granularity of operations, a transition system tends to be more complicated (with more states and transitions). Certifying such a system will be fundamentally involved. For example, if we take a BLAST [32]-style approach to generate proofs for a compiled program, the result certificate size will be even larger than that generated by BLAST, which performs certification for an intermediate language close in semantics for the source program.

On the other hand, the predicates discovered for model checking the source program are at a higher level of abstraction: they do not involve the temporary variables introduced during compilation. As long as the compiled program maintains a semantics that refines that of the source program, the abstraction for the source is preserved as an abstraction of the compiled program. Such a refinement can be understood as maintaining the input-output functionality of a higher abstraction level.

Formally, the concept of refinement of semantics is defined as follows. A mapping $s' : A' \rightarrow B$ refines mapping $s : A \rightarrow B$, if there is a surjection ξ from A' to A, such that for any $a \in A$ and there exists such an a' that $\xi(a') = a$, we have $s'(a') = s(a)$. Intuitively, s' can be regarded as the machine state of a compiled program, A is the set of source variables and A' is the set of memory locations. There can be several memory locations corresponding to one source variable, hence the surjection.

For a source program statement `stmt`, its semantics is defined as a function f on program states. A program state s at the source level is a mapping from program variables to their respective type-preserving domains. The semantics of an instruction sequence can be defined as a function f', where f' is a function from one machine state to another. A machine state s' is a mapping of memory locations and registers of that machine. We represent a memory location using the name of variable it is designated to. Thus, a machine state is regarded as a refinement of the program state. An instruction sequence I is a refinement of `stmt`, if its semantics is defined as a function f' such that for any s_1, and for any s'_1 that refines s_1, if $f'(s'_1)$ is defined, then $f'(s'_1)$ refines $f(s_1)$.

Such a concept of a refinement is the basis of abstraction-preserving compilation, which is introduced below.

5.2.2 Abstraction-preserving Compilation

A key innovation of ACC is the use of source code model checking to guide target code model checking. As a first step, ACCEPT/While preserves the Boolean program computed for the source program as an abstraction of the target language. Thus model checking the same BP will successfully establish the temporal properties of the target program. The primary advantage, as discussed, is that the state space of a BP for the source is significantly smaller than a BP generated by an automated approach to model check the compiled program directly. Later in Section 5.4, I will explain how we may transform the BP and the target program together to optimize the target program while still preserving the temporal property.

It is worth noting the distinction between abstraction preserving compilation and certifying compilation for temporal properties. The former is a (sometimes overly) conservative approach, which is used in this section to illustrate the concepts. Yet the latter can be implemented through abstraction-preserving compilation plus an optimization step.

In order to preserve the abstraction, a source program fragment that defines one step in the transition must correspond to a target program fragment that has the same effect as the source fragment. If we require the surjection ξ to be a bijection, then we have a sufficient condition to preserve the abstraction. This bijection requirement is for the convenience of presentation. Ultimately, to support basic techniques of compilation, such as register allocation, a surjection is required, as shown in Section 5.4.

When ξ is a bijection, it implies that there is a dedicated memory location for every program variable. When we refer to temporal properties, the meaning of a source variable v is interpreted over its inverse image $\xi^{-1}(v)$.

Observable Variables

For a machine state to refine a program state, there must be a correspondence between program variables and memory locations. However, it is not necessary to watch all the variables. It only matters that we have a refinement for the set of variables that affects the temporal property. These variables, including those appearing in the property and the predicates involved in predicate abstraction, are called observable variables.

We require that a compiled program must have designated memory locations for observable variables. That is, they are not omitted during optimization, if any is performed. These memory locations are singled out when considering a machine state, which is regarded as a refinement of the program state.

The next condition is to maintain the refinement relation between a machine state and the program state.

Refinement of State

The simplest way to ensure a bijection is to make the observable variables behave like volatile variables in C. If a variable is declared to be volatile, then the compiler always emits a store instruction for it whenever it is modified. When observable variables are treated like volatile variables, we can observe the atomic propositions always in terms of the values stored in memory, instead of their buffered copies in the registers. For example, we require that variables involved in the temporal property, such as $\texttt{buf}_i$ in the bounded buffer example, to be written through.

Next, the condition below addresses the atomic steps, a hidden condition in abstracting a program for model checking purpose.

Atomicity

A hidden assumption in the predicate abstraction-based software model checking is that the source language element to be abstracted is atomic. For example, in SLAM, a (pure) C statement that does not involve function calls, is considered atomic. An (observable) side effect could be hazardous. For example, suppose we first clear x (to 0) when compiling the assignment $x = 3$. The assembly code will be:

$\{\texttt{mov}\ r_0\ 0;\ \texttt{store}\ r_0\ x\ ;\texttt{mov}\ r_0\ 3\ ;\ \texttt{store}\ r_0\ x\}$

If we are verifying the global property $x > 2$, a correct Boolean program of the compiled sequence is to assign *true* to a Boolean variable representing predicate $x > 2$. The boolean program model checks successfully. However, the property does not hold on the concrete assembly program because we have mistakenly abstracted away some intermediate states that violate the property. For this reason, we demand the compiled assembly level code to satisfy the following requirement.

Atomicity Requirement. A piece of assembly code compiled from an atomic source program element may have at most one `store` (to observable variables) in it.

Compilation Scheme

To summarize the steps that ACCEPT/While takes to preserve the computed Boolean program, ACCEPT/While takes a conservative approach to compilation: 1) it preserves the control flow of the original program; 2) it uses a write-through strategy to maintain in memory (as opposed to registers) the value of those variables that are observable via the state predicates; and 3) the same level of atomicity as the source program abstraction is maintained. These conditions guarantees there is a one-one correspondence between assembly blocks and source atomic steps; the compilation is made local. A local compilation refers to a compilation of a source atomic step without any knowledge of the other part of the source program. If the local compilation is correct, then the second condition guarantees that the correspondence between an instruction sequence and its source counterpart is a refinement. The third condition guarantees that the observation on the execution of the compiled program will either repeat the pre-machine state, or else show a correct post-state that refines the post state in the execution of source programs. Following the definition of the refinement, the sequences of observations (made on the source and on the compiled program) are identical, modulo state repetition. An consequence of this is that an abstraction of the source is still an abstraction of the target.

To show how control flow is preserved, assume, for a concrete program $\texttt{Prog}_C$, that we have a corresponding Boolean program $\texttt{Prog}_B$. To simplify the presentation, assume that all the statements in $\texttt{Prog}_C$ are assignments, if-statements or goto statements. The goal of abstraction-preserving compilation is to generate an untyped intermediate program $\texttt{Prog}_D$, for which $\texttt{Prog}_B$ remains an LTL preserving abstraction.

First, for every observable variable, make sure there is a unique memory location designated for that variable. Call this mapping ξ. The mapping from the source variable to its memory copy is particularly important, because, when the $\texttt{Prog}_B$ is interpreted as the abstraction for the $\texttt{Prog}_D$, a Boolean variable b corresponds to the mapping of $\xi(\gamma(b))$, where γ is concretization function. For the While language, all the variables can be allocated either in the global pool or in the thread pool. In fact, this mapping is easy to identify because there are no stack allocated variables, such as the local variables in C.

Next, we organize $\texttt{Prog}_C$ and $\texttt{Prog}_D$ as transition systems. To draw this transition system as a graph, the nodes in the graph represent program locations and edges represent assignment statements or predicates that enables the transition.

Then, we directly refine the transition system into $\texttt{Prog}_D$. An assignment in C is compiled to a block (called an assignment block) in $\texttt{Prog}_D$. An if-statement's conditional test is compiled to a block in $\texttt{Prog}_D$ (called a test block), which must have a conditional jump. A conditional jump in the intermediate language takes two labels. The translation abides by the atomicity and write-through requirements. Following the discussion at the beginning of this section, the abstraction represented by $\texttt{Prog}_B$ is preserved as long as the compilation is correct.

Thus, a $\texttt{Prog}_D$ assignment corresponds to a simultaneous assignment group in $\texttt{Prog}_B$, and a test block in $\texttt{Prog}_D$ corresponds to non-deterministic choice in $\texttt{Prog}_B$. The targets of the transition instructions are associated with the `assume` statements in $\texttt{Prog}_B$. Informally, this correspondence defines the abstraction relationship between a Boolean program and an assembly program.

The result of the abstraction-preserving compilation is an un-optimized program. Optimization can be performed on the server side during compilation or on the client side, after re-verification. Server side optimizations have to transform the code and the annotations simultaneously, while post- reverification optimizations, similar to the concept of a just-in-time (JIT) compiler, have to preserve the temporal properties. Client side optimizations are sometimes necessary because the server can perform only some of the common optimization operations, and the runtime performance of the module may be of concern. Later, I show that some of the commonly used optimization techniques are still applicable in a certifying compiler. The development derives from Necula's research on certifying compilation in TouchStone.

Completeness

The ACCEPT/While system is not complete; a module may hold a temporal property of concern, but cannot be certified. However, it is relatively complete, in the sense that if we have found the right set of predicates for the source program, then we are able to generate a BP for the compiled program. This is guaranteed by the abstraction-preserving compilation.

Validation of $\texttt{Prog}_B$ on $\texttt{Prog}_D$ is thus to verify: 1) that an assignment block in $\texttt{Prog}_D$ is abstracted into its corresponding assignments in $\texttt{Prog}_B$, and 2) that the condition assumed by an assume statement in $\texttt{Prog}_B$ always holds at the corresponding label in $\texttt{Prog}_D$. Such validation can be viewed as the validation of Hoare triples. Knowing the semantics of the assembly language, we can easily design an algorithm to validate the Boolean program and to add this algorithm to the client's trust base. In Section 3.5, I will argue for an alternative approach, used by ACCEPT/While:

the use of index types. We encode a Boolean program as type annotations in a typed assembly language [46, 45, 72] and validate the Boolean program through type checking.

Next, I introduce the encoding of Boolean programs in SDTAL, which answers the question how the Boolean programs are represented in the compiled program and sets the scene for the revalidation of the BP by a client.

5.2.3 Encoding of Boolean Programs

Type annotations are generated for each basic block based on the corresponding fragment of the Boolean program. In the translation to assembly code the `if` and `goto` statements are directly reflected in control. The `assume` statement is translated into a type assertion (a type state) in the target program. The most interesting case is the simultaneous assignment statement. All assignment statements can be stated in the form:

$$(b_1, \ldots, b_n) = (\mathtt{choose}(e_{1,1}, e_{1,2}), \ldots, \mathtt{choose}(e_{n,1}, e_{n,2}))$$

From this, the following logical constraints on the type of each block are calculated. Note how we write the components of M and M'. The meta-function γ' is the composition of γ and θ.

$$\begin{gathered}(\mathtt{int}(x_0), \ldots, \mathtt{int}(v_0), \ldots) \rightarrow (\mathtt{int}(x_0'), \ldots, \mathtt{int}(v_0'), \ldots) \\ \mid \bigwedge_{i \in n}((\gamma'(e_{i,1}) \Rightarrow \gamma'(b_i)') \wedge (\gamma'(e_{i,2}) \Rightarrow \gamma'(\neg b_i)'))\end{gathered}$$

In the example, the BP fragment: $b_3 = \mathtt{choose}(\mathit{false}, b_3)$ is translated into: $(\mathit{false} \Rightarrow \gamma'(b_2)') \wedge (\gamma'(b_0) \Rightarrow \gamma'(\neg b_0)')$ which simplifies to: $\mathtt{nPacket} = \mathtt{nPacketOld} \Rightarrow \mathtt{nPacket}' \neq \mathtt{nPacketOld}'$.

If we restrict the form of the index types that encode the BPs, then we can write a function `decode` that, when given an SDTAL program the index types in which encode a BP, will recover the BP from the index type annotations.

5.3 Soundness

This section answers the implementation questions whether the type checking of SDTAL is sound and whether the ACCEPT/While is sound by stating and proving soundness theorem.

The theorem implies that the Boolean program encoded by the type annotations is valid if the code is type correct. In other words, it establishes the correspondence between an intermediate program and a Boolean program. This result, combined with the preservation of LTL formulas discussed in Section 3.4.2, determines the general soundness of ACCEPT/While.

Proposition 5.3.1 *Let* $\texttt{Prog}_D = (l_1\ (I_1 : \tau_1), l_2\ (I_2 : \tau_2), ..., l_n\ (I_n : \tau_n))$ *be a program. Assume* $\forall i < n, \vdash I_i : \tau_i$. *If* $\texttt{BP} = \texttt{decode}(\tau_1, ..., \tau_n)$, *and* $\texttt{BP} \models \phi$, *then* $D \models \phi$, *for any next operator free LTL formula* ϕ.

Proof: Because $\forall i < n, \vdash I_i : \tau_i$, from the soundness theorem of SDTAL, for every τ_i of the form $M|P_i \rightarrow M'|Q_i$, P_i must be true when control reaches l_i and every pair of states satisfy $Q_i[\theta^{-1}]$, where θ is the function mapping variables to index variables. If Q_i is a well-formed BP encoding, then BP is a valid abstraction of D.

According to Theorem 5.1.1, the set of runs Π_B for BP covers that of D's: Π_D. From the definition of covering, for every $\pi \in \Pi_B$, there is a $\pi_1 \in \Pi_D$, such that for every j, we have $\pi_1^j \Rightarrow \pi^j$. Therefore, if $\Pi_B \models \phi$, then $\Pi_D \models \phi$ too.

5.4 Optimization

The result of abstraction-preserving compilation is an annotated compiled program. The program itself maintains the control structure of the source, with instruction sequences implementing the atomic steps. Because of the bijection requirement on ξ, there are a lot of write-throughs, which are not efficient. But as we mentioned, the bijection requirement is not necessary.

If ξ is only surjective, then the refinement condition requires that, for any observable variable, one of its reverse images under ξ must hold the current value of this observable variable. Thus, we need to keep track of which memory location or register holds the current value of an observable variable. This information, however, must be independently verified. In particular, the annotations need to be modified. The modification is similar in nature to the operations on annotations performed by optimization techniques such as common sub-expression elimination.

I use a common sub-expression elimination example to demonstrate how to optimize an SDTAL program without invalidating the associated boolean program. Suppose we have two blocks.

The first block computes an expression e and assigns it to a register r_1; the second block computes the same expression again and stores it in another register r_2.

Common sub-expression elimination will get rid of the second computation of e. Instead, we will move the content of r_1 to r_2. The challenge is: the instructions of the second block may be associated with function types, and in typing the instruction sequence, we need the fact that e is in r_2 to prove the postconditions of those function types. For example, e is $2 * y + 3$, and a post condition Q contains an implication $y > 0 \Rightarrow r_2 > 0$, which can be proved with the original sequence but not with the new sequence, because we know nothing about r_1 yet.

The trick is to add a precondition to types of the form $(M|P \rightarrow M'|Q)$. In this example, we add $(x_1 = e[\theta])$ to the precondition P, resulting in the type $(M|(P \wedge (x_1 = e[\theta])) \rightarrow M')|Q)$. Note that x_1 is the index variable that refines the type of register r_1 and $e[\theta]$ is the lifting of the expression e to the index domain. In the verification process, the precondition is verified each time the control may reach this program location. Also, the precondition is taken as the context to verify the post condition. Thus, we will know enough to prove Q.

However, a sequence such as $v := e$ may affect the evaluation of the desired properties. As a result, we need to check if the optimization affects the desired properties. If it does, the optimization will not be performed. What is interesting is that, even if the optimization is not correct, either the error is caught by the type checking on the client side, or the temporal properties will not model check, or the temporal properties still hold, but the program may not function as desired. Nonetheless, in each case we can be sure that the security policy is respected.

5.5 Complexity Analysis

This section analyzes the complexity of the certificate size generated by ACCEPT/While, the model validation algorithm, and the model checking time. It answers the implementation questions about the size of the certificate and the tractability issue of model validation and property reverification.

5.5.1 Certificate Size

The certificate size is measured relative to three factors, the number of nodes and edges in a control graph, as well as the number of predicates. We use numbers $|V|$, $|E|$ and $|\phi|$ for these factors.

The certificate size can be viewed as the size of type state assertions plus the size of state transformer types. The number of type state annotations, in the worst case, is $|V|$. The number of the state transformers, in the worst case, is $O(|E|)$.

Each of the type state annotations, in the worst case, can have $O(3^{|\phi|})$ terms, while in practice, this number is seldom reached. An optimized (and less precise) predicate abstraction algorithm may limit the number of the terms to a small number. The experience reported by several systems [24, 17] shows that this number can be less than ten.

In the worst case, for each of the state transformers, we have $O(|\phi| * 3^{|\phi|})$ number of terms, which again, is usually not reached in practice.

5.5.2 Model Revalidation Time

Model validation in ACCEPT/While is through the index type checking of SDTAL. The model revalidation time is primarily the index type checking time.

Index type checking time is dominated by the VC discharging time. As discussed, index type checking generates VCs for a decision procedure to solve. The problems tackled by such a decision procedure include SAT or integer programming; both are hard problems. Furthermore, the communication between the type checker and the decision procedure introduces overhead. In comparison, the local computation, such as type inference, handles relatively small data structures and does not involve iteration.

Consequently, in evaluating the index type checking time, it is practical to evaluate the numbers of decision procedure calls. To count this number, it is also required that each decision procedure call is of the form (P_i's are atomic predicates):

$$P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow P_{n+1}$$

We call a verification condition of this form a reduced VC.

We decompose the index type checking into parts and count the decision procedure calls involved. As described earlier in Section 3.5, type checking in SDTAL can viewed as two parts. The first part is the checking of local state transformer types; the second part is the validation of type states. We analyze the worst case performance of the two computations below.

- For checking an instruction sequence, we generate one big VC, the antecedent of which is a conjunct of the index propositions, while the conclusion is still a conjunction of implications. The number of these implication is the number of predicates involved. In the worst case, the number of reduced VC is $O(|\phi|)$. For a whole program, the number of decision procedure calls is $O(|V| \times |\phi|)$.

- For checking a type state, we need to check every transition to where this type state is specified. Altogether in a program, this number is $O(|E|)$. Checking a Boolean assume statement will generate one VC. Overall, this part of computation will generate $O(|E|)$ reduced VCs.

As a conclusion, in the worst case, type checking a SDTAL programs that encode BP takes $O(|V| * |\phi| + |E|)$ theorem proving calls.

5.6 Experience

The ACCEPT/While was constructed primarily for testing the concepts. It was implemented in OCaml, with about 4,000 lines of code. ACCEPT/While contains a front end of the while language, an abstractor, an interface to SVC (Stanford Validity Checker), a compiler to SDTAL and a translator to Promela program. We also wrote the SDTAL type checker, although it is not considered as part of the ACCEPT/While.

After parsing, the abstractor will generate an intermediate representation of a BP, given a set of predicates as input. The computation of the BP needs to call SVC in the approximation of the weakest precondition. Once the BP is computed, we can translate the BP into a Promela program to be model checked by Spin, which is explained in Appendix A.

Our experience shows quite some work can be reused. We were able to use existing code[58] in the parsing and BP generation, which saved our some time in development. A similar algorithm was later reused in ACCEPT/C. Because we had built a similar index type checker (in Haskell) in the past, the amount of work in writing the index type checker is not very much. The core part of

	Program	Certificate	Checking Source	Checking Abs.
Bounded Buffer	$702B$	$232B$	$30s$	$0.4s$
Vector Addition	$808B$	$308B$	$16s$	$0.5s$
Reader/Writer	$1570B$	$282B$	$5s$	$1s$
Sleeping Barber	$809B$	$200B$	$4s$	$1.2s$

Table 5.1: ACCEPT/While Experiments

the type checker is only a few hundred lines of code. The interface to SVC is made easy by the OCaml compiler.

The implementation was evaluated by certifying temporal properties of well studied examples from concurrency theory, such as showing deadlock freedom of a system of readers and writers. Because the language was not an existing language, it was difficult to evaluate the system on large scale examples. The set of program/properties includes:

- A bounded buffer example: some liveness properties such as FullToNonFull are certified.
- Synchronized vector addition: a safety property, that the vector index is within bound is verified.
- A reader and writer example: that parallel reading and writing operations are safe. For example, at most one writer is accessing the critical section at any given time.
- The Sleeping Barber problem: Properties such as a customer will be served once she enters the barber's shop.

The liveness properties are certified with an assumption of a fair scheduling algorithm. The compiled result of the bounded buffer example and the translation into Promela is in the Appendix.

For each of the programs, the number of theorem proving calls generated during index type checking is small (less than 20). Each proposition sent to the theorem prover is an implication with a conjunction of predicates as antecedents and a single predicate as the conclusion.

In Table 5.1 we list the size of the index-type annotations (certificate), relative to the size of the SDTAL program. The numbers are based on the sizes of the assembly programs generated by a prototype compiler for the While language.

The implementation established the end-to-end feasibility of the ACC architecture. Even for the toy examples studied, in which the model was often comparable in complexity to the source program, certificates are significantly smaller than proof-based approaches. Model validation time is quick and the model checking time is significantly smaller than model checking the concrete program. Whether or not these figures scales to real world problems is a question that we address in the next chapter.

Chapter 6

ACCEPT/C

This chapter studies the application of the ACC framework to more realistic certification tasks. I choose the reachability properties for programs compiled from C. Representative applications include the certification of API correctness for operating system device drivers. We construct a tool, ACCEPT/C, to investigate the scalability of the ACC framework by measuring the sizes of the certificates, the model revalidation time and model checking time. The experience of constructing ACCEPT/C and applying it to various certification tasks is reported.

Although the focus of the investigation is the performance data that illustrates the feasibility of the ACC framework, there are other theoretical or implementation problems involved in the construction of the ACCEPT/C toolkit. Altogether, the following questions are to be answered in this chapter.

- Can we extend the ACC framework to build a certifier/reverifier for reachability properties of programs compiled from sequential C program? For such a tool to be practical, the server side activities ought to be automatic. Can the automation be achieved?

- Is there a modular way to integrate existing systems to construct the certifier/reverifier? The application domain is complicated. For example, the input programs are syntactically unrestricted C programs instead of the toy While language, which implies that we have to deal with function calls, pointers, structures, and many other C features. Property specification, effective abstraction, as well as efficient model checking are all potential problems, not to mention certifying compilation and model revalidation. Many of the subtasks have been addressed by existing tools. Whether we are able to glue these tools together with relatively small efforts is one important problem.

- Are the sizes of the certificates small enough? Is model revalidation time short enough? Is the model checking time acceptable? More importantly, when the size of the problem increases, do these figures scale well? The client side reverification of temporal properties needs to be reasonably fast to facilitate these applications. The criteria for how fast this process should be are application-specific. For example, for a user invoked dynamic kernel module loader, such as Linux's`insmod` program, the response time should be in seconds, while re-verification of an SSH client could be longer. How we measure the performance of ACCEPT/C is another question to be answered.

These questions are addressed through a combination of software implementation, simulation, algorithm analysis and measurement. Ultimately, our experience supports the following statements:

- The ACC framework can be used to create a fully automated reachability certifier/reverifier for an intermediate language (BAL) program that is compiled from a C program. A BAL program is close to a bytecode program, more realistic than one written in SDTAL.

- The construction of this certifier/reverifier is modular. In particular, the indexed intermediate language and the BP model checker (Moped[63]) forms a core part; an ACC system can be constructed by plugging in a source level abstraction based model checker (BLAST [34] in ACCEPT/C) and a certifying compiler. This construction is more general and much easier than other state-of-the-art approaches.

- The resulting certifier computes compact certificates that can be reverified in a relatively easy manner: the size of the certificate is comparable to a state-of-art certifier (a feature in BLAST) that also certifies reachability properties for simple properties and scales better when the properties/programs become complicated.

- Both model reverification time and model checking time are acceptable in practice for the applications we studied.

The rest of the chapter is organized as follows. Section 6.1 describes the problem domain and the concrete technical problems involved in constructing an experimental platform. It provides an overview of ACCEPT/C. The ACCEPT/C system integrates several components, some of which are existing tools. Later sections elaborate the mechanisms of the different components of ACCEPT/C. In particular, I introduce how predicates are automatically discovered in BLAST [34] (Section 6.2) and how Boolean programs are model checked using Moped [63] (Section 6.5). Also, I discuss how we integrate those components for certification (Section 6.3) and the certifying compilation. The soundness of ACCEPT/C is discussed in Section 6.6. To set the goals our measurement, we then analyze the complexity of the revalidation process as well as the structure of the certificate. An estimation of the size of certificates is compared to the certificate size of BLAST's. The experimental results are presented in Section 6.8.

6.1 Overview of ACCEPT/C

6.1.1 Application Domain

The application domain chosen for ACCEPT/C is reachability properties of programs compiled from C source. As mentioned above, the goal of constructing ACCEPT/C is to investigate the extensibility of the ACC framework to real world problems and to facilitate the experiments to measure the performance of the resultant certifier. For this purpose, I focus on the certification of API safety properties for programs compiled from C source code. For these applications, several automated software model checking tools have already been built; both test cases and tools abound. API safety properties (and safety properties in general) can be converted into a finite state machine. Using the techniques outlined in Section 3.3.2, the satisfaction of the safety properties can be treated as a reachability problem.

Certification of reachability properties enables a variety of applications. In these applications, the programs are sequential components of a larger system, where there is typically a clearly defined interface with the rest of the system. Examples are:

- OS device drivers: Properties include safe use of API. For example, the drivers should use locks correctly.
- Open source programs: Properties include safety properties. For example, a program should not call `system` () after calling `setuid` (), or should not be allowed to make an Internet connection after opening a sensitive file.

Different applications have different criteria for acceptable reverification time. For the above set of applications, normally we expect the reverification time to be within a second. For the size of the certificates, we expect them to be a small fraction of the size of the program and to scale when the complexity of the problem increases.

To understand the complexity of the problem, we focus on API compliance. We study reachability properties. It is good to be aware of the API security problem from which the reachability problem is obtained because in the API security problem we have a separate representation of the program and the property, while in the reachability problem we have only one instrumented program. The complexity of the problem is measured by the number of states and transitions in the specification automaton and by the size of the relevant part of the program. The latter is often hard to measure. Some properties are simpler than others, partly because the specification automaton (translated from an LTL specification, for example) contains fewer states and transitions, partly because the program logic involved in verifying the properties is simple. Use of locks is considered a simple property. The specification of the absence of erroneous sequence of events in an SSH server, however, often contains more than 10 states, and the relevant program logic may contain multiple `case` statements.

6.1.2 System Architecture

To build a certifier/reverifier for these problems, an ACC system needs a source-level model checker and a certifying compiler. The source-level model checker should generate an abstract model as part of the abstraction-model checking paradigm. The certifying compilation is guided by abstraction. The encoding of an abstract model as index types is covered in theory in ACCEPT/While. An index typed intermediate language can be developed once and reused in different ACC systems.

On the client side, again, an index type checker is needed in addition to a model checker.

Modular Development of ACCEPT/C

One of the contributions of the ACC framework is that it provides a modular approach to the construction of a certifier/reverifier for different application domains. The interface between different components in an ACC system is relatively simple and is clearly defined. In ACCEPT/C, this

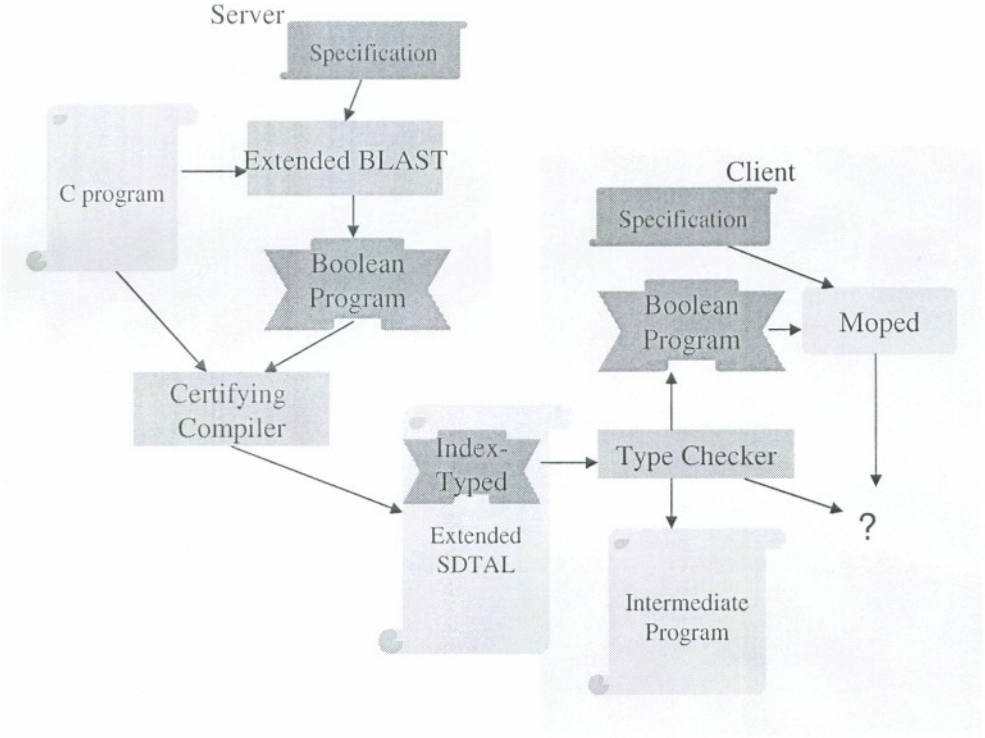


Figure 6.1: Overview of ACCEPT/C

modularity simplifies the implementation; existing systems can be introduced in a relatively easy manner.

The system architecture of the ACCEPT/C system is illustrated in Figure 6.1. ACCEPT/C accepts as input only the program and the safety specification. This is different than in ACCEPT/While, where predicates need to be provided by the programmer.

To enable automated predicate discovery, the ACCEPT/C adopts the Berkeley Lazy Abstraction Software Verification Tool (BLAST). BLAST [32] in turn uses the C Intermediate Language (CIL) to deal with the full syntax of C [43]. BLAST is able to specify API usage in terms of a monitoring automaton. The monitor automaton is synthesized with the original C code; the instrumented program is then model checked to see whether an ERROR label (inserted by the instrumentor) that represents the wrong use of the API is reachable. BLAST further adopts so-called lazy abstraction to simplify the model checking process and provides a new technique to speed up the predicate discovery process. A further introduction to BLAST is in Section 6.2.

BLAST does not output a Boolean program; instead it generates a set of predicates to abstract a C program. We modify BLAST to include a BP generator. Then a certifying compiler will compile the C program and the BP into an indexed typed BAL program. We extend BAL with an index type system that resembles SDTAL so that the BP can be encoded and validated.

On the client side, the index type system we designed for BAL is deployed to validate the BP. After that, we use Moped to model check the BP. Moped is a fast, state-of-the-art model checker for push-down system. Its model checking capacity is more than what is needed by the ACCEPT/C system: Moped checks not only reachability properties but also general LTL formulas. That is, if the certifier certifies liveness properties, Moped on the client side can still verify the liveness properties. Therefore, the Moped/BAL can be reused in many other ACC systems.

From the account above, we may conclude that an ACC system can be acquired by plugging in a source level, predicate abstraction based model checker. The client side tools will remain stable between different ACC systems. This modularity creates flexibility for both the client and the server side and allows the ACC framework to scale to different application domains.

Implementation Note

ACCEPT/C is constructed to facilitate the measurements of key performance of an ACC system. During implementation, some parts that were not relevant to these measurements were omitted. Most notably, I did not implement a full certifying compiler. The current compiler only generates intermediate code when an index type annotation is needed; the intermediate code fragment is then type checked to measure the expense of model revalidation. With this implementation, the size of the certificate, the time spent on model validation and property reverification can be measured.

Below, we introduce the components and techniques used by ACCEPT/C. First, I present the counter-example driven predicate abstraction technique and the BLAST toolkit.

6.2 Counter-example Driven Predicate Abstraction

One of the differences between ACCEPT/C and ACCEPT/While is that, in ACCEPT/C, the certifying compilation is fully automatic. This difference is the result of the different types of programs and properties considered by the two systems. For reachability properties on sequential programs, automatic tools based on so-called counter-example driven predicate abstraction work well. However, this technique has yet to be applied successfully in practice to general temporal properties on concurrent programs.

This section describes counter-example driven predicate abstraction, a technique used to automatically compute the set of predicates that, when used to predicate-abstract a program, enables successful model checking of a temporal property. Currently, it is adopted by many automatic software model checking tools [34, 31, 4].

A tool of this sort takes a temporal property and a program as input. It iteratively performs an abstraction-model checking-refinement process, which is illustrated by Figure 6.2.

- *Phase 1: Abstraction*

  From the temporal property, and other heuristics, an initial set of predicates is computed. The system uses this set of predicates to predicate abstract the program. This set may be recomputed later in the refinement phase to include more predicates. In each iteration, a new abstraction is computed in Phase 1 for a new set of predicates.

- *Phase 2: Model Checking*

  A model checker attempts to verify the temporal property on the abstract model. If the model checking is successful, the iteration stops. If not, the model checker returns an error trace for the predicate refinement phase to compute new predicates.

- *Phase 3:Predicate Refinement*

  When the model checking fails, there are two possibilities. One is that the program contains a bug; the other is that the abstraction is not accurate enough. The error trace from the model checker is analyzed to decide whether this error trace represents a real error. If an error trace reported by the model checker represents a real error, the trace is called *feasible*. If a trace is not feasible, the system will attempt to find a new set of predicates. A new iteration is started with the new set of predicates.

For example, Figure 6.3 (a) shows a C program. Figure 6.3 (b) shows the model checking tree.

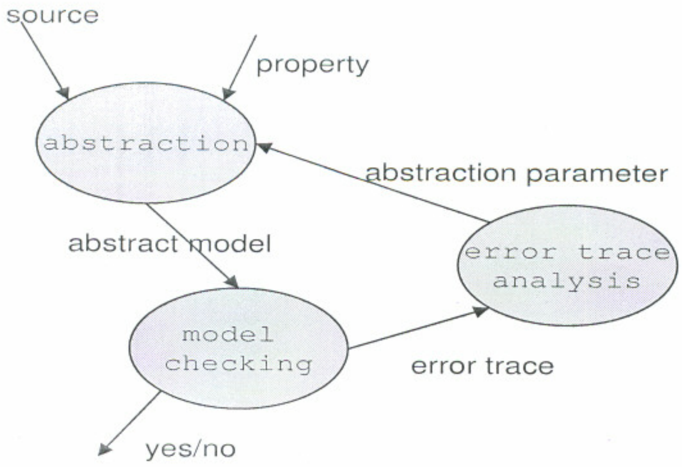


Figure 6.2: Data Flow in Refinement Based Software Model Checking

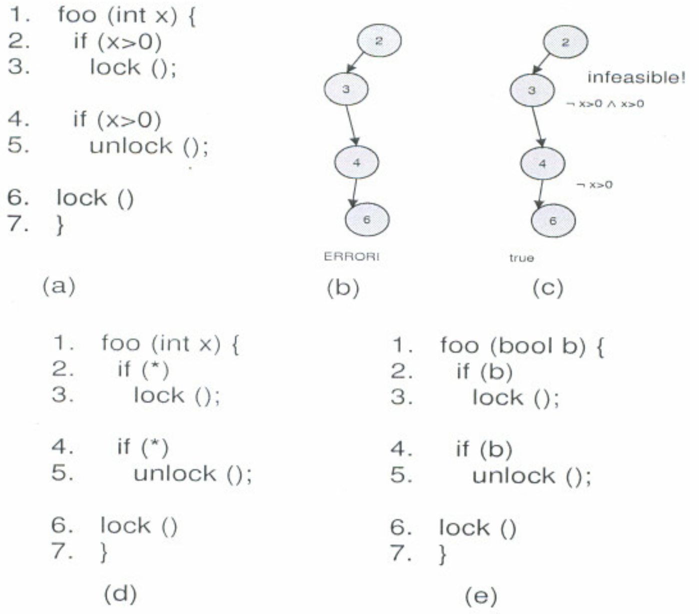
Figure 6.3 shows an error trace. And Figure 6.3 (d) shows the model checking with a new set of predicates, which induces a successful model checking tree in Figure 6.3 (e).

The rest of this section explains the predicate refinement phase. I will explain how to decide whether an error trace is (in)feasible, and how new predicates are discovered. The account is based on a general method and does not cover new approaches such as those based on Craig Interpolation [33, 16].

6.2.1 Predicate Refinement

The predicate refinement phase contains two sub-phases: first, the error trace is concretized to see if the trace is feasible. Feasibility is determined by the discovery of the real cause of an error. When an error trace is found infeasible, a second phase attempts to find a new predicate with the help of the results from the feasibility test. The second phase may or may not find a new predicate. In this sense, this approach is not complete.

An error trace is represented as a sequence of transitions in the transition system. An example is shown in Figure 6.3, where, if we do not track the condition $x > 0$ in the abstract model, then the resultant model cannot be successfully model checked. An error trace is returned, which says the control enters the then clause of the first if statement but the else branch of the second if statement. This error trace is not feasible because the condition $x > 0$ will have the same truth



```
1.  foo (int x) {
2.    if (x>0)
3.      lock ();

4.    if (x>0)
5.      unlock ();

6.  lock ()
7.  }
```

(a)

```
1.  foo (int x) {
2.    if (*)
3.      lock ();

4.    if (*)
5.      unlock ();

6.  lock ()
7.  }
```

(d)

```
1.  foo (bool b) {
2.    if (b)
3.      lock ();

4.    if (b)
5.      unlock ();

6.  lock ()
7.  }
```

(e)

Figure 6.3: Counter-example Driven Predicate Abstraction: Example

value at both places.

To detect the infeasibility of an error trace, the concrete program will be studied. The idea is that if the error trace is feasible, then the abstract state computed at a program location should be at least consistent with the weakest precondition of `true`, with respect to the sequence of statements involved in the error trace. For example, in Figure6.3, the abstract state after the first `if` indicates $x > 0$ is true, while the weakest precondition of true with respect to the `else` branch of the second `if` indicates $x > 0$ is false. Because of this conflict, we say the error trace is not feasible.

The counter example driven predicate abstraction is based on the discovery of infeasible paths. In fact, when we find a path infeasible, we often have enough information to deduce a predicate that will eliminate the imprecision and can add that predicate into the predicate set. Iteration of this process may either lead to a real error trace, or a situation when we can no longer discover any useful predicates.

6.2.2 BLAST

As a software model checking tool for C programs, BLAST extends the counter-example driven predicate abstraction-based software model checking process in two major ways. First, BLAST short-cuts the three-phase iteration by refining the abstraction on the fly. That is, the abstraction is refined only when the part of the model checking tree demands a refinement; and the refinement is effective only in the part that is necessary. Second, BLAST adopts Craig interpolation, which is an approach to finding the most relevant predicate. Lazy abstraction shortens the amount of time of model checking, while the Craig interpolation approach often results in a smaller set of predicates.

Traditional predicate abstraction builds an abstract model of a program based on a fixed set of predicates. We will call such systems *uniform* because they use the same abstraction basis for every statement in the source program. Traditional approaches are also *static* because they construct the model completely prior to model checking.

Lazy abstraction is a *dynamic, non-uniform* abstraction technique. It builds a model on-the-fly during model checking. The abstraction function is different in different parts of the model checking search tree. Predicates are added only in those portions of the search tree where they are needed.

BLAST does not compute a Boolean program. Although a BP is not hard to compute, we need to gather the predicates scattered around different portions of the model checking tree in BLAST.

6.3 Generation of Boolean Programs

6.3.1 Collecting Predicates

The model checking tree of BLAST is of the same structure as any other model checking tree. The root node represents the initial states. A successor state of a state is represented a child node of the node representing that state. A node can be viewed as a tuple that contains the previous abstract state, the operation (represented by the CIL object), the post abstract state, and other information such as the set of predicates being used (an edge seem to be logically sound to store this information). The CIL representation of the operation encodes the C statement and metadata such as source locations.

To collect the set of predicates for computing the BP, we traverse the model checking tree. We construct a table for each source location. When a node is visited, the set of predicates is added to the entry in the table representing associated with a source location. The addition is primarily set union, with a test to remove logically redundant predicates.

6.3.2 Relative Completeness

With BLAST, the abstract model generated in lazy abstraction is dynamic and non-uniform. A Boolean program, on the other hand, is static. We will use a static, non-uniform Boolean program to approximate the dynamic, non-uniform model generated by BLAST to benefit from the strength of both approaches. The BP computation is sound by construction; we need to make sure the BP computed is as effective as the dynamic model used by BLAST: if a program is successfully model checked by BLAST, the same must be true of the BP we computed.

First, we describe the collection of predicates. In the model checking tree, BLAST associates a predicate set with each tree edge. To build the Boolean program, predicate sets on all the edges corresponding to the same transition must be collected. The predicates used to construct a Boolean program must be logically equivalent to the union of the predicate sets of the tree edges representing the corresponding transition. We then apply the method described in Section 3.4.2. to compute the Boolean program. For example, for an assignment statement in C, we may compute an approximation of the weakest precondition of a predicate as the condition for the corresponding Boolean variable to be true.

As to the preservation of effectiveness, that the new BP still model checks the same property, the proposition below characterizes that the static Boolean program generated is still as appropriate a model of the original program as the dynamic model obtained by lazy abstraction, provided the lazy abstraction used Boolean abstraction locally. That is, we show that if the lazy abstraction-based model checking does not reach an error state, neither will the model checking of the Boolean program constructed above.

A state in lazy abstraction or in a Boolean program can be viewed as a bit vector. Assume the order of the bits are fixed, that is, the same bit in different states corresponds to the same predicate. We say state s_1 refines state s_2 if s_1 agrees with s_2 on every bit of s_2.

Proposition 1 *If state s_1 refines state s_2, $s_1 \hookrightarrow_1 s_1'$, and $s_2 \hookrightarrow_2 s_2'$, where $\hookrightarrow_1$ is an abstract*

$$\begin{array}{llcl}
\texttt{expressions:} & E & ::= & c \mid v \mid f(E) \mid E_1 + E_2 \mid \&E \mid \cdots \\
\texttt{predicate:} & P & ::= & \texttt{true} \mid \neg P \mid P_1 \vee P_2 \mid \cdots \\
 & & & \mid E_1 = E_2 \\
\texttt{Boolean assignment} & A & ::= & b = \texttt{schoose}(E_1, E_2) \\
\texttt{Boolean assumption} & & ::= & \texttt{assume}(P)
\end{array}$$

Figure 6.4: Syntax of the BP

transition relation induced by a subset of the predicates that induces abstract transition relation $\hookrightarrow_2$*, then* s'_1 *refines to* s'_2*.*

Therefore, if none of the leaves on the search tree contain an error state, neither will any reachable state generated by model checking the Boolean program.

These results show that ACCEPT/C does not weaken any significant properties of the models we construct.

The compilation of our example to an intermediate language is illustrated in Figure 1.4 in Chapter 1. The tests of conditions ((1) and (2)) in the intermediate program are not associated with an edge. Instead, they are associated with a node that has multiple out-edges. The direction of edges in the figure is from top to bottom unless explicitly indicated.

6.4 Certifying Compilation

The compiler in ACCEPT/C is similar in function to that in ACCEPT/While: it takes a C program and a BP as input and generates an annotated BAL program as output. This section reviews the compilation process.

6.4.1 Syntax of BP

There are two forms of annotations. A set of Boolean assignments annotates an intermediate program block compiled from a source statement. The second form of annotation associates a program location with a predicate, representing a program assertion. For example, at the beginning of a block compiled from a C `then` clause of an `if` statement we may specify such an assertion. This assertion corresponds to an `assume` statement in a Boolean program. As we will see later, this form of annotation can also be used to specify other forms of assertions.

$$\begin{array}{llll}
\text{expressions} & e & ::= & x \mid c \mid e_1 + e_2 \mid e_1 - e_2 \ldots \\
\text{indices} & iv & ::= & \text{(index variables)}x \\
\text{index propositions} & P & ::= & e_1 = e_2 \mid e_1 \leq e_2 \ldots \\
 & & & \neg P \mid P_1 \vee P_2 \mid \ldots \\
\text{type constants} & \texttt{tc} & ::= & \texttt{int} \mid \ldots \\
\text{types} & \tau & ::= & \texttt{tc}(ix) \mid (\tau_1, \tau_2) \\
 & & & \mid \tau_1 \rightarrow \tau_2 \mid (\tau | P) \ldots \\
 & & & \mid \texttt{st} \\
\text{stack type} & \texttt{st} & ::= & \cdot \mid S \mid \tau :: \texttt{st} \\
\text{ins sequence} & I & ::= & \texttt{ins}^* \\
\text{blocks} & K & ::= & l : \tau_K, I_1 : \tau; I_2
\end{array}$$

Figure 6.5: Syntax of index types in IBAL

6.4.2 Extending BAL with Index Types

Following the steps outlined in Section 3.6.2, we lift LAP into the index domain to build an index type for BAL. The syntax of the type system is similar to that of SDTAL, except that a type state now includes a component representing a stack. For details of BAL languages, refer to Appendix C.

Type checking in this system is similar to that of SDTAL. The checking of type state assertions is the same. The subtyping rules are extended with a subtyping rule for stacks. One stack type is a subtype of another if they are of the same shape and the corresponding items satisfy the subtyping relation.

$$\frac{}{\texttt{st} \sqsubseteq S} \qquad \frac{\tau_1 \subseteq \tau_2 \quad \texttt{st}_1 \subseteq \texttt{st}_2}{\tau_1 :: \texttt{st}_1 \sqsubseteq \tau_2 :: \texttt{st}_2}$$

The symbolic evaluator is a lifting of the symbolic evaluator in Section C.6.

6.4.3 Compilation

Compilation follows the scheme described in Section 5.2. Because C programs are first translated into CIL representation, the local compilation is to translate the CIL representation into IBAL blocks.

CIL

CIL (C Intermediate Language) is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs [43]. CIL's syntax can be viewed as a more structured and cleaner subset of C. A reduced number of syntactic and conceptual forms are used. For example, loops are limited to certain forms and the use of pointers are reduced to the dot (.) form. CIL removes ambiguity and redundancy in the C syntax and structures.

CIL is also higher-level than a typical intermediate language. Most importantly, the types from the source program are retained. With CIL, a programmer can manipulate the structures of a program in a convenient way.

Translating CIL into BAL

In ACCEPT/C, we only compile the part of the program that is relevant to verification. Because the predicate abstraction that we use concerns primarily simple integer arithmetic, if a statement modifies an observable variable, then it must perform integer operations. Conservative approaches are adopted to handle the casting of other types of expressions. The abstraction engine, as well as the theorem prover, does not support floating point theories. So in compilation, we simply generate fake instructions, ones that are not accepted by BAL, to estimate the size of the compiled program. The semantics of BAL will give an un-determined value to the result of such operation.

Aliases

With the presence of pointers, the computation of a (sound) Boolean program from a C program is expensive because every possible alias condition needs to be addressed; the resulting Boolean program tends to be unnecessarily large. Boolean abstraction on C often adopts pointer analysis to remove impossible cases. For example, if we have an assignment $x := 0$ and a predicate $*p = 0$, then the assignment may make the predicate true. If alias analysis can guarantee that p and x are not related, then in computing of the Boolean program, we do not have to handle the predicate $*p = 0$. For alias information to be trusted by a client, we need to put assertions at the beginning of the block, such as $\neg(\&x = p)$, indicating that p is not pointing to x. We will use the second form of the annotations (type state assertions). Verification of such assertions is the same as verifying an assertion introduced by an `assume` statement.

6.5 Model Checking Boolean Programs

The BPs generated for C programs involve function calls, the model checking of which cannot be solved using the algorithm listed in Section 5.1; that algorithm applies to finite state transition systems. The state of a BP can be viewed as a pair $(\texttt{gl}, \texttt{stk})$, where `gl` denotes the global variables and `stk` denotes a stack. An element of this stack can be interpreted as an active record, including the local variables and the current program counter. Model checking such a system is different than model checking a finite state system, because a stack can be infinitely deep.

The correct model of a BP is a pushdown system, defined as a tuple (P, Γ, Δ, s_0), where P contains the control locations, Γ is the stack alphabet, Δ is a finite subset of $(P \times \Gamma) \times (P \times \Gamma*)$. A normalized set of rules of the form $(p, w) \hookrightarrow (p, w')$, where $p \in P$ and $w, w' \in \Gamma*$, requires that w' is only one symbol longer than w. Intuitively, a stack symbol represents the active record of a function call.

We explain how an imperative program with function calls can be translated into a pushdown system [63]. The translation involves the following steps.

- First, we translate the control structure into a pushdown system. This is done in a function-by-function manner. Each function is translated to a flow graph, the nodes of which correspond to the control points in a program, and the edges of which are annotated with statements. Then a pushdown system $(\{\cdot\}, N, \delta, (\cdot, \texttt{main}_0))$ is constructed, where symbol $\cdot$ denotes the only control location, and set N is the union of the nodes in the flow graphs. The starting control location is the first node in function `main`, called $\texttt{main}_0$. The transition rules are:

  1. $(\cdot, n) \hookrightarrow (\cdot, n')$ if n and n' are connected (not by an edge annotated by a function call) nodes in a flow graph.
  2. $(\cdot, n) \hookrightarrow (\cdot, f_0 n')$ if n and n' is connected by a function call edge (calling function f), and f_0 is the entry point of the function f.
  3. $(\cdot, n) \hookrightarrow (\cdot, \epsilon)$ if the edge leaving n is annotated with a return statement.

- Second, we take global and local variables into consideration. Global variables are modeled through the control locations, while the local variables are encoded in the stack symbols. For example, if the program contains two global boolean variables, then the control locations may encode all the value combinations of the two variables. If an assignment changes the value of one of these variables, the corresponding transition rule is thus $(g_1, n) \hookrightarrow (g_2, n')$, where n and n' are the program points before and after the assignment. Control locations g_1 and g_2 are different in the bit representing the variable being modified. Similarly, local variables are encoded in the stack symbols. For example, if the function f has one local variable, the stack symbol will be set $\{0, 1\} \times N_f$, where N_f is the program points in function f and the first component is the value of the variable.

Moped [63, 23, 11] is able to solve model checking problems for Push-down systems. In ACCEPT/C, Moped is used to solve reachability problems while in general Moped can be used to evaluate LTL formulas on a pushdown system.

6.6 Soundness Issue

The difference between C and the While language is that C has aliases through the use of pointers. When aliases are present, the computation of a Boolean program is complicated. For example, consider a C statement:

```
(*p) = 0;
```

Unless we know for sure that the pointer p does not point to a variable x, we cannot decide that this statement does not affect predicate $x > 0$. A faithful translation of this statement will be Boolean statements corresponding to the C statement:

```
if (p== &x) x=0 ;
```

For bug-hunting purposes, to avoid complexity in computing Boolean programs, a tool may simply assume that aliases do not exist, or an alias analysis can be performed to acquire accurate information. These approaches are unsound and incomplete, but useful in practice, as shown by the efforts of several research projects [4, 22].

For sound certification, there are several possibilities:

First, we can use a restricted intermediate language, where aliases are not possible. This means the programs we can compile are limited. This is the approach we used in Chapter 5.

Second, we can try to encode alias information as annotations in the assembly programs. One straightforward approach records the does-not-point-to relation between pointers and values. Potentially, for every pointer's definition, we have to provide information for every live observable variable. This assumes the full knowledge of aliases.

Third, we may utilize domain knowledge to ease the task of point-to analysis. This way we may decrease the number of programs we can compile. Such domain knowledge provides hints on aliasing information. The accuracy of such hints is still subject to the verification of a client. For example, in Linux device drivers, some pointers acquired from the kernel cannot possibly point to a structure on the local stack of the driver. If the addressing offsets based on these pointers do not exceed the boundary of the structure, then we know that such a dereference cannot modify an observable variable on the stack. That the addressing offsets are within the bound of the structure can be formalized as a memory safety problem, the solutions to which include PCC and index typed assembly languages.

6.7 Discussion

This section estimates the certificate size, model validation time and model checking time. The results are similar to that of ACCEPT/While's. The estimates are compared with those of BLAST's certification method. Such comparison forms the basis for the hypotheses to be validated in the experimental section (Section 6.8).

In the estimation below, we assume that the size of the program (in term of number of non-composite statements) is n and the number of predicates involved in predicate abstraction is n_c.

6.7.1 Certificate Size

The BP in ACCEPT/C is slightly different than that in ACCEPT/While, where there are no function calls. In particular, ACCEPT/C translates function call/returns into BAL instructions. The size of the BP is relevant to the number of predicates involved in parameters.

To estimate the certificate size, assume that the number of function calls is n_f and that each function call may have up to n_p parameters. In the worst case, the portion of the BP relevant to function calls is of size $O(n_p{*}n_f{*}2^{n_c})$. Recall that in the worst case, the BP size can be $O(n{*}2^{n_c})$. In ACCEPT/C, the worst case estimate of the BP size is larger than that in ACCEPT/While by a factor of n_f. For most programs in practice, this number n_f is limited.

6.7.2 Reverification

Model revalidation is again measured by the number of basic theorem prover calls. A basic theorem prover call queries the truth value of an implication that has as antecedent a conjunction of atomic predicates and as conclusion an atomic predicate. Similar to that in ACCEPT/While, this number is proportional to the size of the BP and may be exponential in the number of predicates. In the worst case, the number is $O(2^{n_c} * n)$.

ACCEPT/C deals with sequential programs. Therefore, the model checking time is primarily spent on model checking a push-down system. In the worst case, the model checking will not terminate.

6.7.3 Certification in BLAST

BLAST certifies reachability properties at the CIL level. The certification is based on the concept of *invariant sets* [32]. For a transition system, an invariant set is a set I of states that : (1) $s_0 \in I$; (2) I does not include an error state; and (3) I is closed under transitions. An invariant set includes all the states reachable from the initial state.

Certification is thus to provide an invariant set together with the proofs for these conditions. The invariant set can be represented as a mapping from states to propositions: the proposition must hold whenever model checking reaches the corresponding state. Therefore, for a state s, the corresponding proposition $I(s)$ is the disjunction of all the labeling propositions generated during

```
for the current node
  begin
   find the abstract state p1 associated with the node;
   for every outgoing edges e
     begin
      find the operations op
      compute p2=post(op, p1)
      find the abstract state p3 associated target of e
      generate proof for that p2 implies p3
     end
  repeat this process for all the children
  end
```

Figure 6.6: Pseudo Code of BLAST's Proof Generation Strategy

the model checking process. We call such propositions collective labeling propositions.

Generating the proofs for the three conditions is through calling a proof-generating theorem prover for a set of verification conditions. Among the three conditions, VCs for (1) and (2) are not complicated. If the system has precondition, VCs for (1) imply that the precondition is satisfied by $I(s_0)$. VCs for (2) imply that $\texttt{pc} = \texttt{ERROR}$ never holds for a state s given $I(s)$. VCs for condition (3) require that for every transition made during model checking, the labeling propositions associated with the starting and ending states must be consistent with the semantics of the CIL language. Namely, $\texttt{sp}(I(s_1), \texttt{op}) \Rightarrow I(s_2)$, where $\texttt{sp}$ is the strongest postcondition of transition label $\texttt{op}$.

The pseudo code for proof generation is listed in Figure 6.6. The algorithm takes as input a model checking tree. BLAST traverses this tree. For each node, BLAST will find out the operations (post operator) and associated abstract state, the latter represented as a proposition. The most important proof is part of the proof for the condition (3) above: the proof that from an abstract state P_1, through a (concrete operation), we can reach another state P_2. A proof for this can be is one for $\texttt{post}(P_1) \Rightarrow P_2$. In the current BLAST implementation, other kinds of proofs are omitted.

For the SLAM example in Figure 1.2, one example of such one-step proof is constructed for the following VC. This VC represents one step of transition in the model checking tree corresponding to the if statement in the loop. This VC is trivially true.

$$\begin{array}{ll} & (nPackets = nPacketsOld) \\ \wedge & (lockStatus = 1) \\ \wedge & \neg(lockStatus = 0) \\ \wedge & \neg(request = 0) \\ \Rightarrow & (nPackets = nPacketOld) \end{array}$$

With such certificates, property reverification on a client side in BLAST system is a simplified model checking process. Every time the next state is needed, the model checker looks up in the certificate the two associated states and verifies that one state (its implicit representation) is a precondition of the other by checking the proof.

In the worst case, the size of the certificate is proportional to the size of the model checking tree, which can be exponential to the size of the program. However, when the logic in the program concerning the desired property is relatively simple, the shape of the search tree is narrow, and the size of the certificate will not be large. With BLAST, because the certificate utilizes the knowledge discovered during model checking, the size of the certificate could be smaller than a BP statement. For example, if a BP statement implies the precondition of a proposition is A or B, while B will never occur during execution. Without global knowledge, a BP generator will keep the part of the BP that deals with B, which is not considered by the BLAST certifier because the condition B never appears in the model checking tree.

However, when the problem becomes more complicated, the certificate size will increase drastically due to the state explosion phenominon. Thus BLAST certificates do not scale well to large program or complicated properties.

6.8 Experiments

ACCEPT/C provides an adequate platform to test ACC's practicality. We applied ACCEPT/C to NT and Linux device drivers to certify the correct usage of locks; we also specified and certified other API usage properties for some public domain programs. The results of the measurements are encouraging, confirming our intended claim that the ACCEPT/C toolkit is a practical certification and verification tool for important properties.

Two sets of experiments are reported. One set is to certify simple properties of Linux device drivers. The other set is for more complicated properties of larger programs. While the first set of experiments shows the capability of ACCEPT/C, the second set of experiments allows us to investigate the scalability of the toolkit.

6.8.1 Case Studies 1: Simple Properties

The first case study is on device drivers; simple properties such as the use of locks are studied. The primary goal of this set of experiments is to demonstrate that: 1) ACCEPT/C can perform reasonably well in an application domain where software model checking has been successful. 2) The size of the certificate is comparable to those generated by BLAST. Referring back to the discussion in Section 6.7, for simple properties, the approach adopted by BLAST may generate a smaller certificate.

Validation of the BP and model checking the temporal properties will not be a primary concern for simple properties in this application domain. The verification of certified device drivers may happen when a user loads the driver into the system, or during the integration of the driver by the kernel team into a distribution. A reasonable expectation of the verification time is probably a few seconds, which is easily satisfied by the actual model validation and property verification time.

Expectation

We expect the size of the certificates to be comparable to those generated by tools of similar expressive power. TPCC is not optimized for the certificate size; our primary goal is to compare our results with BLAST's [35]. As discussed in the earlier chapters, for the simple properties and almost straight-line programs, BLAST's results tend to be slightly better than ACC's. As reported by the above paper, the size of the certificates generated for this set of programs ranges from a few hundred bytes to a few thousand bytes.

Experiments

Spinlocks are used extensively in the Linux kernel; they offer short term protection of critical operations. To avoid the runtime overhead introduced by function calls, spinlocks are usually implemented through C macros. At the C level, the use of spinlocks can be identified by tracking the use of the macros `spinlock` and `spin_unlock`. For the intermediate language, additional annotations are needed to mark the beginning and end of such macros. These annotations will be striped off after verification. So there will be no runtime overhead. However, the client must make sure that each marked sequence matches the implementation of these macros on the client machine.

Device drivers contain functions that are called by the system kernel. For example, the functions may be called by an event handler. In our experiment, we simulate these function calls by invoking them from a pseudo `main` function. This new program is then instrumented with lock specifications and run through ACCEPT/C, which measures the size of the certificate, the model validation time and model reverification time.

Measurements and Interpretations

In Figure 6.7, we show the certificate size/instrumented program size ratio relative to the size of the instrumented program for 18 Linux network device drivers. The certificate size/source size ratio ranges from 80% to 10%; it tends to be smaller when the size of the program increases. Two facts can be observed: first, the ratio increases as the size of the program does. Second, the range of the ratio is one or two levels of significance larger than that of BLAST's.

The cause of the second observation is that the BP (and henceforth its encoding) includes function calls, which are not included in BLAST's certificates. To make a better comparison, we remove the function call related part from the BP; the result is shown in Figure 6.8. The range of the ratio is now comparable to that of BLAST's.

After adjusting for function calls, the ratios for several bigger programs are larger. This is because those programs contain more lock related operations.

6.8.2 Case Study 2: Scalability of ACCEPT/C

The second sets of programs studied includes larger programs and more complicated properties. We applied ACCEPT/C to the following programs/properties.

- NT drivers, for example cdaudio.c. Those are from BLAST test cases. The properties are the correct use of an API.

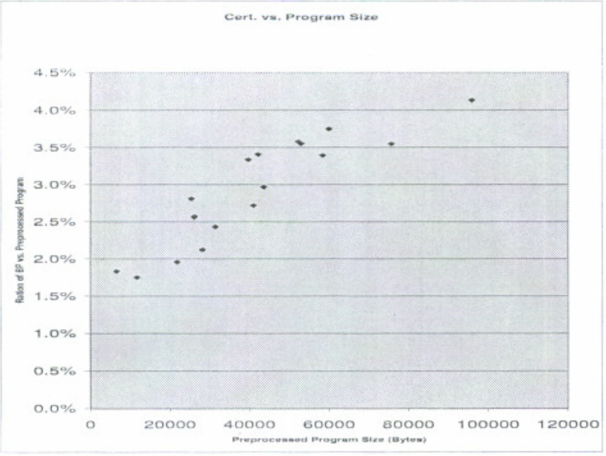


Figure 6.7: Certificate Size vs. Program Size

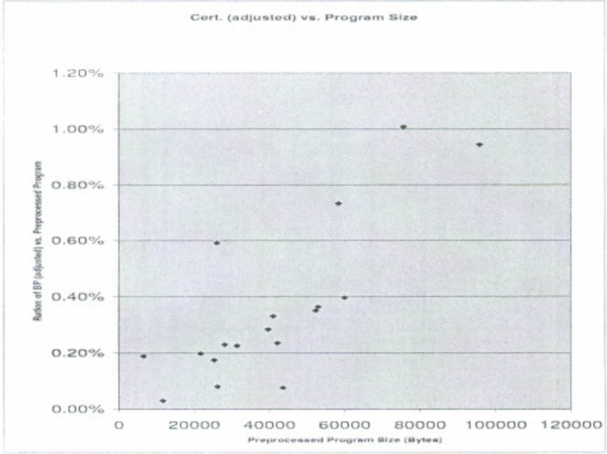


Figure 6.8: Certificate Size vs. Program Size (Adjusted)

	Program Size	Cert. Size	BLAST Size	Model Valid.	MC Abs.	MC Source
cdaudio.c	456K	55K	23M	0.5s	0.15s	600 s
qpmouse.c	263K	51K	3M	0.1s	0.05s	13.4s
ssh client	76K	13K	0.7M	0.1s	0.48s	800 s
ide.c	410K	21K	18M	0.1s	0.23s	13.2s

Table 6.1: Representative Results

- SSH implementations, including a server and a client. The properties are the absence of certain (20) undesired sequences of states. This is from Magic test cases [12].
- Public domain programs, for example, GAIM, an instant messaging program. The properties are security concerns. These are programs studied by MCC [65]

The size of the program ranges from a few hundreds lines to a few thousand of lines. Even this number is not significantly larger than that of the Linux device drivers, the part of the programs relevant to the properties certainly is. The property specifications typically contain 6-20 states. For these applications, we expect the model validation time, model checking time and the size of a certificate are acceptable. This time, two criteria are used: the (absolute) criterion for being acceptable is that the size should be a fraction of the size of the program and the verification time is within seconds. The (relative) criterion is that the certificate size should be a lot smaller than those generated by BLAST. The measurements given in Table 6.1 confirm our hypothesis.

6.8.3 Comparison with BLAST

The sizes of the certificates generated by BLAST are significantly larger especially when the properties are complicated. For reachability properties, this means that there are many branches in the instrumented program. For example, the SSH client contains many case statements. These condition tests expand the size of the search tree. For a general LTL property, if the translation of this property contains many transitions, then the out-degree of nodes is relatively large, and the instrumented program tends to contain many `case` statements pertaining to the transitions in the specification automaton. These case statements will be retained in any abstraction of the program and affect the complexity of the model checking search trees. As discussed earlier, when the shape of the search tree is no longer linear, the size of the proof tends to increase rapidly. For example, with atp.c in Table 6.1, the part of the program and the specification is rather simple, the certificate size generated by BLAST is almost the same as that generated by ACCEPT/C.

For the use of spinlocks in Linux network drivers, we have reported the certificate sizes of the two tools are similar. When we study more complicated properties of larger programs, for example, SSH client and server, the ratio grows drastically, as shown on Figure 6.9.

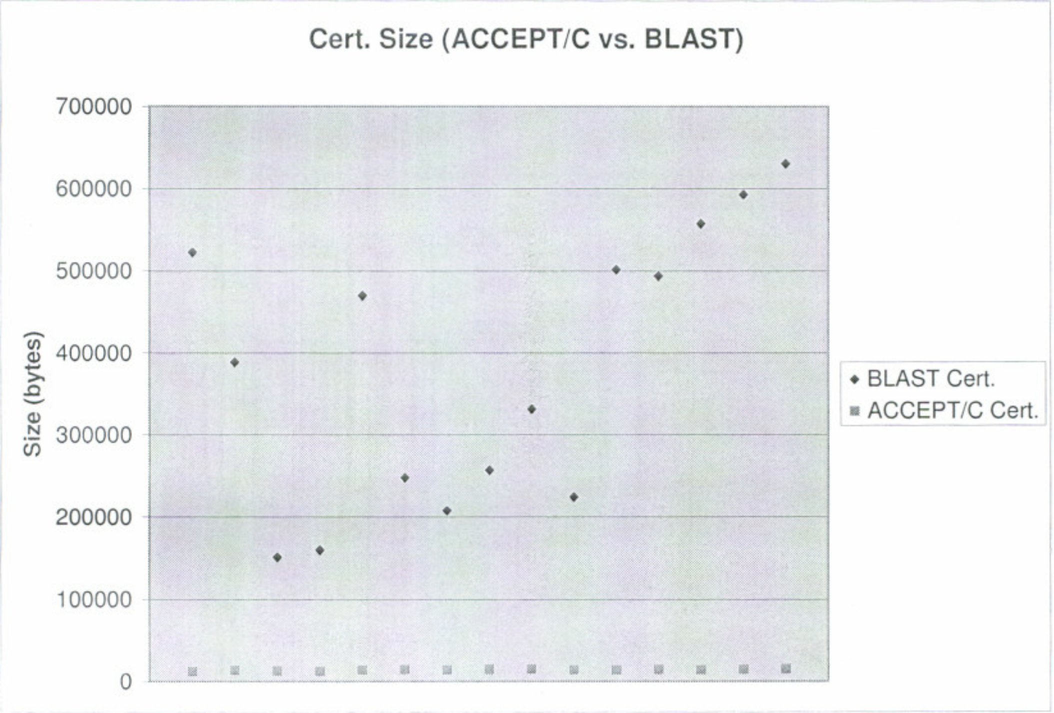


Figure 6.9: Comparison between ACCEPT/C and BLAST: Complicated Problems

Chapter 7

Related Work, Future Research and Conclusion

We have presented the framework of ACC, two versions of prototype systems, and the experience of applying the prototypes to a set of certification tasks. This chapter generalizes our results and provides a high-level view of the ACC approach. We discuss how we place ACC among other certification methods and how the ACC prototypes that we have built may be extended to support more realistic or more interesting applications.

7.1 Related Work

In the previous chapters, we have established that ACC provides a practically useful alternative to the PCC-style certification/re-verification of temporal properties of a program. It represents an important point in the solution space. To sketch an overview of this solution space and to place the existing or unexplored approaches onto this space, we focus on the following criteria.

- The practical expressive power of an approach. A property is practically certifiable when there is a defined algorithm or heuristics to generate the certificate. It is the properties that are certified in practice that matter; classes of properties that can be certified in theory but require excessive computational efforts will not count.

- The trust base, that is, the set of programs entrusted by the client. Traditional PCC approaches aim to reduce the size of the trust base. ACC adds to include a model checker and a decision procedure. An approach is better in this aspect if the trust base is compact, which not only increases the confidence of security but also allows certain applications where the number and size of programs running on a client is limited (for example, a PDA).

	Expressive Power			Trust Base					Scales	Server Activity		
	type safety	mem safety	temp safety	proof checker	VCGen	model checker	decision procedure	other tools		theorem proving	model checking	other analysis
PCC	√	√		√	fixed				√	√		some
Extended PCC	√	√	strong	√	flex					√	√	some
MCC		√	weak		flex			√	√			heavy
ACC	√		strong		flex	√	√		√	√	√	
ship source	√	√	strong			√	√	√	√	none		

Table 7.1: Comparison of different certification method

- The certificate size. Instead of comparing whether an approach generates a small certificate, I focus on whether an approach scales in terms of certificate size when the size of the problem grows. For most approaches, when the properties are simple and the program does not involve many alternative control paths, the certificate tends to be of reasonable size. Scalability concerns do prevent some of the approaches from being applied in practice.

- The computation performed by a server. I focus on the existence of a well-defined certification procedure. Because the activities are performed off-line, the cost of performing such activities is not our primary concern.

Client-performed activities are not listed as a criterion because this aspect is addressed by the trust base, which indicates the computation performed by a client.

As discussed earlier, the original PCC[50] and related systems[46, 45, 40] address type safety and memory safety. The trust base used in these systems tends to be small. In most cases, only a proof checker is involved. Fundamental PCC [1] attempts to further reduce the size of the trust base. A certificate in a PCC system includes data-flow annotations as well as safety proofs. PCC tends to scale well when the size of the problem increases. Server side activities in PCC involve proof-generating analysis of the program; client side activities are simply proof-checking.

PCC is extended to address temporal properties (Extended PCC in Figure 7.1). TPCC ([8]) addresses the problem of reducing the need of a VCGen. Although TPCC has the potential to certify temporal properties, the lack of a well defined process of certification does not support this in practice. There are other approaches that may generate a temporal proof. For example, Namjoshi's certified model checker [48] attempts to generate a temporal proof from the result of a model checker. This can be combined with automatic predicate abstraction. Namjoshi described how to construct a proof of temporal properties from the model checking result [48] and how to lift the proof, if the model checking is on an abstract model, to a concrete model [49]. In these approaches, the trust base is still a proof checker. As a proof-based approach, it does not scale well when the problem size increases.

One of the design goals of TPCC was to remove VCGen from the trust base of PCC: a module will carry the temporal property with it. Although the same applies to ACC, the size of the trust base is not reduced because a VCGen has to be written to validate the abstract model. Still, the VCGen in ACC, like that in TPCC, is very different than the VCGen in PCC: ACC's VCGen, implemented as an index type checker, is independent of the properties of concern.

The idea of sending a model of the program as a certificate is close to that of MCC [64, 65]. MCC achieves similar expressive power as that of ACC. One difference is that MCC generates certificates for a standalone program while ACC certifies a software module. The method of constructing a model in ACC is through abstraction, which is different than that in MCC. The reported property tends to be simple in MCC (2-6 states). Also, policy enforcement in MCC is a mixture of static checking and dynamic monitoring while in ACC the enforcement is static. The abstraction techniques used in ACC can be used as a replacement of the model construction technique in the current MCC implementation. In fact, ACC started with a more ambitious goal similar to that of MCC. The ideas for ACC originated in Spring 2001, about the same time as the MCC project started.

Along with ACC, another idea is to send the source code directly to the client side to be verified, compiled and executed. This is only theoretically interesting because it represents a point in the solution space. It carries nothing; yet the amount of work on the client side can be practically unacceptable to verify a general temporal property.

History of Index Typed Assembly Languages

Typed assembly languages [46, 45, 72] attempt to build a sound basis for an intermediate language. Morrissett's TAL and STAL are able to compile many language features including parametric polymorphism. A well-typed program in TAL or STAL can guarantee type safety of the code. Efforts are also made by TAL to handle memory safety. In particular, TALx86[44] involves two macros to access array elements, which require an explicit array size as a parameter. The expansion of the macro will check the array index to see whether it is out-of-bounds at run time. The type system of TALx86 uses singleton types to regulate the use of such an array size. The size parameter, which may be a variable, must be equal to the size of the array. To support this, equality checking must be handled. Yet TALx86 does not allow us to optimize the macros. DTAL [72] attacks this problem with an index type system. With the extra decision-making power that comes from the ability to solve linear inequalities, DTAL allows us to open up the macros and remove unnecessary run-time array bound checks. In our ACC research, we have found that the decision-making power of a DTAL-family language is sufficient to support the encoding and revalidation of a predicate abstraction, provided that the prediates used in predicate abstraction are expressible in the index language of the DTAL-family language.

7.2 Future Work

Future work includes the extension of a real intermediate/assembly language with index types. Such extension creates a typed assembly language to address both memory safety and temporal safety (by working with an ACC framework). In fact, a sound certification for a language with aliases can be reduced to a memory safety issue. Practical automated software model checkers for C all use an approximate alias analysis or an assumption that no aliasing occurs. For certain applications where the absence of aliases can be proven, we may be able to construct a memory safety property that ensures absence of aliases. In Linux device drivers, for example, if we are assured that: 1) every indirect memory access from a pointer does not exceed the size of the structure, 2) different pointers are not aliases of each other, and 3) the offset of indirect memory accesses from a pointer does not touch an observable field unless there is a BP annotation, then we can guarantee that there is no aliases. Index types can be used to express these conditions. Conditions (1) and (3) are typical conditions checked by an index type checker.

Another direction is to investigate the flexibility implied by the ACC framework. In ACC, counter-example driven predicate discovery is suggested, which means that the server must know the temporal property to compute an abstract model. Still, it is possible for the server to work on a cluster of possible security properties and to generate a model to certify all of them. The server thus does not have to know which one of the properties will be enforced by a particular client.

The certification of a family of properties is encouraged by the fact that the size of a boolean program is typically not sensitive to the property being certified, as we have observed in the SSH test bed. In fact, the SSH server problem we studied allows us to explore one of the initial motivations for considering the use of models as certificates: in principle, one model may be able to certify multiple properties. (Ultimately we would like to validate a property that was unknown at the time the model was created, perhaps due to API evolution.)

In the SSH server example, ACCEPT/C calculated a single Boolean Program that can be used to certify all 16 properties. This collective certificate of the 16 properties is only 25% larger than the largest certificate necessary to verify a single property. This suggests that ACC may be useful for building certificates that can support the validation of multiple properties.

7.3 Conclusions

Abstraction-carrying code provides a framework for the certification of properties of engineering significance about programs in an untrusted environment. Experience with the ACCEPT/C toolkit shows that ACC certificates are reasonably compact and that the generation, validation, and reverification of certificates is tractable.

When compared with Temporal PCC, ACC may require more client side computation but generates significantly more compact certificates. ACC also requires a larger trust base than most PCC variants. An ACC client must trust a model checker and an automatic theorem proving tool capable of establishing the validity of the certificate.

Future work includes the completion of the C-based ACC implementation. The most significant outstanding task is to implement client side support for the annotated intermediate language. In our preliminary investigation of ACC we have completed this task for a Java virtual machine variant. We expect the same techniques to apply.

The contributions of this dissertation are:

- We propose an alternative approach to certify temporal properties for a reusable module. The idea of evidence-based program certification is introduced by PCC. The basic idea that a proof can be recreated using stored sketches can be further traced back to the Edinburgh

LCF[29]. ACC extends the expressive power of these techniques to temporal properties in a practical way. ACC uses an abstract interpretation as a certificate.

- In ACC, certificates are generated through predicate abstraction and certifying compilation. Predicate abstraction has previously been applied successfully in hunting bugs. Predicate abstraction over-approximates a program; it can also be used in proving LTL formulas. Automatic predicate discovery techniques made it possible for a server to find the right predicates and to certify a module automatically. Through certifying compilation, we transform a certificate for the source code to one for the compiled code; this idea is inherited from PCC. Also, optimization of compiled code is through a process similar to that used in PCC: the parallel transformation of annotations as well as program representations.

- Index types are used to encode and verify a Boolean program. Index types are similar to dependent types; their application to intermediate/assembly program verification has been studied. Both ACCEPT tools adopt existing type systems from such index typed assembly languages. In applying index types to the representation of a Boolean program generated during successful software model checking process, we manage to limit the size of a model validation problem.

- Our experience with Both ACCEPT tools shows that this approach is valid and generates certificates that scale well when the size of the problem increases. We implemented ACCEPT/While and ACCEPT/C. When extended to a full system, each has its own potential applications. The experiments generate certificates the sizes of which are a small percentage of the source program sizes. The computational resources consumed by the client side, namely the time spent on model validation and property reverification are small in the application domain that we have studied, which include certification of practical useful properties for device drivers and some public domain programs.

Bibliography

[1] APPEL, A. Foundational Proof-carrying Code. In *Proceeding of 16th IEEE Symposium on Logics in Computer Science* (Boston, MA, June 2001), pp. 247–258.

[2] AUGUSTSSON, L. Cayenne - a Language with Dependent Types. In *Proceedings of the third International Conference on Functional Programming* (Baltimore, MD, 1998), pp. 239–250.

[3] B. BERARD ET AL. *Systems and Software Verification.* Springer, 2001.

[4] BALL, T., AND RAJAMANI, S. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of the eighth International SPIN Workshop on Model Checking of Software, Lecture Notes in Computer Science (LNCS) 2057* (Toronto, Canada, May 2001), Springer-Verlag, pp. 103–122.

[5] BALL, T., AND RAJAMANI, S. K. Bebop: A Symbolic Model Checker for Boolean Programs. In *Proceedings of the seventh International SPIN Workshop on Model Checking Software* (Stanford University, CA, 2000), pp. 113–130.

[6] BARRETT, C., AND BEREZIN, S. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Proceedings of the sixteenth International Conference on Computer Aided Verification* (Boston, MA, July 2004), pp. 515–518.

[7] BARRETT, C. W., DILL, D. L., AND LEVITT, J. R. Validity Checking for Combinations of Theories with Equality. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science (LNCS) 1166* (Palo Alto, CA, 1996), Springer, pp. 187–201.

[8] BERNARD, A., AND LEE, P. Temporal Logic for Proof-carrying Code. In *Proceedings of the eighteenth International Conference on Automated Deduction, Lecture Notes in Artificial Intellegence (LNAI) 2392* (Copenhagen, Denmark, July 2002), pp. 31–46.

[9] BIAGIONI, E., HARPER, R., LEE, P., AND MILNES, B. G. Signatures for a Network Stack: A Systems Application of Standard ML. In *Proceedings of the ACM Conference on LISP and Functional Programming* (Orlando, FL, June 1994), ACM Press, pp. 55–64.

[10] BOARD OF INQUIRY. Ariane 5 flight 501 failure. English version available at http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

[11] BOUAJJANI, A., ESPARZA, J., AND MALER, O. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *International Conference on Concurrency Theory* (Warsaw, Poland, 1997), pp. 135–150.

[12] CHAKI, S., CLARKE, E., GROCE, A., AND JHA, S. Modular verification of software components in c. In *Proceedings of the twenty-fifth International Conference on Software Engineering* (Portland, OR, 2003), pp. 385–395.

[13] CHEN, Z. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Addison-Wesley, 2000.

[14] CLARKE, E., GRUMBERG, O., AND PELED, D. *Model Checking.* MIT Press, 1999.

[15] COUSOT, P., AND COUSOT, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the Fourth International Symposium on Principles of Programming Languages* (Los Angeles, CA, Jan. 1977), pp. 238–252.

[16] CRAIG, W. Linear Reasoning. *Journal of Symbolic Logic 22* (1957), 250–268.

[17] DAS, S., DILL, D., AND PARK, S. J. Experience with Predicate Abstraction. In *Proceedings of the Eleventh International Conference on Computer Aided Verification, Lecture Notes in Computer Science (LNCS) 1633* (Trento, Italy, July 1999), pp. 160–171.

[18] DEAN, D., FELTEN, E., WALLACH, D., AND BALFANZ, D. Java Security: Web Browsers and Beyond. In *Internet Beseiged: Countering Cyberspace Scofflaws* (New York, 1997), D. E. Denning and P. J. Denning, Eds., ACM Press, pp. 241–269.

[19] DETLEFS, D., NELSON, G., AND SAXE, J. B. Simplify: A theorem prover for program checking. Technical Report HPL-2-3-148, HP Laboratories, Palo Alto, CA, 2003.

[20] DWYER, M., HATCLIFF, J., JOEHANES, R., LAUBACH, S., PASAREANU, C., VISSER, R., AND ZHENG, H. Tool-supported Program Abstraction for Finite-state Verification. In *Proceedings of the twenty-third International Conference on Software Engineering* (Toronto, Canada, 2001), pp. 177–187.

[21] E.CLARKE, AND E.EMERSON. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of Workshop of Logic of Programs, Lecture Notes in Computer Science (LNCS) 131* (New York, 1981), Springer.

[22] ENGLER, D. R., CHEN, D. Y., AND CHOU, A. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles* (Chateau Lake Louise, Banff, Canada, 2001), pp. 57–72.

[23] ESPARZA, J., KUČERA, A., AND SCHWOON, S. Model-Checking LTL with Regular Valuations for Pushdown Systems. *Information and Computation 186*, 2 (November 2003), 355–376.

[24] FLANAGAN, C., AND QADEER, S. Predicate Abstraction for Software Verification. In *Proceedings of International Symposium on Principle of Programming Languages 2002* (Portland, OR, 2002), pp. 191–202.

[25] FRANK YELIN. Low level security in Java. In *Proceedings of the Fourth International World Wide Web Conference* (Damstadt, Germany, 1995), O'Reilly, pp. 369–379.

[26] FREUND, AND MITCHELL. A Type System for Object Initialization in the Java Bytecode Language. *ACM Transactions on Programming Languages and Systems 21(6)* (1999), 1196–1250.

[27] GASTIN, P., AND ODDOUX, D. Fast LTL to Büchi Automata Translation. In *Proceedings of the thirteenth Conference on Computer Aided Verification* (Paris, France, 2001), G. Berry, H. Comon, and A. Finkel, Eds., no. 2102 in Lecture Notes in Computer Science, Springer Verlag, pp. 53–65.

[28] GONG, L. *Inside Java 2 Platform Security: Architechtures, API Design and Implementation.* Addison- Wesley, 1999.

[29] GORDON, M., MILNER, R., AND WADSWORTH, C. Edinburgh LCF. In *Lecture Notes in Computer Science (LNCS) 78* (1979), Springer.

[30] HARPER, R., AND POLLACK, R. Type Checking with Universes. *Theoretical Computer Science 89*, 1 (1991), 107–136.

[31] HAVELUND, K., AND PRESSBURGER, T. Model Checking JAVA Programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer 2*, 4 (2000), 366–381.

[32] HENZINGER, T., JHALA, R., MAJUMDAR, R., NECULA, G., SUTRE, G., AND WEIMER, W. Temporal-Safety Proofs for Systems Code. In *Proceedings of the Fourteenth International Conference on Computer-Aided Verification* (Copenhagen, Denmark, 2002), pp. 526–538.

[33] Henzinger, T. A., Jhala, R., Majumdar, R., and McMillan, K. Abstraction from Proofs. In *Proceedings of the Thirty-first Annual Symposium on Principles of Programming Languages (POPL)* (New Orleans, LA, 2004), ACM Press, pp. 232–244.

[34] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. Lazy Abstraction. In *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages* (Portland, OR, 2002), pp. 58–70.

[35] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. Software Verification with BLAST. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)* (Portland, OR, 2003), LNCS 2648, Springer-Verlag, pp. 235–239.

[36] Holzmann, G. J. The Model Checker SPIN. *Software Engineering 23*, 5 (1997), 279–295.

[37] Jackson, D., Shlyakhter, I., and Sridharan, M. A Micromodularity Mechanism. In *Proceedings of the ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference* (2001).

[38] J.Quielle, and J.Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the Fifth International Symposium on Programming* (1982), pp. 337–350.

[39] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[40] Kozen, D. Efficient Code Certification, 1998. Technical Report, Computer Science Department, Cornell University.

[41] Martin-Löf, P., 1984. Intuitionistic Type Theory, Bibliopolis-Napoli.

[42] McGraw, G., and Felten, E. *Securing Java.* John Wiley & Sons, 1999.

[43] McPeak, S., Necula, G. C., Rahul, S. P., and Weimer, W. CIL: Intermediate Languages and Tools for C Program Analysis and Transformation. In *Proceedings of Conference on Compiler Construction* (Grenoble, France, March 2002).

[44] Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., and Zdancewic, S. TALx86: A Realistic Typed Assembly Language. In *Proceedings of ACM SIGPLAN Workshop on Compiler Support for System Software* (1999), pp. 25–35.

[45] Morrisett, G., Crary, K., Glew, N., and Walker, D. Stacked-based Typed Assembly Language. In *Proceedings of workshop on Types in Compilation, LNCS 1473* (Kyto, Japan, 1998), Springer Verlag, pp. 28–52.

[46] MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21*, 3 (1999), 527–568.

[47] MOSKEWICZ, M., MADIGAN, C., ZHAO, Y., AND ANDS. MALIK, L. Z. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC2001)* (Las Vegas, NV, 2001), pp. 10–15.

[48] NAMJOSHI, K. S. Certifying Model Checkers. In *Proceedings of the Thirteenth Conference on Computer Aided Verification* (Paris, France, 2001), pp. 8–19.

[49] NAMJOSHI, K. S. Lifting temporal proofs through abstractions. In *Proceedings of the Fourth International Conference on Verification, Model Checking and Abstract Interpretation* (New York, 2003), pp. 20–32.

[50] NECULA, G. *Compiling with Proofs.* PhD thesis, Carnegie Mellon University, 1998.

[51] NECULA, G. A Scalable Architecture for Proof-Carrying Code, 2001. An invited paper at the Fifth International Symposium of Functional and Logic Programming.

[52] NECULA, G., AND LEE, P. Safe, Untrusted Agents using Proof-Carrying Code. Lecture Notes in Computer Sciences Special Issue on Mobile Agents.

[53] NECULA, G., AND LEE, P. Safe Kernel Extensions without Runtime Checking. In *2nd Symposium on Operating System Design and Implementation* (Seattle, WA, 1996), USENIX, pp. 229–243.

[54] NIPKOW, T. Verified bytecode verifiers. In *Foundations of Software Science and Computation Structures, LNCS 2030* (2001), Springer-Verlag, pp. 347–363.

[55] O'CALLAHAN, R. A simple, comprehensive type system for java bytecode subroutines. In *Proceedings of 26th Symposium of Principles of Programmming Languages* (1999), ACM Press, pp. 70–78.

[56] PODELSKI, A., AND RYBALCHENKO, A. Transition Invariants. In *IEEE International Symposium on Logic in Computer Science* (Turku, Finland, July 13-17 2004), pp. 32–41.

[57] QIAN, Z. A Formal Specification of Java Virtual Machine Instructions for Objects, Methods, and Subroutines. In *Formal Syntax and Semantics of Java, Lecture Notes in Computer Science (LNCS) 1523* (1998), Springer-Verlag.

[58] ROBBY. Code for a class project implementing a predicate abstractor for a simple language.

[59] ROSE, E., AND ROSE, K. Lightweight Bytecode Verification. In *Proceedings of Workshop on Formal Underpinning of Java* (1998).

[60] Published on website: www.rtca.org.

[61] RUANE, L. Process Synchronization in the UTS Kernel. *Computing Systems*, 3 (1990).

[62] SAIDI, H. Model-checking Guided Abstraction and Analysis. In *Proceedings of SAS'00, LNCS 1824* (Santa Barbara, CA, USA, July 2000), Springer-Verlag, pp. 377–389.

[63] SCHWOON, S. Moped software. available at http://wwwbrauer.informatik.tu-muenchen.de/~ schwoon/moped/.

[64] SEKAR, R., RAMAKRISHNAN, C., RAMAKRISHNAN, I., AND SMOLKA, S. Model-carrying Code(MCC): A New Paradigm for Mobile Code Security. In *New Security Paradigm Workshop* (2001).

[65] SEKAR, R., VENKATAKRISHNAN, V., BASU, S., BHATKAR, S., AND DUVARNEY, D. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proceedings of ACM Symposium on Operating System Principles* (2003), pp. 15–28.

[66] SHLYAKHTER, I., SEATER, R., JACKSON, D., SRIDHARAN, M., AND TAGHDIRI, M. Debugging Overconstrained Declarative Models Using Unsatisfiable Cores. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE 2003)* (Montreal, Quebec, Canada, October 2003), pp. 57–69.

[67] STATA, R., AND ABADI, M. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems 21(1)* (1999), 90–137.

[68] STUMP, A., BARRETT, C. W., AND DILL, D. L. CVC: A Cooperating Validity Checker. In *14th International Conference on Computer Aided Verification (CAV)* (2002), E. Brinksma and K. G. Larsen, Eds., vol. 2404 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 500–504. Copenhagen, Denmark.

[69] TAN, L., AND CLEAVELAND, R. Evidence-based Model Checking. In *Proceedings of the Thirteenth Conference on Computer Aided Verfication (CAV)* (2001), pp. 455–470.

[70] VALPANO, D., AND SMITH, G. A Type-based Approach to Program Security. In *Proceedings of TAPSOFT'97, Lecture Notes in Computer Science (LNCS) 1214* (1997), Springer-Verlag, pp. 607–621.

[71] XI, H. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1997.

[72] XI, H., AND HARPER, R. Dependently Typed Assembly Language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (September 2001), pp. 169–180.

[73] XI, H., AND PFENNING, F. Dependent Types in Practical Programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas* (New York, NY, 1999), pp. 214–227.

[74] XI, H., AND XIA, S. Towards Array Bounds Check Elimination in JVML. In *Proceedings of CASCON'99* (Toronto ON Canada, November 1999), pp. 110–124.

[75] ZENGER, C. Index Types. In *Theoretical Computer Science* (1998), vol. 187, Elsevier, pp. 147–165.

[76] ZENGER, C. *Indizierte Typen.* PhD thesis, University of Karlsruhe, 1998.

[77] ZHANG, L., AND MALIK, S. The Quest for Efficient Boolean Satisfiability Solvers. In *Proceedings of 8th International Conference on Computer Aided Deduction(CADE 2002) and also in Proceedings of 14th Conference on Computer Aided Verification (CAV2002)* (Copenhagen, Denmark, July 2002), pp. 2–7.

Appendix A

Translate While to Promela

This chapter explains how we translate the While-language into Promela programs.

Although the While language is a programming language and the Promela language is a modeling language, the While language is similar in semantics to the Promela language. The semantics defined for the While language has a coarser granularity in single steps than an ordinary programming language does. A consequence of this granularity is that The translation to Promela program is relatively easy.

The synoposys of the translation is as follows.

- A definition of a thread body is translated into a proc.
- An atomic step in the while-program is translated into an atomic block in Promela.
- A `sync` block in a While program is implemented in Promela as a crticial section protected by a lock. Because there is only one monitor in a While program, there will be one system-wide lock for this purpose.
- The monitor semantics of `wait` and `notify` is implemented using an algorithm first described by Ruane[61]. As in the monitor semantics, a non-preemptive CPU scheduling is implemented.

For example, the bounded buffer program will be translated into the Promela program below.

```
active proctype run1()
{
do
        :: 1 ->
                /* take1 */
atomic { (lk1 == 0) -> lk1 = 1 }; /* spinlock(&lk1) */

do /* while (empty) */
:: (empty1 == 1) ->        /* wait           */
atomic {first_wait = 1;
State1 = Wakeme;
lk1 = 0;} /* freelock(&lk1) */
(State1 == Running);         /* wait for wakeup */
atomic {(lk1 ==0) -> lk1 = 1};                         /* regain the lock lk1 */
:: else ->        /* empty1 == 0 */
break
od;

assert(empty1 == 0); /* should still be true */

atomic {
temp1 = almostfull1;
if
                      :: (full1) -> almostfull1 = 1;
                      :: else -> almostfull1 = 0;
                fi;

                if
                      :: almostfull2 -> empty1 =1
                      :: (!almostfull2 && (temp1 || full1)) -> empty1 =0
                      :: else -> empty1 = top;
                fi;


                full1 = 0;
                nonfull1 = 1;


                assert ( full1 || nonfull1);
                };
lk1 = 0; /* freelock(&lk1) */

                /* now we need to signal whoever is waiting on the
                   first_wait. */
/* wakeup routine */
/* waitlock(&lk1) */
if
:: first_wait ->/* someone is sleeping */
atomic { /* spinlock on sleep queue */
(sleep_q == 0) -> sleep_q = 1
};
first_wait = 0;

(lk1 == 0); /* waitlock(&lk1) */
if
:: (State2 == Wakeme) -> progress0:
State2 = Running;
:: else ->
fi;
                        sleep_q =0;
:: else ->
fi;
         progress1:
        /* put2 */
atomic { (lk2 == 0) -> lk2 = 1 }; /* spinlock(&lk2) */
                end1:
do /* while (empty) */
::  full2  ->                 /* wait          */
atomic {second_wait = 1;
State1 = Wakeme;
```

Appendix B

Frequently Asked Questions

In this part, we compile the answers to a set of questions. In general, the answers are covered by my presentation.

Q: What properties can ACC certify?

A: The shortest answer is temporal properties. But what exactly logic is supported depends on a lot of factors. If general abstraction is performed, then only ACTL is supported, because only ACTL properties are preserved through abstraction. If compilation is involved, that is, the abstraction is not performed directly on the compiled program, then we cannot certify temporal properties involving a Next operator. Furthermore, if automatic certification is required, then reachablity on sequential programs is probably where counter-example driven predicate abstraction is practial.

If the client and the server need to agree on what properties to check, then we should have a universally accepted way of specifying properties. As mentioned, definitions of atomic properties becomes a problem, because refering to the registers in the compiled code may be meaningless to a client. Ideal way is to have a piece of shared knowledge of the runtime. Thus API comformance is the most expressible property.

Q: What programs can ACC certify?

A: Again, if automatic certification is required, then concurrency presents an engineering challenge. Existing research in that direction is still underway when this dissertation is written, for example, Podelski's work [56]. On the other hand, because abstraction is ubiqitous in software model checking, ACC can be applied to potentially any programs that are model checkable.

When predicate abstraction is performed, aliases presents another challenge. Full knowledge of alias is required for the system to be sound, at the cost of increased complexity. ACC is more useful for an alias free language. This dissertation also discusses approaches to handle alias for certain applications.

Q: How is ACC different from PCC in property specification?

A: One interesting aspect of ACC is that the trusted programs used by a client is independent of the properties of concern. In orignal version of PCC, the VCGen, a program used by the client to generate the formulas to prove, hard codes the safety property of concern. Thus in ACC, a client and a server may negotiate on what properties to certify, or a server can simply allow the mobile program to carry the property. This observation is also published by the TPCC project [8].

Appendix C

BAL Language

This appendix gives a detailed account of BAL, the target language of ACCEPT/C. We describe the virtual machine that it is running on, its intruction set, semantics, and how post conditions are computed in the language. The computation of a post condition will be lifted into index type checking. We also illustrate how a PCC system might be built with BAL as the target language.

C.1 Architecture of the Virtual Machine

BAL programs run on a virtual machine called BALM, which is similar to the JVM.

A BALM contains one heap. A programmer can allocate a piece of contiguous memory from the heap by calling a specific allocation instruction. Such a piece of memory, called a region, is identified by a handle. Upon allocation, a region has a fixed size. This size cannot change during the life time of the region. Explicit deallocation of a region is not necessary because BALM supports garbage collection. To access a heap location, a handle and an offset are required; heap access out of the bounds of the allocated region designated by the handle is prohibited.

Like a JVM, a BALM supports multiple threads. Each thread has a frame stack, which contains one or more frames. Each frame is can be viewed as a form of an activation record. Formally, a frame can be viewed as a triple (`stk`, `reg`, `pc`), where `reg` is a set of registers, `stk` is an operand stack local to the current function call and `pc` points to a program location to be executed. The registers in a BLAM can be accessed by either names or numbers. For a single threaded system, a BALM can be abstracted as a pair (`fs`, `heap`), where `fs` is a frame stack, and `heap` is a heap. The `pc` of the top frame on the frame stack determines the current instruction.

When a function is called through an *invoke* instruction, a new frame is created. The registers of this frame will be initialized to hold the arguments of the function call. The right number and type of arguments must be stored on the caller's operand stack before the function call and would be popped off afterward. The new frame also contains an empty operand stack and a pc pointing to the first instruction of the callee. This new frame is pushed onto the top of the frame stack. The pc in the caller's frame is updated to the program location after the invocation, to be used when function call returns.

When a function call returns through a *return* instruction, the top element of the operand stack will be kept as the return value, the frame destroyed and the return value pushed onto the operand stack in the caller's frame. The execution resumes from the program location pointed to by the pc in the caller's frame.

C.1.1 Operands

For simplicity, BALM has only integer and handle as the types of operands. They can be stored in registers, operand stacks and the heap. There are four addressing modes in BLAM:

- Immediate: a number embedded in the instruction. For example, the instruction *push 3* pushes a constant 3 onto the operand stack.
- Register: the operand is a register; we use an index (written as a subscript) to refer to the registers. For example, r_0.
- Automatic: operands are on the stack and are not explicitly specified in the intruction. For example, instruction *add* takes two operands from the stack.
- Heap: the operand is on the heap. Any heap access takes the handle of a region and an offset, both stored in the operand stack.

C.1.2 Instruction Set

BAL instructions can be divided into the following categories.

- Arithmetic instructions: BAL provides arithmetic instructions such as `add`, `sub` and `mul`. All these instructions take (pop) operands from the operand stack and push the result back to the stack.

```
arithmetics   a-ins     ::=  add | sub | mul | ···
transitions   jmpcond   ::=  jeq | jle | jgt | ···
instructions  ins       ::=  a-ins
                             load reg | storereg
                             | hload | hstore
                             | jmpcond label
                             | invoke fname | return
                             | jumplabel
```

Figure C.1: Syntax of BAL

- Transition instructions: Conditional transition instructions `jeq`, `jne`, ..., compare the top two operands on the stack and jump to the specified label or the next instruction sequence accordingly. The operands are popped from the stack. There is also an unconditional jump.
- There are instructions `load`, `store`, `hstore` and `hload`. Load and store transfers data between registers and the operand stack. `hstore` and *hload* transfer data between the operand stack and a heap location. For example, instruction `load` r_3 pushes the value of register 3 onto the stack. Instruction `store` r_3 pops the top value of the stack and stores it in the register 3. An *hstore* instruction transfers the top element to a heap location. During the execution, the top three elements of the stack will be the data to be transfered, the offset and the handle of the heap location, in that order on the stack. Similarly, for an `hload`, we pop the offset and the handle of a heap location and push the data stored in the heap location onto the operand stack.
- Function call and return instructions, as introduced in the previous section.
- Miscellaneous instructions. Examples are `pop` and `push`.

A list of BAL instructions is in Figure C.1.

C.2 Operational Semantics of BAL

I present a set of evaluation rules for the BAL instructions. For each instruction, we list the expected form of the machine state before the instruction is executed, and the form of the state

Instruction	Expected Machine State	Resultant Machine State
add	$((x :: y :: s, -, pc) :: -, -)$	$(((x + y) :: s, -, pc + 1) :: -, -)$
sub	$((x :: y :: s, -, pc) :: -, -)$	$(((x - y) :: s, -, pc + 1) :: -, -)$
mul	$((x :: y :: s, -, pc) :: -, -)$	$(((x * y) :: s, -, pc + 1) :: -, -)$
push i	$((s, -, pc) :: -, -)$	$((i :: s, -, pc + 1) :: -, -)$
pop	$((x :: s, -, pc) :: -, -)$	$((s, -, pc + 1) :: -, -)$
load i	$((s, r[i = x], pc) :: -, -)$	$((x :: s, r, pc + 1) :: -, -)$
store i	$((x :: s, r, pc) :: -, -)$	$((s, r[i \mapsto x], pc + 1) :: -, -)$
hload	$((o :: a :: s, -, pc) :: -, h[(a, o) = x])$	$((x :: s, -, pc + 1) :: -, h)$
hstore	$((x :: o :: a :: s, -, pc) :: -, h)$	$((s, -, pc + 1) :: -, h[(a, o) \mapsto x])$
jmp l	$((-, -, pc) :: -, -)$	$((-, -, l) :: -, -)$
jeq	$((x :: s, -, pc) :: -, -)$	$((s, -, pc + 1) :: -, -) \quad x \neq 0$ $((s, -, l) :: -, -) \quad x = 0$
invoke f	$((x_1 :: x_2 :: ... x_n :: s, r, pc) :: fs, -)$	$((\cdot, r[r_1 = x_1 ...], f) :: (s, r, pc + 1) :: -, -)$
return	$((x :: s, r, pc) :: (s', r', pc') :: fs, -, l)$	$((x :: s', r', pc') :: fs, -)$

Table C.1: Evaluation Rules for BAL Instructions

after execution. If the machine state is not of the expected form when we are about to execute an instruction, then an error will occur. We assume that the machine will get stuck when such an error condition occurs.

As noted earlier, a (single-threaded) machine state is represented by a pair that contains a frame stack and a heap. The frames on the frame stack are represented as triples (`stk`,`reg`,`pc`). We represent both the registers and the heap as mappings. For a heap `heap`, the indices is a pair comprising of a handle and an offset. For a register file, the indices are integers. A stack is represented as a list: we write :: as the CONS operator in a list.

For a mapping $\theta : X \rightarrow Y$, we write $\theta[P]$ for a mapping where a proposition P holds. Expression $\theta[x \mapsto e]$ denotes a new mapping that differs from θ in only that x's value is changed to e.

A stack of the form $x :: y ::$ `stk` has at least two elements x and y. This notation is similar to the pattern matching syntax (for lists) in a functional language such as ML. A dot $(\cdot)$ represents an empty stack. A hyphen $(-)$ represents a value that we don't care about and that wouldn't change in the evaluation rules

The evaluation rules are listed in Figure C.1. Some of the representative ones are explained below:

- The rule for *add* : The top two elements of the current operand stack are two integers. We will consume the two operands, and the sum will be put back onto the stack. The `pc`is incremented. In the notation we use, an expression $(x :: y :: s, -, pc)$ represents a frame where the operand stack has at least two elements named x and y, respectively, and the register file of which we do not care.

- The rule for *hload* : Before execution, on the top of the current operand stack are the offset and the handle of a heap location. This instruction pops the offset and the handle from the stack. If the pair (handle, offset) represents a valid heap access, that is, the offset is within the memory range associated with the handle, then the value stored at that heap location is fetched and pushed onto the current operand stack.

- The rule for *jeq l* : Before execution, on the top of the current operand stack is a value. The value is consumed and tested by the execution. If the value is equal to zero, then the next instruction to be executed is at the program label l, as indicated in this instruction. Otherwise, sequential execution is assumed, that is, the instruction following this *jeq l* in the program will be executed.

- The rule for *invoke f* : To call a function f, the caller must push arguments $x_1, x_2, ..., x_n$ onto the current operand stack. The execution will consume all these arguments and use them to initialize a frame, with an empty operand stack, and with the registers $1, 2, .., n$ initialized to contain $x_1, x_2, ..., x_n$, respectively. This newly created frame is pushed onto the frame stack, on top of the current frame stack. The control is transfered to the beginning of function f, labeled by `f`.

- The rule for *return* : Before we return from a function, the callee must push the return value onto its operand stack. The current frame is popped from the frame stack and discarded. The return value is pushed onto the now-current operand stack, that of the caller's. The value stored in the `pc` component of the caller's frame is resumed as current `pc`.

C.3 Safety Policy

In this section, we describe, in English, a safety policy to be enforced by a client. This safety includes type safety and memory safety.

- *Type Safety*

  In BAL, an operand is either an integer or a handle. The policy demands:

  - For arithmetic operations, both operands are required to be integers.
  - Heap access requires an offset, which is an integer, and a handle. An integer can never be used as a handle.
  - The only way we can have a handle is through allocation, or passed as an argument.

  These requirements help prevent a module from violating the safe heap access requirement below.

- *Memory Safety*

  Memory safety is primarily interpreted as safe heap access. Namely, associated with a handle there is a size of the region. It is required that for a heap access the offset should not exceed the size. When the kernel passes a data structure to a module, the module will have access to data needed for its operation. A malicious module might make an out-of-bound region access. If granted, this could allow a malicious module to access other kernel data structures that the module should not access.

C.4 A Safety Logic

Now I introduce the syntax and evaluation rules for the safety logic. The evaluation rules under a machine state will determine what we expect to be a valid formula in this logic.

C.4.1 Syntax of the Safety Logic

The syntax of the safety logic involves expressions and formulas. Possible forms of expressions are:

- a register that represents a local variable, for example, r_0, or
- a location in the current operand stack, indexed by a constant, for example, $\texttt{Stack}[0]$ for the top element of the stack, or
- a location in the heap, represented by a handle and an offset, for example, $\texttt{heap}[a, o]$, or
- an integer constant, for example, 3, or
- a composite expression constructed by an arithmetic operator $(+, -, *)$ and an appropriate number of subexpressions

Examples of an expression are: 3, $x + 1$, $3 * (\texttt{Stack}(0) - 10)$.

Formulas include atomic formulas and composite formulas. Atomic formulas can be of one of the following forms.

- true and false;
- comparisons: the comparison between expressions, for example, $3 < r_0$;
- typing: that an expression is an integer or an address;
- safety: This includes safe memory access, safe typing, or safe transition (that the target transition are within the instruction space).

Composite formulas are formulas joined together by propositional logic connectives, including negations, conjunctions, disjunctions, and implications. The syntax also includes quantifications.

C.4.2 Valuation of the Safety Logic

In this section, I define the intended meaning of the logic formulas in Figure C.2. The *model* s for this logic is the machine state, namely a pair $(\texttt{fs}, \texttt{heap})$.

We first define a valuation function $E(s, e)$ to interpret an expression in the safety logic under a machine state s.

With this evaluation function, we are able to define the validity of a proposition, as shown in Figure C.4. Relation $s \models p$ defines the validity conditions in a meta-language. The meta

variables	x, y		
stack elements	*st*	$::=$	$\texttt{stack}[i]$
heap values	*hp*	$::=$	$\texttt{hp}(h, o)$
int constants	c		
expressions	e	$::=$	$x \mid c \mid \mathit{st} \mid \mathit{hp} \mid e_1\ \mathit{aop}\ e_2$
arithmetic ops	*aop*	$::=$	$+ \mid - \mid * \ldots$
types	t, τ	$::=$	$\texttt{int} \mid \texttt{handle}$
comparison	*cop*	$::=$	$\leq \ldots$
predicates	P	$::=$	$\texttt{safeheapaccess}() \mid \texttt{type}()$
formulas	f	$::=$	$\texttt{true} \mid \texttt{false}$
			$\texttt{cop}\ e_1 e_2 \mid$
			$\neg f \mid f_1 \vee f_2 \mid f_1 \wedge f_2 \mid$
			$f_1 \Rightarrow f_1 \mid \forall x.f \mid \exists x.f$
			$P^n(e_1, ..., e_n)$

Figure C.2: Syntax of the Safety Logic

$$E(s, c) = c$$
$$E(((\texttt{stk}, \texttt{reg}, \texttt{pc}) :: \texttt{fs}, \texttt{heap}), \mathit{stack}[i]) = \texttt{stk}[i]$$
$$E(((\texttt{stk}, \texttt{reg}, l) :: \texttt{fs}, \texttt{heap}), pc) = l$$
$$E(((\texttt{stk}, \texttt{reg}, l) :: \texttt{fs}, \texttt{heap}), r_i) = \texttt{reg}[i]$$
$$E(((\texttt{stk}, \texttt{reg}, l) :: \texttt{fs}, \texttt{heap}), \texttt{heap}(a, o)) = \texttt{heap}[(a, o)]$$
$$E(s, x + y) = E(s, x) + E(s, y)$$
$$E(s, x - y) = E(s, x) - E(s, y)$$
$$E(s, x * y) = E(s, x) * E(s, y)$$

Figure C.3: The Valuation Function of Expressions

$s \models \texttt{true}$	
$s \models x_1 \leq x_2$	iff $E(s, x_1)$ is less than or equal to $E(s, x_2)$
$s \models \neg f$	iff $s \not\models f$
$s \models f_1 \vee f_2$	iff $s \models f_1$ or $s \models f_2$
$s \models \texttt{type}(x, \tau)$	iff x is an integer and τ is type `int`, or x is a handle and τ is the type constant handle
$s \models \texttt{safeheapaccess}(a, o)$	iff $s \models \texttt{type}(a, \texttt{handle})$, $\texttt{type}(o, \texttt{int})$, and $o \leq \texttt{range}(a)$
$s \models \forall x.P(x)$	for all x in its domain, $s \models P(x)$

Figure C.4: Definition of Validity of the Safety Logic

language provides a means of interpreting the formulas. Such interpretation reflects our intention of the logic system; any formal proof system must be sound, i.e., consistent with the intended interpretation.

For example, the validity relation states that $(\texttt{type}(x, \tau))$ is true only when x is an integer and τ is the type constant `int`, or when x is a handle and τ is type constant handle, which is exactly what these types are intended to mean. Also, the $\texttt{safeheapaccess}(a, o)$ is true only if a is a handle, o is an integer and o is within the offset range of the handle (as returned by the meta function `range`), which simply paraphrases the requirement for a heap access.

Now that we have defined the syntax and semantics of this safety logic, we sketch a theorem prover that can actually prove a formula. In particular, we expect that such a theorem prover proves a formula that is true under every possible model when a program location is reached. As mentioned earlier, the computational resources to run a theorem prover on the server side are more than those on the client side.

Referring back to Figure 2.3, this theorem prover is the certifying compiler. It produces dataflow annotations and proofs that are carried as the certificate. In the next section, we introduce the certifying compiler.

C.5 Certifying Compiler

The ultimate goal for a server is to prove the safety of the compiled module and attach the proof as a certificate. Traditionally in PCC, this task is decomposed into several subtasks: The safety is proved first on the source program and then through a safety-preserving compilation process, retained on the target program. Such a decomposition is based on the fact that safety properties are easier to prove on the source program. For example, type safety can be proved through the type checker of most high-level programming languages. The negative side is that the compilation may not optimize the target code without justification that the optimization will not hinder the compilation from generating safety proofs. However, Necula's research demonstrates a large number of optimizations are still possible, although typically the scope of the optimization is limited.

The activities of a certifying compiler include:

- Program analysis that establishes the safety of the program. For example, it may type-check the source programs to make sure type safety.

- Compilation of the source program into intermediate programs while preserving the safety property.

- Annotation of the intermediate program to assist the VCGen to be able to generate VCs without repeating the program analysis task. For example, program analysis may find out some of the array accesses are safe, which depends on a fact found by the analyzer that in a loop, a variable is bounded.

- Invocation of a VCGen to produce verification conditions that are essential to the safety properties.

- Generation of checkable proofs for the VCs.

Below we show how to establish the type and memory safety for the C-like program in Figure C.5. This program is influenced by the example given on page 16 of Necula's dissertation [50]. The function `foo` takes two arguments, A is an array and B is an integer. The type safety is maintained by the high level language where there is no type coercion. The program is memory

```
int foo(int A[], int B)
{
      int i;
      int j;

      j=B;
      for (i=0; i < length(A); i++)
        j = j+A[i];
}
```

Figure C.5: An Example C Program

safe because the variable i, used as the index to the array A, never exceeds the length of the array. Function `length` returns the upper bound of the array.

We compile the source code into the BAL code in Figure C.6. We maintain the type safety and the memory safety. First, a register is annotated with its type. For example, at the beginning of function `foo`, we add a precondition indicating that A is a handle. Thus the type of r_1 is handle. Note that these annotations are written in the safety logic presented earlier, where the symbol ":" is used as an infix predicate for "type()". The simplest way to achieve type safety in BAL is to ask that r_1 always stores a handle. Similarly, registers r_2, r_3 and r_4, allocated for B, i and j, respectively, are integers.

It is necessary to annotate the loop with a loop invariant indicating the types for these registers. Doing so, we only need to check that these types are maintained after every loop and the types indicate the operations in the loop body are type-safe, whatever the control flow of the loop body is.

For memory safety, it is necessary to annotate the loop with an invariant that demands the value of i is between `length`$(A) - 1$ and 0, inclusive. This invariant, again, allows us to check the safe heap access, while we do not have to perform a data flow analysis that typically iterates before it reaches a fixpoint. It is worth noting that the certifying compiler may perform this analysis as an optimization technique. The loop invariant is discovered during the analysis [50]. This is different than the loop invariants for the type safety, which represent the rules set by the compilation process.

```
foo
   Precondition: r1:handle, r2:integer

   load r2
   store r4

   push 0
   store r3

   Loop:
     INV: r2:handle, r3,r4: integer, r3>0 AND r3<length(r2)
     load r1
     invoke length

     load r2
     jl   Finish

     load r1
     load r2
     hload
     load r4
     add
     store r4

     load r3
     inc
     store r3

     goto Loop

   Finish
     return
```

Figure C.6: Annotated BAL Program

Now we have an annotated BAL program, the safety of which is justified. The actual proof is done through the invocation of VCGen and a general purpose theorem prover.

C.6 A VCGen

Overall, a VCGen expects as input an annotated program and as output formulas in the safety logic known as VCs. VCGen generates VCs using its built-in knowledge of the semantics of the intermediate language and of the safety property. VCs are discharged by a theorem prover at the server side and by a proof-checker, when proofs are attached, at the client side. If we view the whole server side as a big theorem prover that proves the safety properties on the target program, the VCGen can be regarded as a component that deals with the intermediate language semantics in this big theorem prover.

A VCGen generates VCs statically. A common practice is to evaluate the intermediate program symbolically, collecting formulas that must be true at program locations. Then VCGen interprets the safety policy as conditions to be held at different program locations. Finally, the VCGen compares the two set of formulas and computes the VCs. VCs are thus written in a subset of the safety logic that can be discharged without a knowledge of the semantics of the intermediate language.

On Soundness

When talking about the soundness of such a VCGen, normally the soundness results contain two parts: that the VCGen is sound with respect to the operational semantics of the intermediate language and that the derivations of the formulas are sound with respect to their interpretations at any models that can appear at a program location.

On Completeness

Completeness happens only when we have a checkable safety property. Even so, due to the amount of computations that might be involved, approximation is common.

In Table C.7, we list the symbolic representation of the pre- and post state of executing an instruction.

A VCGen will read the annotations generated by the certifying compiler, which include pre- and post- conditions of the program as well as loop invariants. The VCGen invokes the symbolic evaluator to process the program, with the pre- condition and the first instruction of the program as inputs. The returned post- state of the evaluator is used by the VCGen to call the evaluator again with the next instruction. The VCGen keeps applying the evaluator until the end of the basic block, where VCGen may check to see if other program annotations are consistent with the symbolic representation (formulas) of the current state. In this presentation, checking the consistency between formulas is one of the two occasions where VCs are generated, which we refer to as first-category VCs.

Another place VCs are generated is the safety requirements of the individual instructions. For example, when there is an hload instruction, VCs are generated to ensure the `safeheapaccess` are enforced. We call such VCs second-category.

At the basic block level, given an initial symbolic representation of the initial state, a VCGen will generate a symbolic representation of the end of the basic block. In this computation, we have as side effects the VCs generated to fulfill the safety requirements of individual instructions. If the end of the basic block is not a transition instruction, then the VCGen will generate VCs to test if the symbolic representation of the current state implies the annotations, if any, at the beginning of the next basic block in sequence. If the basic block ends with an unconditional jump, the VCGen will generate VCs to see if the current symbolic state implies the annotations at the beginning of the target block. If the last instruction is a conditional jump, the VCGen will add to the current symbolic state the condition for the jump to happen to check the annotations at the beginning of the target block, and add to the current symbolic state the condition for the jump not to happen to check the annotations at beginning of the next block. These rules are described in Figure C.7.

Notation

We write $P \twoheadrightarrow^{\mathrm{I}} Q \Uparrow R$ to represent the (repeated) application of the symbolic evaluator to the instructions `I`, where P is the input symbolic state, Q is the resultant symbolic state, and R is the second-category VCs generated in the process.

C.6.1 Symbolic Evaluator and Second-category VCs

The symbolic evaluator accepts as input an instruction and a symbolic representation of the machine state, typically a formula in the safety logic. The result is another symbolic representation of the machine state. For a machine state s represented by P, if the instruction of concern mutates

Current symbolic state	Instruction Sequence	Verification Conditions
P	no transition; l : ...	$P \to A(l)$
P	jmp l; ...	$P \to A(l)$
P	jeq l; m : ...	$(P \wedge s[0] = s[1]) \to A(l)$
		$(P \wedge s[0] = s[1]) \to A(m)$

Figure C.7: VCGen: End of Basic Block

X, where X is the set of registers, stack or heap locations, then the next machine state in the next state will be represented as: $(\exists X.P(X) \wedge Q(X, X'))$, where $P(X)$ is the previous symbolic state and $Q(X, X')$ is a proposition that characterizes the changes the instruction makes. We call $Q(X, X')$ the characteristic formula of an instruction. Thus the type for the symbolic evaluator `seval` is:

$$\texttt{seval} : P(X) \to \texttt{instr} \to P(X')$$

For example, consider the stack under *ipush 3*. $Q(r)$ will be $s' = 3 :: s$. Suppose $P(s)$ is $s[0] = 1$, then the new symbolic state will be:

$$\exists \texttt{stk}.(\texttt{stk}[0] = 1 \wedge \texttt{stk}' = 3 :: \texttt{stk}) \tag{C.1}$$

Formula (1) thus represents a relation over m', as expected. In practice, (1) is often represented as:

$$\texttt{stk}'[1] = 1 \wedge \texttt{stk}'[0] = 3 \tag{C.2}$$

Formula (2) thus can be used to prove data-flow facts represented as predicates over m'. Suppose the fact is represented as $Q_1(X')$, an invariant at the program location that follows the instruction. We can simply test whether (2) implies $Q_1(X')$.

When we use this post- symbolic state as the pre- symbolic state for the next instruction, we substitute all occurrence of primed variables to un-primed variables.

To present the evaluator, it is sufficient to give the definition of Q in Figure C.2 below. For a machine state $s = (\texttt{fs}, \texttt{heap})$, Q is a predicate over s and s'. We refer to the components of s' as `fs`' and `heap`'. We also refer to the component of `fs` as `stk`, `reg` and `pc`.

PInstruction	Q	Verification Conditions
add	$\texttt{stk}'[0] = (\texttt{stk}[0] + \texttt{stk}[1])$	$\texttt{stk}[0] : \texttt{int} \wedge \texttt{stk}[1] : \texttt{int}$
sub	$\texttt{stk}'[0] = (\texttt{stk}[0] - \texttt{stk}[1])$	$\texttt{stk}[0] : \texttt{int} \wedge \texttt{stk}[1] : \texttt{int}$
mul	$\texttt{stk}'[0] = (\texttt{stk}[0] * \texttt{stk}[1])$	$\texttt{stk}[0] : \texttt{int} \wedge \texttt{stk}[1] : \texttt{int}$
$\texttt{load}\, r_i$	$\texttt{reg}[i]' = \texttt{stk}[0]$	
$\texttt{store}\, r_i$	$\texttt{reg}[i] = \texttt{stk}'[0]$	
$\texttt{hload}$	$\texttt{stk}'[0] = \texttt{newvar}$	$\texttt{stk}[0] : \texttt{handle} \wedge \texttt{stk}[1] : \texttt{int}$ $\wedge\texttt{safeaccess(Stk[0], Stk[1])}$
$\texttt{hstore}$	$\texttt{stk}' = \texttt{tail}^3(\texttt{stk})$	$\texttt{stk}[1] : \texttt{handle} \wedge \texttt{stk}[2] : \texttt{int}$ $\wedge\texttt{safeaccess(Stk[1], Stk[2])}$
$\texttt{alloc}$	$\texttt{stk}'[0] : \mathit{handle} \wedge \texttt{length}(\texttt{stk}[0]') = \texttt{stk}[0]$	$\texttt{stk}[0] : \texttt{int}$
$\texttt{pop}$	$\texttt{stk}' = \texttt{tail}(\texttt{stk})$	

Table C.2: VCGen: Symbolic Evaluator

C.7 Discharge Verification Conditions

A client applies the same VCGen on the same annotated program as a server does, which guarantees the same set of VCs are reproduced. However, the client and the server have different ways to discharge the verification conditions. The server invokes a theorem prover to prove the VCs; the theorem prover should be able to generate independently checkable proofs, which are attached to the mobile program.

In this section, I will discuss the design choices for a server to compute and encode a proof. First of all, we build a formal system from which a proof can be derived. An example of such an axiomatic system is illustrated in the rule for `safeheapaccess`.

$$\frac{\Sigma \vdash a : \texttt{handle} \quad \Sigma \vdash o : \texttt{int} \quad \Sigma \vdash o \leq \text{range}(a)}{\Sigma \vdash \texttt{safeheapaccess}(a, o)}$$

The environment Σ is a set of propositions. The rule defines the condition for `safeheapaccess` to be infered: the address and offset must be of the right type and the offset should not be out of the range, reflecting the semantics of the safe heap access. The judgment $\Sigma \vdash p$ is defined recursively. For example,

$$\frac{p \in \Sigma}{\Sigma \vdash p}$$

This rule says that p is infered if it is in the environment. Another rule says that p is infered if it is implied by the propositions in Σ, testing of such rules will invoke a theorem prover.

Prove the VCs

The VCs generated in most PCC system, including in Section C.6 are from a quantifier-free predicate logic that includes equality, uninterpreted function symbols, simple arithmetic (such as Presburger Arithmetic) and simple data structures. A formula in this logic is decidable by an automated theorem prover (for example, simplify or SVC or CVC, CVC-lite) [7, 68, 6, 19] to prove the verification conditions following these derivation rules. Effort must be made to generate independently checkable proofs.

Represent the Proof

Normally, checking if a proof is correct is quicker than computing the proof itself. Most notably checking a proof avoids the backtracking performed by a theorem prover. And for theorems that are not easily automatically proved, a hand-crafted proof may be checked without much difficulty. In PCC, proofs are encoded in ELF, a logic framework designed for representing various kinds of logics. The type system of ELF can encode the safety logic; a type expression in ELF can be a safety logic formula. By the Curry-Howard isomorphism, a formula in ELF corresponds to a proof of its type. Checking whether a proof is a valid one for a safety logic formula is equivalent to checking whether an ELF formula has a type. Therefore, a type checker in ELF becomes a proof-checker to be used by the client.

In Necula's implementation of PCC, an automated theorem prover is extended to generate ELF format proofs. The extended theorem prover is used by the server to generate certificates. The type checker from ELF is deployed on the client side, as part of the client's trust base, to check the proof.

Conventions

`stk`	concrete operand stack
`reg`	concrete registers
`heap`	concrete heap
`pc`	program counter
`fs`	frame stack
`fr`	concrete frame
`Tr`	a transition system
`S`	Set of states
α	An abstraction function
γ	A concreterization function
Π	Sets of sequence of states
π	sequence of states
$\rightarrow$	transition relation
$\twoheadrightarrow$	postcondition
s	a state
p	concrete predicate