D6.1 Programming model and annotations

Frank Piessens, Bart Jacobs, Jan Smans, Pieter Philippaerts (K.U.Leuven), Isabelle Simplot-Ryl (INRIA Lille), Elisa Chiarani (UNITN)

Document information

Document Number	D6.1
Document Title	Programming model and annotations
Version	4.0
Status	Final
Work Package	WP 6
Deliverable Type	Report
Contractual Date of Delivery	31 January 2010
Actual Date of Delivery	31 January 2010
Responsible Unit	KUL
Contributors	INR, UNITN
Keyword List	Programming models, verification
Dissemination level	PU

Document change record

Version	Date	Status	Author (Unit)	Description
1.0	18 Dec 2009	Draft	Frank Piessens, Bart Jacobs, Jan Smans, Pieter Philippaerts (KUL)	First draft
2.0	7 Jan 2010	Draft	Frank Piessens (KUL), Isabelle Simplot-Ryl (INR)	Addressed comments and feedback from Isabelle Simplot-Ryl
2.1	11 Jan 2010	Draft	Elisa Chiarani (UNITN)	First quality check completed; Some minor remarks
3.0	15 Jan 2010	Draft	Frank Piessens (KUL)	Addressed UNITN remarks
3.1	19 Jan 2010	Draft	Elisa Chiarani (UNITN)	Final quality check completed; Minor remarks
4.0	22 Jan 2010	Final	Frank Piessens (KUL)	Addressed UNITN remarks and finalized document

Executive summary

This document summarizes the work performed in Task 6.1 of Work Package 6 of the SecureChange project funded by the European Commission within the Seventh Framework Programme.

The overall objective of Work Package 6 is the development of verification techniques for evolving systems, with a strong focus on the development time and run time phases of the software lifecycle. Task 6.1 focuses on development time. The objective of Task 6.1 is the development of programming models that can ensure the absence of classes of vulnerabilities. A *programming model* consists of a set of programming guidelines designed to avoid a specific class of vulnerabilities. Source code *annotations* make the programming model explicit, and can support formal verification of compliance with the programming model.

Two important results were obtained in Task 6.1. First, we developed a programming model to avoid so-called *dependency-safety* errors. A dependency-safety violation happens when a particular piece of code fails (throws an exception), and consequently other code that essentially depends on the successful completion of that piece of code is executed. Dependency safety violations are typically caused by improper exception handling. Our programming model allows a developer to make dependencies explicit through annotations, and provides more strict exception handling machinery such that dependency safety can be guaranteed. The theory behind this model has been published in the ECOOP 2009 conference, one of the top programming languages conferences in Europe.

Second, we developed a programming model to avoid safety issues related to the dynamic loading and unloading of modules. These run-time modifications to the code of an application are an important technique to support evolvability of a software product, but they introduce several safety risks, including for instance the creation of dangling function pointers, pointing to code that is already unloaded. We developed a programming model and annotations in a variant of separation logic that allow a developer to verify the safety of the resulting system. This second result is work in progress, and its current status is summarized in a Technical Report of the department of computer science of the K.U.Leuven.

These two models, as well as existing programming models, address specific software quality and security issues and can be used independently. However, an important advantage of the programming model approach is that it is relatively straightforward to combine models. Several models can be used together and can strengthen eachother.

Since the ECOOP paper and the Technical Report mentioned above provide excellent descriptions of these two results, the core of the deliverable consists of these two publications. We first provide a small introduction situating the work in the entire SecureChange project, and a glossary defining the Work Package 6 use of terms. Then we add the two publications as appendices : they describe the core technical contributions.

Index

1 Introduction

The key objective of the SecureChange project is the development of tools and techniques to ensure lifelong compliance to evolving security, privacy and dependability requirements for long-running and evolving software-based systems.

To achieve this objective, the project studies and improves the state-of-the-art in several phases of the software lifecycle, including requirements engineering, architectural design, detailed design, implementation, verification and testing.

The focus of Work Package 6 of the project is on the implementation and verification phases, but it includes also the usage phase, as the question how to securely update and evolve running systems is very much in scope for Work Package 6.

The same terminology (terms such as vulnerability, threat, countermeasure and so forth…) is used in all phases of the software lifecycle, but these terms often have a more specific meaning when specialized to, for instance, the development phase. To avoid ambiguities, this deliverable includes a glossary (see Section 4) of these security-specific terms, and how they are used in Work Package 6.

Work Package 6 has three main lines of work. The first line (consisting of Tasks 6.1 and 6.2) concerns the development of programming models that can ensure the absence of classes of vulnerabilities, and the development of proof-of-concept tools that can verify compliance with a programming model. Such models and tools support secure and correct evolution of the code and changes to the code by making implicit developer assumptions explicit in annotations, and by checking that these assumptions are not violated during code evolution. The first half of this line of work (Task 6.1) is finished, and the results are reported in this deliverable.

The second line of work (Tasks 6.3, 6.4 and 6.5) concerns the development of on-device verification algorithms. This includes techniques to verify the information flow security of dynamically loaded code, and extensions to the Security-by-Contract paradigm for information flow security. This line of work is ongoing, and the first results will be made available at month 18 of the project in Deliverable D6.3.

Finally, the third line of work (Task 6.6) studies the interplay between development-time verification and on-device verification. This line of work will only start at the end of year 2 of the project.

The glossary provided in Section 4 covers the terms relevant to all these three lines of work. But all the other content of this deliverable is specific to the first line of work, development time verification using programming models.

2 Technical results

A *programming model* consists of a set of programming guidelines designed to avoid a specific class of vulnerabilities. Source code *annotations* make the programming model explicit, and can support formal verification of compliance with the programming model.

As such, programming models are very similar to pluggable type systems [3]. A wide variety of programming models or related type systems already existed at the start of the SecureChange project, including systems that deal with concurrency vulnerabilities or bugs [4,5], aliasing bugs [6], or code access security vulnerabilities [7].

In the context of SecureChange, two new results were obtained, and we discuss these in the next two subsections.

2.1 Dependency safety

First, we developed a programming model to avoid so-called *dependency-safety* errors. A dependency-safety violation happens when a particular piece of code fails (throws an exception), and consequently other code that essentially depends on the successful completion of that piece of code is executed. Dependency safety violations are typically caused by improper exception handling. The Ariane 5 crash is a well-known example of the fact that proper exception handling is a major concern during code evolution.

This line of work was ongoing when the SecureChange project started, and we finalized the programming model and annotations in the first five months of the project. Our programming model allows a developer to make dependencies explicit through annotations, and provides more strict exception handling machinery such that dependency safety can be guaranteed.

The theory behind this model is described in detail in an ECOOP 2009 publication [1]. This publication is included verbatim in the Appendix of this deliverable. Ongoing and future work includes a further practical validation, and the study of the interaction between dependency safety and information flow security as studied in the other Tasks in WP 6.

2.2 Safety of unloadable modules

Second, we developed a programming model to avoid safety issues related to the dynamic loading and unloading of modules. These run-time modifications to the code of an application are an important technique to support evolvability of a software product, but they introduce several safety risks, including for instance the creation of dangling function pointers, pointing to code that is already unloaded.

This line of work is still ongoing, but the programming model and annotations are stable. The annotations are done in a variant of separation logic. The current status of this work is described in a Technical Report [2]. This technical report is included verbatim in the Appendix of this deliverable.

3 Conclusion

Task 6.1 of the SecureChange project has completed on schedule. We have designed two programming models and the corresponding annotations, one to ensure dependency safety, and another one to guarantee safety of dynamically loadable and unloadable code.

Task 6.2 will build on this work and develop a prototype verifier for at least one of these programming models, and evaluate it on one of the SecureChange case studies. Two case studies are under consideration: the POPS case study, where provable absence of certain classes of run time exceptions in JavaCard code is an important concern, and the HOMES case study, where the safety of the software on the home gateway (both C code and Java code) in the presence of evolution (for instance loading / unloading of modules) is a concern.

4 Glossary

Attack: the execution of a program or program module with *exploit* input.

Contract: a specification of the security-relevant behaviour of a program or program *module*. Examples include: *information flow* contracts that specify how information can flow from inputs to outputs, or access control contracts that specify the possible traces of security-relevant events that a program could generate.

Countermeasure: a technique to prevent, remove or tolerate *vulnerabilities*.

Exploit: A set of inputs to, or an interaction with a program or program *module* that triggers a *vulnerability*, and hence makes the program (*module*) deviate from its *contract*.

Information flow: how outputs of a program (directly or indirectly) depend on inputs of a program.

Matching (policy-contract): the process of checking whether a *contract* is compatible with a *policy*: is everything that is specified as possible security-relevant behaviour by the *contract* also allowed by the *policy*. For access control *contracts* and *policies*, where security-relevant behaviour can be formalized as a set of allowable traces of security-relevant events, matching corresponds to set inclusion.

Modular verification: a *verification* process that verifies each *module* separately. While verifying a *module*, the *verification* relies only on the *contracts* of dependent *modules*, not on their implementation. Modular verification can make the *verification* process more scalable to large programs, and can make it lighter for new versions of the system.

Module: a logically self-contained part of a program. Packages, classes, or methods are examples of modules of different granularity. Modules have an implementation and a specification. The security-relevant part of the specification is called the *contract*.

Policy: a specification of the security constraints that a deployment context wishes to impose upon a program. Examples include: *information flow* policies that specify how information is allowed to flow from inputs to outputs, or access control policies that specify the traces of security-relevant events that a program is allowed to generate.

Programming model: a programming model is a set of guidelines on how to use the features of a given programming language. These guidelines will typically be designed in such a way that they avoid the introduction of certain classes of vulnerabilities in the code.

Verification (code-contract): the process of checking the compliance of a program or program *module* with its *contract*.

Vulnerability: a vulnerability is a security-relevant bug in a program, i.e. the program is not satisfying its *contract*.

References

[1] Bart Jacobs, Frank Piessens, Failboxes: Provably safe exception handling, ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genova, Italy, July 6-10, 2009, Proceedings, volume 5653, pages 470-494, Genova, 6-10 July 2009.

[2] Bart Jacobs, Jan Smans, Frank Piessens, Verification of unloadable C modules - status report, Technical Report (CW Reports), volume CW567, Department of Computer Science, K.U.Leuven, October 2009.

[3] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. SIGPLAN Not., 41(10):57–74, 2006.

[4] Bart Jacobs, K. Rustan M. Leino, Frank Piessens, andWolfram Schulte. Safe concurrency for aggregate objects with invariants. In Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods, pages 137–146. IEEE Computer Society, 2005.

[5] Boyapati, C., Lee, R., and Rinard, M. 2002. Ownership types for safe programming: Preventing data races and deadlocks. In Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA), S. Matsuoka, Ed. SIGPLAN Notices 37, 11, 211-230

[6] David Clarke, John Potter, James Noble, Ownership types for flexible alias protection, SIGPLAN notices, volume 33, issue 10, pages 48-64, October 1998

[7] Jan Smans, Bart Jacobs, and Frank Piessens. Static verification of code access security policy compliance of .NET applications. Journal of Object Technology, 5(3), April 2006.

5 APPENDIX

This appendix contains the two papers:

- Bart Jacobs, Frank Piessens, Failboxes: Provably safe exception handling, ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genova, Italy, July 6-10, 2009, Proceedings, volume 5653, pages 470-494, Genova, 6-10 July 2009.
- Bart Jacobs, Jan Smans, Frank Piessens, Verification of unloadable C modules - status report, Technical Report (CW Reports), volume CW567, Department of Computer Science, K.U.Leuven, October 2009.

Failboxes: Provably Safe Exception Handling*

Bart Jacobs** and Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{bart.jacobs,frank.piessens}@cs.kuleuven.be

Abstract. The primary goal of exception mechanisms is to help ensure that when an operation fails, code that depends on the operation's successful completion is not executed (a property we call *dependency safety*). However, the exception mechanisms of current mainstream programming languages make it hard to achieve dependency safety, in particular when objects manipulated inside a try block outlive the try block.
Many programming languages, mechanisms and paradigms have been proposed that address this issue. However, they all depart significantly from current practice. In this paper, we propose a language mechanism called failboxes. When applied correctly, failboxes have no significant impact on the structure, the semantics, or the performance of the program, other than to eliminate the executions that violate dependency safety.
Specifically, programmers may create failboxes dynamically and execute blocks of code in them. Once any such block fails, all subsequent attempts to execute code in the failbox will fail. To achieve dependency safety, programmers simply need to ensure that if an operation B depends on an operation A, then A and B are executed in the same failbox. Furthermore, failboxes help fix the unsafe interaction between locks and exceptions and they enable safe cancellation and robust resource cleanup. Finally, the Fail Fast mechanism prevents liveness issues when a thread is waiting on a failed thread.
We give a formal syntax and semantics of the new constructs, and prove dependency safety. Furthermore, to show that the new constructs are easy to reason about, we propose proof rules in separation logic. The theory has been machine-checked.

1 Introduction

If a program is seen as a state machine, a programmer's job may be seen as writing code to deal with each of the states that the program may reach. However, programmer time is limited and some states are less likely to occur during production than others. Therefore, in many projects it is useful to designate the most unlikely states as *failure states* and to deal with all failure states in a uniform way, while writing specific code only for non-failure (or *normal*) states.

An extreme form of this approach is to simply ignore failure states and not care what the program does when it reaches a failure state (i.e., when it *fails*).

* We used the term *subsystems* in preliminary work.

** Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

This is often what happens when subroutines indicate failure conditions as special return values, and programmers have no time to write code at call sites to check for them.

A major problem with this approach is that it is *unsafe*: a failure may lead to the violation of any and all of the program's intended safety properties. Specifically, the approach violates *dependency safety*, the property which says that when an operation fails, code that depends on the operation's successful completion is not executed.

To fix this, modern programming languages offer constructs that make it easy for programmers to indicate that a state is a failure state, and deal with failure states by terminating the program by default. The underlying assumption is that termination is always safe. For example, in Java, a failure state is indicated by throwing an unchecked exception. We will focus on the Java language in this paper; the related work section discusses other languages.

Whereas by default, when a program throws an exception it terminates immediately, the programmer can override this default through the use of try-catch statements and try-finally statements. Furthermore, in a multithreaded program, when a thread's main method completes abruptly (i.e., an exception was thrown and not caught during its execution), only that thread, not the entire program, is terminated. Also, when a synchronized block's body completes abruptly, the lock is released before the exception is propagated further.

These deviations from strict termination behavior are useful and are used for two reasons. Firstly, not all exceptions indicate failure. Sometimes, programmers throw and catch exceptions to implement the program's functional behavior. Typically, in Java, checked exceptions are used for this. Secondly, programmers sometimes wish to increase the program's robustness by not considering the program to be a single unit of failure but rather by identifying multiple smaller units of failure. Common examples are extensible programs, where poorly written or malicious plugins (such as applets or servlets) should not affect the base system; and command-processing applications (such as request-response-based servers, GUI applications, or command-line shells) where a failure during the processing of a command should simply cause an error response to be returned, while continuing to process other commands normally.

However, by continuing to execute after a failure, the risk of safety violations reappears. In particular, safety violations are likely if the code that fails leaves a data structure in an inconsistent state and this data structure is then accessed during execution of a finally block or after the exception is caught, or by another thread. In other words, there is a safety risk if a try block manipulates an object that outlives the try block. More generally, if we define *dependency safety* as the property that if an operation fails, no code that depends on the operation's successful completion is executed, then dependency safety may be violated if pieces of code outside a try block depend on particular pieces of code inside the try block either not executing at all or executing to completion successfully. This is the problem we address in this paper.

To remedy this, we propose a language mechanism called *failboxes*. Programmers may create failboxes dynamically and execute blocks of code in them. Once any such block fails, all subsequent attempts to execute code in the failbox will fail. To achieve dependency safety, programmers simply need to ensure that if an operation B depends on an operation A, then A and B are executed in the same failbox. Furthermore, failboxes help fix the unsafe interaction between locks and exceptions and they enable safe cancellation and robust resource cleanup. Finally, the Fail Fast mechanism prevents liveness issues in the presence of failure in cooperating concurrent computations.

Failboxes are very lightweight: a failbox can be implemented as an object with a boolean field indicating if the failbox has failed, and a parent pointer. Executing a code block in a failbox essentially means that before and after executing the block, the thread-local variable that designates the current failbox is updated, and before a failbox is made current, it is checked that it has not failed.

We give a formal syntax and semantics of the new constructs, and prove dependency safety. Furthermore, to show that the new constructs are easy to reason about, we propose separation logic proof rules and prove their soundness.

The rest of the paper is structured as follows. In Section 2, we illustrate the problem with an example and discuss existing approaches. In Section 3, we introduce failboxes. We show additional aspects and benefits of the approach for multithreaded programs in Section 4. Section 5 briefly discusses how the approach enables safe cancellation and robust compensation. To show that it is easy to reason about the new constructs, we propose separation logic proof rules for the envisaged usage patterns in Section 6. We end the paper with sections on implementation issues (Section 7), related work (Section 8), and a conclusion (Section 9).

The theory of this paper has been machine-checked using the Coq proof assistant [12].

2 Problem Statement

Consider the example program in Figure 1. It shows a program that continuously receives commands and processes them. The code for processing commands is not shown, except that it involves calls of *compute* and calls of *addEntry* on a *Database* object *db* that is shared across all command executions. If the processing of a command fails, e.g. because it requires too much memory, the exception is caught, an error message is shown to the user, and the next command is received.

This program is unsafe. Specifically, some executions of this program violate the intended safety property that at the start of each loop iteration, object *db* is *consistent*, i.e., satisfies the property that *count* is not greater than the length of *entries*. In particular, consider an execution where method *addEntry* is called in a state where *entries* is full. This means *count* equals *entries.length*. As a result, after incrementing *count*, *addEntry* will attempt to allocate a new,

```
class Database {
  int count;
  int[] entries := new int[10];
  /* invariant: count ≤ entries.length */
  void addEntry(int entry) {
    count++;
    if (count = entries.length + 1) {
      int[] es := new int[count * 2];   // *** A ***
      System.arraycopy(entries, 0, es, 0, entries.length);
      entries := es;
    }
    entries[count − 1] := entry;   // *** B ***
  } ... }
class Program {
  public static void main(String[] args) {
    Database db := new Database();
    while (true)
      /* invariant: db is consistent */
    {
      String cmd := readCommand();
      try {
        ··· compute(cmd); ···
        ··· db.addEntry(···); ···
      } catch (Throwable e) { showErrorMessage(e); }
  }} ... }
```

Fig. 1. An unsafe program

larger array. Now assume there is not enough memory for this new array and an *OutOfMemoryError* occurs at location A. At this point, *count* is greater than the length of *entries* and the *Database* object is inconsistent. Next, the exception is caught in method *main* and the loop is continued, violating the safety property.

Note: In this case, the safety violation results in an *ArrayIndexOutOfBounds-Exception* at location B in each subsequent call of *addEntry*; however, in general, safety violations might remain undetected and lead to data corruption, incorrect results, or sending incorrect commands to hardware devices.

The following approaches exist to deal with this complication:

– **Never catch unchecked exceptions.** Never catching unchecked exceptions makes it easier to preserve safety properties, since the many implicit control flow paths created by catching unchecked exceptions are avoided. However, catching unchecked exceptions can be useful, as in the example. Note also that **try-finally** blocks are equivalent to **try-catch** blocks that catch unchecked exceptions; specifically, assuming S_1 does not jump out of

the **try** block, a statement

$$\textbf{try } \{ \ S_1 \ \} \textbf{ finally } \{ \ S_2 \ \}$$

is equivalent to

$$\textbf{try } \{ \ S_1 \ \} \textbf{ catch } (\mathit{Throwable}\ t) \ \{ \ S_2 \textbf{ throw } t; \ \} \ S_2$$

and is subject to the same complication: S_2 might depend on the successful completion of certain sub-computations within S_1. Never catching unchecked exceptions would imply never using **try-finally** blocks, or modifying their semantics so that they ignore unchecked exceptions. The semantics of synchronized blocks would need to be updated similarly.

- **Always maintain consistency.** It is often possible to ensure that objects used across try-catch blocks, like the *Database* object in the example, are in a consistent state at all times. Often it is sufficient to reorder assignments; e.g., in the example, moving the count increment after the assignment to *entries* preserves consistency. Another approach is to use a functional programming-like approach, where a new object state is built up separately and then installed into the object using a single assignment. In the example, method *addEntry* would return a new *Database* object rather than updating the existing one. Yet another approach is to use transaction-like technologies, such as software transactional memory [19, 5]. However, these approaches either require the programmer to perform non-trivial additional reasoning and/or programming work, or impose a potentially significant performance overhead.
- **Never fail during critical sections.** It might be possible in some cases to guarantee absence of failure at points where failure would violate safety. This requires careful programming to avoid operations that might encounter resource or implementation limitations, such as heap or stack memory allocations or operations on bounded integers, or to move these operations out of the critical section. Furthermore, this might require virtual machine support if the virtual machine may perform resource allocations implicitly. For example, the .NET Framework's JIT compiler may allocate memory at any time to store a newly compiled piece of code. Therefore, starting with version 2, the .NET Framework offers constructs to "prepare" a piece of code that must execute without failure [21]. However, this approach imposes a significant burden on the programmer.
- **Ensure dependent code is not executed.** In this approach, steps are taken to ensure that if a computation fails with an unchecked exception, then no computations that depend on the failed computation's successful completion ever get to run. There are at least two ways to achieve this:
  - **Use separate threads.** In this approach, threads are adopted as the units of failure. Within a thread, unchecked exceptions are never caught; that is, an exception in the thread causes the entire thread to die. All data structures are local to threads. Instead of running a block of code in a try-catch block, it is run in a separate thread. During this time, the

original thread waits for the termination of the child thread; additionally, the original thread may accept messages on a message queue. If the child thread needs to perform an operation whose failure should cause the parent thread to fail (such as an *addEntry* call on the *Database* object), the child thread may perform a remote procedure call into the parent thread via the parent thread's message queue. This is more or less the approach used in operating systems, in the Erlang language [1], and in the SCOOP multithreading approach for Eiffel [17].

- **Guard dependent code manually.** The programmer can manually arrange to ensure that dependent code is not executed. For example, the programmer could associate a boolean flag with each object used across try-catch blocks that tracks whether the object is in a consistent state, and check this flag before accessing the object [13]. If the flag is false, an exception is thrown.

In this paper we present a new approach in the fourth category, which, like the use of separate threads and manually guarding dependent code, supports catching exceptions and does not require that consistency be maintained always or that failures be avoided, but which has less programming and run-time overhead than the use of separate threads and which has less programming overhead than manually guarding dependent code.

3 Failboxes

In our approach, the language is extended with a notion of *failboxes*. Constructs are added for creating a new failbox and for running a piece of code in a designated failbox. As soon as one such piece of code fails (i.e., completes abruptly with an unchecked exception), any subsequent attempt to run code in the failbox fails. To ensure dependency safety, the programmer simply needs to ensure that if a computation B depends on a computation A, then A and B run in the same failbox.

To facilitate composition of program modules, failboxes are ordered hierarchically. When creating a new failbox, a parent may be specified. If an exception occurs in a failbox, both it and its transitive children are marked as failed.

3.1 Syntax and semantics

The syntax of the new constructs is as follows:

$$
\begin{aligned}
s ::= \ & \ldots \\
& \mid x := \textbf{currentfb}; \mid x := \textbf{newfb}; \mid x' := \textbf{newfb}(x); \\
& \mid \textbf{enter}\ (x)\ \{\ \overline{s}\ \}\ \textbf{catch}\ \{\ \overline{s}'\ \}
\end{aligned}
$$

where s ranges over statements, $\overline{s}$ ranges over sequences (i.e., sequential compositions) of statements, and x and x' range over local variable names.

Note: For simplicity, we ignore checked exceptions and exception objects in the formal developments.

A program state is a tuple of the form

$$(L, \Sigma, \Phi, T)$$

where L, the *lock map*, is a partial function that contains a pair (o, t) if thread t holds the lock of object o; Σ is a partial function that maps each allocated failbox to its parent failbox (and a root failbox to itself); Φ is the set of *failed failboxes*; and T is a partial function that maps each thread to its current state. (We omit the heap since our constructs do not interact with it.)

A thread state is a tuple of the form

$$(f, V, \overline{s}, \overline{b}, \overline{F})$$

where f is the thread's *current failbox*, V is a total function that maps each variable name to a value, $\overline{s}$, the *continuation*, is the sequence of statements to be executed by the thread, $\overline{b}$ is the sequence of *enclosing blocks*, and $\overline{F}$ is the sequence of enclosing activation records.

The syntax of an enclosing block is as follows:

$$b ::= \textbf{enter}\ (f)\ \textbf{catch}\ \{\ \overline{s}\ \}\ \overline{s}' \mid \textbf{synchronized}\ (o);\ \overline{s}$$

where an enclosing enter block records the failbox f that was current prior to the enter statement (*not* the failbox that was entered), the catch block body $\overline{s}$, and the statements $\overline{s}'$ that are to be executed after completion of the enter statement; and an enclosing synchronized block records the object o whose lock was acquired, and the statements $\overline{s}$ that are to be executed after completion of the synchronized statement.

In the initial program state of a program with main method body $\overline{s}$, the lock map is empty, there is a single failbox f, whose parent is itself, no failbox is marked as failed, and there is one thread t whose current failbox is f; all of the thread's local variables are bound to **null**, and it has no enclosing blocks and no enclosing activation records:

$$\frac{\mathsf{main}\ \overline{s}}{\mathsf{initial}\ (\emptyset, \{(f, f)\}, \emptyset, \{(t, (f, (\lambda x.\textbf{null}), \overline{s}, \epsilon, \epsilon))\})}$$

The statement $x := \textbf{currentfb}$; assigns the current failbox to variable x:

$$\textsc{CurrentFB}\quad \frac{(t, (f, V, x := \textbf{currentfb};\ \overline{s}, \overline{b}, \overline{F})) \in T}{(L, \Sigma, \Phi, T) \to (L, \Sigma, \Phi, T(t := (f, V(x := f), \overline{s}, \overline{b}, \overline{F})))}$$

The statement $x := \textbf{newfb}$; creates a new root failbox and assigns it to x:

$$\textsc{NewFB-Root}\quad \frac{(t, (f, V, x := \textbf{newfb};\ \overline{s}, \overline{b}, \overline{F})) \in T \qquad f' \notin \mathrm{dom}(\Sigma) \qquad \Sigma' = \Sigma(f' := f')}{(L, \Sigma, \Phi, T) \to (L, \Sigma', \Phi, T(t := (f, V(x := f'), \overline{s}, \overline{b}, \overline{F})))}$$

If x is bound to a failbox f' and f' is not marked as failed, the statement $x' := \mathbf{newfb}(x)$; creates a new child failbox of f' and assigns it to x':

$$\textsc{NewFB-Child}\quad \frac{(t, (f, V, x' := \mathbf{newfb}(x);\ \overline{s}, \overline{b}, \overline{F})) \in T \qquad V(x) = f' \qquad f' \notin \Phi \qquad f'' \notin \mathrm{dom}(\Sigma) \qquad \Sigma' = \Sigma(f'' := f')}{(L, \Sigma, \Phi, T) \rightarrow (L, \Sigma', \Phi, T(t := (f, V(x' := f''), \overline{s}, \overline{b}, \overline{F})))}$$

If x is bound to a failbox f', and f' is not marked as failed, then statement **enter** (x) { $\overline{s}'$ } **catch** { $\overline{s}''$ } records the current failbox, the catch block body $\overline{s}'$, and the current continuation in a new enclosing block, makes f' the current failbox, and starts executing the enter block body $\overline{s}'$:

$$\textsc{Enter}\quad \frac{(t, (f, V, \mathbf{enter}\ (x)\ \{\ \overline{s}'\ \}\ \mathbf{catch}\ \{\ \overline{s}''\ \}\ \overline{s}, \overline{b}, \overline{F})) \in T \qquad V(x) = f' \qquad f' \notin \Phi \qquad \overline{b}' = (\mathbf{enter}\ (f)\ \mathbf{catch}\ \{\ \overline{s}''\ \}\ \overline{s}) \cdot \overline{b}}{(L, \Sigma, \Phi, T) \rightarrow (L, \Sigma, \Phi, T(t := (f', V, \overline{s}', \overline{b}', \overline{F})))}$$

On normal completion of an enter block body, the former current failbox is restored and the catch block is skipped, provided that the former current failbox is not marked as failed:

$$\textsc{Enter-Complete-Normal}\quad \frac{(t, (f, V, \epsilon, (\mathbf{enter}\ (f')\ \mathbf{catch}\ \{\ \overline{s}'\ \}\ \overline{s}'') \cdot \overline{b}, \overline{F})) \in T \qquad f' \notin \Phi}{(L, \Sigma, \Phi, T) \rightarrow (L, \Sigma, \Phi, T(t := (f', V, \overline{s}'', \overline{b}, \overline{F})))}$$

where ϵ denotes the empty sequence.

We model the occurrence of an exception as the replacement of the current continuation with a throw statement. An exception can occur at any time; this reflects the fact that in Java a *virtual machine error* can be thrown at any time [10, §11.3.2].

$$\textsc{Fail}\quad \frac{(t, (f, V, \overline{s}, \overline{b}, \overline{F})) \in T \qquad \overline{s} \neq \mathbf{throw};}{(L, \Sigma, \Phi, T) \rightarrow (L, \Sigma, \Phi, T(t := (f, V, \mathbf{throw};, \overline{b}, \overline{F})))}$$

If variable x is not bound to a failbox, or it is bound to a failbox but the failbox is marked as failed, then both $x' := \mathbf{newfb}(x)$; and **enter** (x) { $\overline{s}$ } **catch** { $\overline{s}'$ } throw an exception (of type *FailboxException*); this is covered by rule Fail.

On abrupt completion of an enter block body with an exception, the current failbox and its descendants are marked as failed, the former current failbox is restored, and the catch block is executed, provided the former current failbox is not marked as failed:

$$\textsc{Enter-Complete-Abrupt}\quad \frac{(t, (f, V, \mathbf{throw};, (\mathbf{enter}\ (f')\ \mathbf{catch}\ \{\ \overline{s}'\ \}\ \overline{s}'') \cdot \overline{b}, \overline{F})) \in T \qquad \Phi' = \Phi \cup (\Sigma^{-1})^*(f) \qquad f' \notin \Phi'}{(L, \Sigma, \Phi, T) \rightarrow (L, \Sigma, \Phi', T(t := (f', V, \overline{s}'\ \overline{s}'', \overline{b}, \overline{F})))}$$

where $(\Sigma^{-1})^*(f)$ denotes the set of f's descendants, including f itself.

On normal completion of an enter block body, if the former current failbox is marked as failed, it is restored but the catch block is skipped and a *FailboxException* exception is thrown:

$$\textsc{Enter-Complete-Normal-Fail}\quad \frac{(t, (f, V, \epsilon, (\textbf{enter}\ (f')\ \textbf{catch}\ \{\ \overline{s}'\ \}\ \overline{s}'') \cdot \overline{b}, \overline{F})) \in T \qquad f' \in \Phi}{(L, \Sigma, \Phi, T) \rightarrow (L, \Sigma, \Phi, T(t := (f', V, \textbf{throw};, \overline{b}, \overline{F})))}$$

On abrupt completion of an enter block body with an exception, if after marking the current failbox as failed, the former current failbox is marked as failed, the former current failbox is restored but the catch block is skipped and a *FailboxException* exception is thrown:

$$\textsc{Enter-Complete-Abrupt-Fail}\quad \frac{\begin{array}{c}(t, (f, V, \textbf{throw};, (\textbf{enter}\ (f')\ \textbf{catch}\ \{\ \overline{s}'\ \}\ \overline{s}'') \cdot \overline{b}, \overline{F})) \in T \\ \Phi' = \Phi \cup (\Sigma^{-1})^*(f) \qquad f' \in \Phi'\end{array}}{(L, \Sigma, \Phi, T) \rightarrow (L, \Sigma, \Phi', T(t := (f', V, \textbf{throw};, \overline{b}, \overline{F})))}$$

3.2 Syntactic sugar

We remove try-catch statements and try-finally statements from the language as separate statements. Instead, we define them as syntactic sugar over the new constructs. Specifically, the statement

$$\textbf{try}\ \{\ \overline{s}\ \}\ \textbf{catch}\ \{\ \overline{s}'\ \}$$

is defined as

$$x := \textbf{currentfb};\ x' := \textbf{newfb}(x);\ \textbf{enter}\ (x')\ \{\ \overline{s}\ \}\ \textbf{catch}\ \{\ \overline{s}'\ \}$$

where x and x' are fresh. That is, a try-catch statement executes the try block in a new child failbox of the current failbox.

The statement

$$\textbf{try}\ \{\ \overline{s}\ \}\ \textbf{finally}\ \{\ \overline{s}'\ \}$$

is defined as

$$\textbf{try}\ \{\ \overline{s}\ \}\ \textbf{catch}\ \{\ \overline{s}'\ \textbf{throw};\ \}\ \overline{s}'$$

This means that a try-finally statement executes its try block in a new child failbox of the current failbox.

Furthermore, we define the following shorthands:

$$\begin{array}{c}\textbf{enter}\ (x)\ \{\ \overline{s}\ \}\ \equiv\ \textbf{enter}\ (x)\ \{\ \overline{s}\ \}\ \textbf{catch}\ \{\ \textbf{throw};\ \} \\ \textbf{reenter}\ (x)\ \{\ \overline{s}\ \}\ \equiv\ \textbf{enter}\ (x)\ \{\ \overline{s}\ \}\ \textbf{catch}\ \{\}\end{array}$$

In words, an enter statement propagates exceptions, and a reenter statement does not. Note: in real implementations, a reenter statement would not cause exception information to be lost, since the exception object would be associated with the failbox at the time the failbox is marked as failed, and an API would be provided to retrieve the stored exception object of a failed failbox.

3.3 Terminology

We use the following terminology: We say that an event in a thread t occurs *in a failbox* f or a statement is executed (or executes) in f if the event occurs or the statement execution starts at a time when f is the current failbox of t. We say that *a failure occurs* in t when an unchecked exception is thrown (i.e., the continuation of t is a throw statement). We say that a statement execution *fails* if it completes abruptly because of an unchecked exception. We say that a failbox f *fails* when a failure occurs in f. We say that an execution step *enters* a failbox f if f is the current failbox after the step and was not the current failbox before the step. Similarly, we say that an execution step *leaves* a failbox f if is not the current failbox after the step and was the current failbox before the step.

3.4 Example

The approach is illustrated and motivated by the example in Figure 2. (Note: In the examples we use a more conventional syntax.) It shows how the unsafe program of Figure 1 can be made safe using failboxes. A failbox f is created and then both the main loop and calls of *addEntry* are executed in f. This ensures that if a call of *addEntry* fails, the main loop terminates.

The example motivates why on entry to a try block, the failbox in which the try-catch statement executes is no longer considered the current failbox. This ensures that failures in method *compute* are properly caught by the try-catch statement, and do not cause the program to terminate.

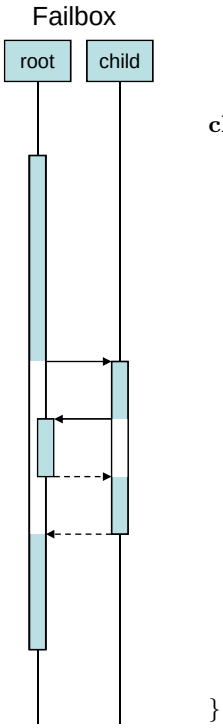
4 Multithreading

4.1 Synchronized statements: safety issues

One common way that the strict termination approach of dealing with failures is overridden, is through the use of **synchronized** blocks. A **synchronized** (o) S block in Java acquires the lock of object o, executes statement S, and then releases the lock of o, even if S failed. This helps prevent deadlocks, but it creates a safety risk. In particular, if a failure occurs while o is inconsistent, the commonly intended safety property that shared objects whose lock is not held are consistent, is violated.

The problem is illustrated by the example program in Figure 3. It is a multi-threaded version of the original example in Figure 1. Rather than processing each command before receiving the next command, the program receives a command, spawns a thread to process it, and immediately receives the next command. The *Database* object is shared by all command processing threads; accesses to the object are synchronized using a **synchronized** block.

This program is unsafe. In particular, in some executions, the intended safety property that whenever a shared object's lock is not held by any thread, the object is consistent, is violated. This property is relied on to guarantee that method




```
class Program {
  public static void main(String[] args) {
    Failbox f := Failbox.getCurrent();
    Database db := new Database();
    while (true)
      /* invariant: db is consistent */
    {
      String cmd := readCommand();
      try {
        ··· compute(cmd); ···
        enter (f) {
          db.addEntry(···);
        }
        ···
      } catch (Throwable e) {
        showErrorMessage(e);
      }
    }
  }
  ···
}
```

Fig. 2. The example of Figure 1, fixed using failboxes. When an *addEntry* call fails, failbox f is marked as failed. When control subsequently exits the try block, this is considered an attempt to enter f; therefore, a *FailboxException* is thrown. As a result, the catch block is skipped, the loop is exited, and the program terminates safely. The sequence diagram shows the failbox transitions.

addEntry is called only on objects that are consistent. Specifically, suppose a failure occurs in method *addEntry* while the *Database* object is inconsistent. This causes the lock to be released. Subsequent command processing threads that acquire the lock will then see the *Database* object in an inconsistent state.

4.2 Failboxes approach for safe synchronized statements

Failboxes can be used to write safe lock-based multithreaded programs, by associating each shared object with a failbox and running the code that accesses a shared object within the associated failbox. This way, when a failure occurs, the failbox is marked as failed, so that when another thread subsequently attempts to enter the failbox in order to access the object, an exception is thrown and the thread is prevented from seeing inconsistent state. The modified safety property is that whenever no thread holds a shared object's lock, either the object is consistent or its associated failbox is marked as failed.

```
class Program {
  public static void main(String[] args) {
    final Database db := new Database();
    while (true) {
      final String cmd := readCommand();
      new Thread() {
        public void run() {
          try {
            ··· compute(cmd); ···
            ··· synchronized (db) { db.addEntry(···); } ···
          } catch (Throwable e) { showErrorMessage(e); }
        }
      }.start();
    }
  }
  ···
}
```

Fig. 3. An unsafe program. A failure in *compute* is handled correctly, but if a failure occurs in method *addEntry* while the *Database* object is inconsistent, the object's lock is released, causing threads that subsequently acquire the lock to see the object in an unexpected state, violating safety.

The approach is illustrated in Figure 4. It is the example of Figure 3, made safe using failboxes. Specifically, the example uses an enter statement to execute the *addEntry* calls in the main thread's root failbox. When an *addEntry* call fails, this failbox is marked as failed before the lock is released. When another thread subsequently acquires the lock and attempts to enter the failbox, an exception is thrown, so that the thread is prevented from unsafely calling *addEntry*.

4.3 Multithreaded failboxes

In a multithreaded program, it is possible for computations in multiple threads to be executing in the same failbox f concurrently. If this happens, we say f is multithreaded. The question then arises as to what happens when one of these computations fails. There are two distinct concerns involved in this matter: preserving the program's intended safety properties, and ensuring useful progress.

4.4 Multithreaded failboxes: Safety

In a well-written program, a failure in one thread should not have safety implications for operations executing concurrently in other threads. Specifically, in a *data-race-free* program, where the program synchronizes accesses to shared memory using the language's synchronization constructs, an operation can see

```
class Program {
  public static void main(String[] args) {
    final Failbox f := Failbox.getCurrent();
    final Database db := new Database();
    while (true) {
      final String cmd := readCommand();
      new Thread() {
        public void run() {
          try {
            ··· compute(cmd); ···
            ··· synchronized (db) { enter (f) { db.addEntry(···); } } ···
          } catch (Throwable e) { showErrorMessage(e); }
        }
      }.startInCurrentFailbox();
    }
  }
  ···
}
```

Fig. 4. The example of Figure 3, made safe using failboxes. If a call of *addEntry* fails, failbox f is marked as failed and subsequent attempts by other threads to enter the failbox will fail. Furthermore, by the Fail Fast feature, a *stop* f signal is sent to all threads running in the failed failbox f or a descendant of f. In the example, this means the program terminates.

the data that was being manipulated by a computation that failed only if the operation is not concurrent with the failure, i.e., the operation was synchronized with the failed computation. (Formally, the failure *happens-before* the operation.) Therefore, to ensure safety, it is sufficient that synchronization constructs perform the necessary failboxes bookkeeping to ensure that if a failure happens in a failbox, no operation that is ordered after this failure through synchronization runs in this failbox. To achieve this, we specify the semantics of synchronized statements with respect to failboxes as follows: after acquiring the lock, the statement checks that the current failbox has not failed; otherwise, it throws a *FailboxException*. Furthermore, before releasing the lock, if the body completed abruptly with an exception, the current failbox is marked as failed. The step rules are shown in Figure 5.

4.5 Properties

We are now ready to state and sketch the proof of the main properties of the failboxes approach.

We first define some terms. An *execution* is a finite or countably infinite sequence of program states. An *execution point* is a nonnegative integer that

$$\textsc{Synchronized}\quad \frac{(t, (f, V, \textbf{synchronized}\ (x)\ \{\ \overline{s}'\ \}\ \overline{s}, \overline{b}, \overline{F})) \in T \qquad V(x) = o \qquad o \notin \mathrm{dom}(L) \qquad f \notin \Phi \qquad \overline{b}' = (\textbf{synchronized}\ (o);\ \overline{s}) \cdot \overline{b}}{(L, \Sigma, \Phi, T) \overset{t:\mathsf{acq}(o)}{\rightarrow} (L(o := t), \Sigma, \Phi, T(t := (f, V, \overline{s}', \overline{b}', \overline{F})))}$$

$$\textsc{Synchronized-Reentrant}\quad \frac{(t, (f, V, \textbf{synchronized}\ (x)\ \{\ \overline{s}'\ \}\ \overline{s}, \overline{b}, \overline{F})) \in T \qquad V(x) = o \qquad (o, t) \in L}{(L, \Sigma, \Phi, T) \rightarrow (L, \Sigma, \Phi, T(t := (f, V, \overline{s}'\ \overline{s}, \overline{b}, \overline{F})))}$$

$$\textsc{Synchronized-Complete-Normal}\quad \frac{(t, (f, V, \epsilon, (\textbf{synchronized}\ (o);\ \overline{s}) \cdot \overline{b}, \overline{F})) \in T}{(L, \Sigma, \Phi, T) \overset{t:\mathsf{rel}(o)}{\rightarrow} (L \setminus \{(o, t)\}, \Sigma, \Phi, T(t := (f, V, \overline{s}, \overline{b}, \overline{F})))}$$

$$\textsc{Synchronized-Complete-Abrupt}\quad \frac{(t, (f, V, \textbf{throw};, (\textbf{synchronized}\ (o);\ \overline{s}) \cdot \overline{b}, \overline{F})) \in T \qquad \Phi' = \Phi \cup (\Sigma^{-1})^*(f) \qquad T' = T(t := (f, V, \textbf{throw};, \overline{b}, \overline{F}))}{(L, \Sigma, \Phi, T) \overset{t:\mathsf{rel}(o)}{\rightarrow} (L \setminus \{(o, t)\}, \Sigma, \Phi', T')}$$

Fig. 5. Step rules for synchronized statements

serves as an index into an execution. A *thread execution point* (k, t) is a pair of an execution point k and a thread identifier t.

Definition 1 (Happens-Before). *The happens-before relation* $\overset{\mathrm{hb}_E}{\rightarrow}$ *on thread execution points of an execution* $E = C_0, C_1, \ldots$ *is the smallest transitive relation that satisfies the following properties:*

- *Any thread execution point of a thread* t *happens-before any subsequent thread execution point of* t

$$k_1 < k_2 \Rightarrow (k_1, t) \overset{\mathrm{hb}_E}{\rightarrow} (k_2, t)$$

- *If execution step* k_1 *is a release of some lock* o *by some thread* t_1*, and subsequent execution step* k_2 *is an acquire of* o *by some thread* t_2*, then* (k_1, t_1) *happens-before* $(k_2 + 1, t_2)$

$$C_{k_1} \overset{t_1:\mathsf{rel}(o)}{\rightarrow} C_{k_1+1} \Rightarrow C_{k_2} \overset{t_2:\mathsf{acq}(o)}{\rightarrow} C_{k_2+1} \Rightarrow k_1 < k_2 \Rightarrow (k_1, t_1) \overset{\mathrm{hb}_E}{\rightarrow} (k_2 + 1, t_2)$$

- *If at execution step* k *thread* t *starts a new thread* t' *(see Figure 6), then* (k, t) *happens-before* $(k + 1, t')$

$$C_k \overset{t:\mathsf{fork}(t')}{\rightarrow} C_{k+1} \Rightarrow (k, t) \overset{\mathrm{hb}_E}{\rightarrow} (k + 1, t')$$

The Main Lemma states that once an exception occurs in a failbox, no code executes in that failbox "afterwards".

Lemma 1 (Main Lemma). *Consider an execution E of a program π of the extended language, and consider two thread execution points (k_1, t_1) and (k_2, t_2) in E, such that (k_1, t_1) happens-before (k_2, t_2). If t_1 is executing in some failbox f_1 in state k_1, and t_2 is executing in some descendant f_2 of f_1 in state k_2, then if t_1 is failing in state k_1, then t_2 is failing in state k_2.*

$$\begin{gathered}\mathsf{exec}(\pi, E) \Rightarrow (k_1, t_1) \overset{\mathrm{hb}_E}{\rightarrow} (k_2, t_2) \Rightarrow \\ C_{k_1} = (L_1, \Sigma_1, \Phi_1, T_1) \Rightarrow T_1(t_1) = (f_1, V_1, \mathbf{throw};, \bar{b}_1, \overline{F}_1) \Rightarrow \\ C_{k_2} = (L_2, \Sigma_2, \Phi_2, T_2) \Rightarrow T_2(t_2) = (f_2, V_2, \bar{s}_2, \bar{b}_2, \overline{F}_2) \Rightarrow \\ f_2 \in (\Sigma_2^{-1})^*(f_1) \Rightarrow \bar{s}_2 = \mathbf{throw};\end{gathered}$$

Proof. It suffices to prove for every prefix of some path from (k_1, t_1) to (k_2, t_2) in the happens-before graph, that at the thread execution point (k_3, t_3) at the end of the prefix, one or more of the following hold:

- the thread is failing and the current failbox is f_1
- failbox f_1 and its descendants have been marked as failed and one or more of the following hold:
  - the current failbox is not f_1 or a descendant of f_1, or
  - the thread is failing.

This can be proved easily by induction on the length of the prefix and case analysis on the last edge.

Now consider an execution E of a program π and a *dependency relation* D on the thread execution points of E. We say E *uses failboxes correctly* with respect to D, if whenever thread execution point p_2 depends on thread execution point p_1, the current failbox at p_2 is a descendant of the current failbox at p_1. We say E is *dependency-safe* with respect to D if whenever p_2 depends on p_1, and p_1 happens-before p_2, and p_1 is failing, then p_2 is failing. We then have the Soundness Theorem: if E uses failboxes correctly with respect to D, then E is dependency-safe with respect to D. This follows directly from the Main Lemma.

A machine-checked proof of these properties is available online [12].

4.6 Multithreaded failboxes: Ensuring useful progress

Even if a computation is safe, it might not be contributing to the useful work of the application. Specifically, if multiple computations are running in the same failbox, then this is taken to mean that they depend on each other for useful progress. As a result, if one of them fails, there is no point for the others to continue, so they should be stopped to free up CPU cycles, memory, and other resources these computations may be using. Therefore, in our approach, at the time a failbox f is marked as failed, a *stop f* signal is sent to all threads currently running in f or a descendant of f. When the signal arrives, this results in a *FailboxException* being thrown in the target thread, provided it is still running in f or a descendant. To allow efficient implementations, we do not impose timing constraints on the delivery of the signal, except that it must arrive eventually.

We call this mechanism the Fail Fast mechanism (after the Fail Fast principle [20]).

The usefulness of the Fail Fast mechanism is illustrated by the example in Figure 4. Once failbox f has failed, all subsequent attempts to access the database fail. Assuming most commands access the database, this means the program's functionality is severely degraded. Therefore, it seems appropriate to escalate the failure and terminate the program. This typically signals a system administrator or service management daemon to restart the program in a clean state, hopefully restoring full service. In the example, this behavior is achieved by running not just the *addEntry* calls, but the main loop as well, in failbox f. When an *addEntry* call fails, an asynchronous exception is thrown in the main thread, which causes the loop to terminate.

In fact, since the existing command processing threads are unlikely to be able to run to completion successfully, it makes sense to terminate these as well. This is achieved in the example by running the command processing threads in failbox f as well, by using method *startInCurrentFailbox* (added by our language extension) instead of *start* to start these threads. (To ensure backward compatibility, method *start* starts the new thread in a newly created root failbox, so that failure of the new thread does not cause a stop signal to be sent to the original thread.)

In the example, the failbox hierarchy is as follows. Failbox f, a root failbox, has one child for each try block execution. This ensures, as before, that exceptions in method *compute* do not cause the program to terminate.

The step rules for thread creation are shown in Figure 6. In the formal language, statement **fork** corresponds with method *startInCurrentFailbox*, and **fork**$*$ corresponds with method *start*.

$$\textsc{Fork}\quad \frac{(t, (f, V, \textbf{fork}\ \{\ \overline{s}'\ \}\ \overline{s}, \overline{b}, \overline{F})) \in T \qquad t' \notin \mathrm{dom}(T) \qquad T' = T(t := (f, V, \overline{s}, \overline{b}, \overline{F}), t' := (f, V, \overline{s}', \epsilon, \epsilon))}{(L, \Sigma, \Phi, T) \overset{t:\mathsf{fork}(t')}{\rightarrow} (L, \Sigma, \Phi', T')}$$

$$\textsc{Thread-Complete-Abrupt}\quad \frac{(t, (f, V, \textbf{throw}; , \epsilon, \epsilon)) \in T \qquad f \notin \Phi \qquad \Phi' = \Phi \cup (\Sigma^{-1})^*(f)}{(L, \Sigma, \Phi, T) \rightarrow (L, \Sigma, \Phi', T)}$$

$$\textbf{fork}* \ \{\ \overline{s}\ \} \equiv\ \textbf{fork}\ \{\ x := \textbf{newfb};\ \textbf{reenter}\ (x)\ \{\ \overline{s}\ \}\ \} \quad \text{where } x \text{ is fresh}$$

Fig. 6. Step rules for thread creation

4.7 Wait dependency safety

A sub-concern of the concern of ensuring useful progress is the concern of ensuring progress. Specifically, one of the correctness properties that are difficult to achieve in the presence of unchecked exceptions is *wait dependency safety*, the property that if, in a given program execution, a wait operation W depends on a computation A, then, assuming that W terminates if A does not fail, W terminates. Analogously to the dependency relation used in the definition of dependency safety, the wait dependency relation used here is an application-specific relation; the intention is that if a wait operation W depends on a computation A, this means that, abstractly speaking, W waits for a signal to be sent by A. In Java, a typical example of this is when W is an *Object.wait* call on some object o and A at some point performs an *Object.notifyAll* call on o.

Failboxes can be used to achieve wait dependency safety. We say that a program *uses failboxes correctly* for the purpose of wait dependency safety if whenever in a given program execution, a wait operation W depends on a computation A, then A runs in a failbox f and W runs in a descendant of f. We then have the property that if a program uses failboxes correctly for the purpose of wait dependency safety, then the program is wait-dependency-safe. Indeed, if A fails, a stop signal is sent to the thread that is running W. As a result, when the signal arrives, either W has already terminated, or W is terminated by the *FailboxException* thrown by the Fail Fast mechanism. We call this property the soundness of the Fail Fast mechanism.

A machine-checked proof of this property is available online [12].

5 Cancellation and Compensation

We propose the use of failboxes in programs to make them safe for failures. However, it turns out that if failboxes are applied correctly in a program, then this also enables safe *cancellation* of computations, with no extra effort, and without the need for polling, through the Fail Fast mechanism. In order to enable cancellation of a computation, the program runs it in a dedicated failbox; to cancel the computation, it calls the *Failbox* object's *cancel* method, which simulates the occurrence of a failure in the failbox and triggers the Fail Fast mechanism. This achieves the convenience of the deprecated *Thread.stop* approach, without the safety risk.

Consider for example the program of Figure 4. The main loop repeatedly receives a command and starts a command thread to process it. The processing is done inside a try-catch statement, and therefore in a per-command child failbox of the root failbox. This program could be extended to enable cancellation of commands as follows. In order to cancel a command, the program calls the command failbox's *cancel* method. If the command thread is executing in the command failbox, it is stopped; however, if it is executing inside the database, it is allowed to continue to execute until it leaves the root failbox and re-enters the command failbox, at which point an exception is thrown. Contrast this with

calling *stop* on the command thread, which would stop the thread even if it was running in the database, causing the entire program to fail.

The failboxes mechanism also enables *safe compensation*. By compensation, we refer to the scenario where a client computation invokes a service offered by a provider computation, which changes the provider's state. This imposes the obligation on the client to invoke a compensating service to restore the provider's state, after the client is done using the service. The conventional approach to compensation is through try-finally statements. However, an unchecked exception can cause the compensation action to be skipped, if the exception occurs after the action that is to be compensated, but before the try block is entered, or if it occurs after the finally block is entered, but before the compensation action completes.

This may be addressed using the failboxes mechanism by performing the following transformation:

```
init();                          ⇒    enter (provider) {
try {                                   init();
  // Use the service                    reenter (client) {
} finally {                               // Use the service
  compensate();                         }
}                                       compensate();
                                 }
```

Before invoking the service, the thread running the client computation enters the provider's failbox. After the service is invoked, it re-enters the client failbox using a nested reenter statement where the client uses the service. When the client is done using the service, it leaves the nested enter statement, causing the thread to re-enter the provider failbox, perform the compensating action, and finally leave the outer enter statement, re-entering the client failbox. This approach guarantees that either the compensation occurs or the provider failbox is marked as failed. If an exception occurs while the client uses the service, the client failbox is marked as failed, but the exception is not propagated by the reenter statement. This ensures that compensation is not skipped. When the thread leaves the outer enter statement, it enters the client failbox, which was marked as failed, and therefore the exception is propagated from that point, as in the case of the try-finally statement.

6 Proof rules

To show that it is easy to reason about programs that use failboxes, in this section we propose separation logic proof rules for the main envisaged usage patterns.

Recall the semantics of separation logic assertions: **emp** describes the empty heap, and the separate conjunction $P * Q$ describes a heap that can be split into one that satisfies P and one that satisfies Q:

$$s, h \vDash \mathbf{emp} \Leftrightarrow h = \emptyset \quad s, h \vDash P * Q \Leftrightarrow \exists h_1, h_2 \bullet h = h_1 \uplus h_2 \wedge s, h_1 \vDash P \wedge s, h_2 \vDash Q$$

We extend the syntax of correctness judgments (but not the syntax of assertions) to be failboxes-aware. Specifically,

$$\Sigma; f \vdash \{P\}\ \overline{s}\ \{Q\}$$

denotes the correctness of statement list $\overline{s}$ under commitment list Σ, current failbox f, precondition P, and postcondition Q. The syntax of commitment lists is as follows:

$$\Sigma ::= \epsilon \mid \Sigma, f : P$$

We say that assertion P is committed to failbox f. Informally, this means that to access the resources of P, f must first be entered. Failboxes are denoted using logical variables.

The above correctness judgment implies the following validity statement:

$$[\![\Sigma]\!] * P \Rightarrow \mathsf{valid}(\overline{s}, [\![\Sigma]\!] * Q, [\![\Sigma]\!] * \mathbf{true})$$

(under the assumption that $\overline{s}$ does not assign to any variables that Σ depends on) where Σ is here interpreted as a separation logic assertion as follows:

$$[\![\epsilon]\!] \equiv \mathbf{emp} \qquad [\![\Sigma, f : P]\!] \equiv [\![\Sigma]\!] * (f \in \Phi \vee P)$$

i.e., for each commitment $f : P$, either P holds (and is owned by the current thread) or f has failed. $\mathsf{valid}(\overline{s}, Q, R)$ is true under a given heap, failed set, and variable environment, if after executing $\overline{s}$ in this state, upon normal completion Q holds and upon abrupt completion R holds.

A throw statement always satisfies partial correctness.

C-Throw
$$\Sigma; f \vdash \{P\}\ \mathbf{throw};\ \{Q\}$$

For verifying a try-catch statement, the heap is split into two parts: part P_f is accessed by the try block only inside **enter** (f) statements, and part P is accessed freely. The second premise of the rule ensures soundness for normal completion of the try block. The third is for the case where the try block fails.

C-TryCatch
$$\frac{\forall f' \bullet \Sigma, f : P_\mathrm{f}; f' \vdash \{P\}\ \overline{s}\ \{Q\} \qquad P_\mathrm{f} * Q \Rightarrow Q' \qquad \Sigma; f \vdash \{P_\mathrm{f}\}\ \overline{s}'\ \{Q'\}}{\Sigma; f \vdash \{P_\mathrm{f} * P\}\ \mathbf{try}\ \{\ \overline{s}\ \}\ \mathbf{catch}\ \{\ \overline{s}'\ \}\ \{Q'\}}$$

(under the assumption that P_f does not depend on any variables that $\overline{s}$ assigns to).

An enter block can access the piece of heap associated with the failbox being entered.

C-Enter
$$\frac{\Sigma; f \vdash \{P * P_\mathrm{f}\}\ \overline{s}\ \{Q * P_\mathrm{f}\}}{\Sigma, f : P_\mathrm{f}; f' \vdash \{P \wedge x = f\}\ \mathbf{enter}\ (x)\ \overline{s}\ \{Q\}}$$

The compensation pattern can be verified as follows.

$$\textsc{C-Compensation}\quad \frac{\Sigma; f \vdash \{P * P_\mathrm{f}\}\ \overline{s}_1\ \{Q_1 * P'_\mathrm{f} \wedge y = f'\} \qquad \Sigma, f : P'_\mathrm{f}; f' \vdash \{Q_1\}\ \overline{s}_2\ \{Q\} \qquad \Sigma; f \vdash \{P'_\mathrm{f}\}\ \overline{s}_3\ \{P_\mathrm{f}\}}{\Sigma, f : P_\mathrm{f}; f' \vdash \{P \wedge x = f\}\ \textbf{enter}\ (x)\ \{\ \overline{s}_1\ \textbf{reenter}\ (y)\ \{\ \overline{s}_2\ \}\ \overline{s}_3\ \}\ \{Q\}}$$

(under the assumption that P'_f does not care about any variables that $\overline{s}_2$ assigns to). The compensation pattern allows the commitment $f : P_\mathrm{f}$ to be replaced temporarily with the commitment $f : P'_\mathrm{f}$.

A machine-checked soundness proof of these proof rules is available online [12].

We developed a prototype verifier based on these ideas [12].

7 Implementation Issues

We created a prototype implementation of the approach on the .NET Framework as a C# 3.0 library. C# 3.0's lambda expression syntax can be used to write reasonably concise enter statements.

A major complication for achieving a fully correct implementation of the approach in the form of a library, is the fact that the .NET Framework Common Language Runtime may throw an exception at any program point, due to an internal resource limit being reached or an internal error being discovered within the execution engine [21]. (The same holds for the Java Virtual Machine. See the Java Virtual Machine Specification, Second Edition [14], Section 2.16.2.) Specifically, if an enter block completes abruptly with an exception, no internal exception must intervene between catching the exception and marking the failbox as failed; otherwise, the enter statement completes without marking the failbox as failed, breaking dependency safety.

Version 2.0 of the .NET Framework introduced constructs specifically for writing code that must execute reliably in the presence of internal exceptions [21]. We used these constructs in our prototype implementation to ensure that on abrupt completion of the body of an enter statement, the failbox and its descendants are marked as failed and stop signals are sent to other threads executing in the failbox or its descendants. Specifically, we used the following API:

$$\mathit{ExecuteCodeWithGuaranteedCleanup}(t, c, u)$$

where t and c are delegates (similar to function pointers in C) and u is arbitrary user data that is passed to t and c. The API first executes t. When t completes, either normally or abruptly, the cleanup delegate c is executed. The API guarantees that no internal exceptions occur during the execution of c, provided that c satisfies certain constraints, such as: no heap memory allocation, and no unbounded call stack memory allocation. Unfortunately, these constraints have not been spelled out very precisely anywhere; we had to make some assumptions as to what can reasonably be executed without the risk of internal exceptions.

We have performed a few microbenchmark performance tests. These indicate the following approximate timings for the following statements:

Statement	**Timing**	**Timing***
try {} **catch** {}	13μs	1.9μs
try { **enter** (f) {} } **catch** {}	23μs	3.4μs

To measure the impact of the *ExecuteCodeWithGuaranteedCleanup* construct, we replaced it with a dummy that uses a simple try-finally statement. The resulting timings are shown in the third column. It turns out that the overhead of this construct dominates the run time.

Even though the current performance is probably acceptable for most real-world applications, we believe it can still be improved significantly, in particular if the constructs are implemented directly in the virtual machine rather than as a library. Performing such an implementation is future work.

We have also prepared a prototype implementation of failboxes as a library on the Java virtual machine. However, due to the absence of constructs to prevent internal or asynchronous exceptions on this platform, the implementation is not safe in the presence of such exceptions.

The prototype implementations are available on line [12].

8 Related Work

To the best of our knowledge, failboxes are the first approach for programmers to achieve dependency safety of their Java-like programs that combines low programming overhead, low performance overhead, and low reasoning overhead, and is compositional (i.e. failboxes can be nested arbitrarily).

Languages as operating systems Many extensions of Java have been proposed that support running multiple programs or *tasks* in the same virtual machine. These can typically be used to enforce dependency safety. However, in contrast to failboxes, all of these have goals beyond dependency safety, typically including protection against malicious code, and accounting of memory and other resources. As a result, they impose greater programming and performance overhead on communication between tasks than the overhead of switching between failboxes.

Perhaps the most closely related such system is Luna [11]. To support memory accounting and immediate guaranteed memory reclamation when a task is killed, the heap is logically partitioned among the tasks; the only way for one task to access an object belonging to another task is through a *remote pointer*, which is distinguished from local pointers through its type. When a task is killed, remote pointers pointing into it are *revoked*, so that if the task was holding a lock, other tasks do not see inconsistent state. Failboxes offer no memory accounting or guaranteed memory reclamation, but in turn impose a lower programming and performance overhead. Specifically, passing data across tasks requires either copying or the use of remote pointers, both of which incur a programming and

performance overhead; failboxes, in contrast, allow data to be passed around freely.

DrScheme [8, 7] is a Scheme environment designed for programs that serve as platforms for other programs. In DrScheme, it is possible for two child programs to share a mutable data structure and yet be killed independently. The solution is to host the data structure in a separate thread, and to access it only via message passing with this thread. DrScheme's contribution is that it enables two untrusted child programs to set up such a shared data structure without circumventing resource policies and without the need for the shared structure to be trusted by the kernel. However, from a dependency safety point of view, the situation is as in Java: DrScheme requires either the use of message passing between separate threads or manually guarding dependent code.

Erlang [1] is a language focused on reliability. Inconsistent data structures within a process are ruled out because the language has no destructive update. Processes communicate through asynchronous message passing. Fail-fast is achieved by linking processes: when a process dies, an exit signal is sent to linked processes, causing those to die as well by default.

Non-compositional approaches Marlow et al. [16] propose an extension of concurrent Haskell with constructs that make it possible to write safe programs where one thread throws an asynchronous exception in another thread. The block e construct disables asynchronous exceptions during execution of e; e can use unblock e' to re-enable them during execution of a sub-expression e'. Unlike failboxes, the block construct is not compositional; for example, in the program of Figure 4, the *addEntry* call could be protected against cancellation of jobs using block; however, imagine the command processing program is part of a larger system. Then one may want to cancel the program as a whole, including any *addEntry* calls. This is possible with failboxes (by cancelling failbox f, which cancels its descendants as well), but not with the block construct. Also, the construct does not help in dealing with failures; for example, a failure during the *addEntry* call would not prevent further accesses to the database. However, the block construct, or something similar, is useful and even necessary to be able to robustly implement failboxes as a library in a given language.

Starting with version 2, the .NET Framework includes reliability features that make it possible to write cleanup routines that are guaranteed to execute even in the presence of failure or cancellation [21]. However, like the block construct, the approach is not compositional: these cleanup routines cannot be cancelled; furthermore, they must be carefully coded to rule out failures within the cleanup routines themselves since those are not dealt with safely. The mechanism is intended only for manipulation of execution environment resources; it is not for general application use.

Three further reliability-related features in .NET Framework version 2 are the following. Firstly, cancellation is disabled during finally blocks. This enables safe cleanup in the presence of cancellation (but not failure). Secondly, an unhandled exception in one thread kills all other threads, without executing catch or finally blocks. However, in the thread that throws the unhandled ex-

ception, finally blocks are executed normally and locks are released, leaving a time window between the release of the lock and the time the exception reaches the toplevel (possibly after executing other finally blocks) where other threads can see inconsistent state. Thirdly, a method *Environment.FailFast* was added, which terminates the program immediately.

Rudys et al. [18] propose weaving code into an untrusted plugin (such as an applet) that polls a cancellation request flag to enable forcibly cancelling the plugin. The flag is also checked whenever the host system calls into the plugin. In our approach, a thread running in one failbox may protect itself from cancellation of its failbox by entering an ancestor failbox to which it has a reference; however, separate techniques (e.g., perhaps by associating permissions with failboxes) could be used to prevent this in case the thread is running untrusted code.

The SCOOP multithreading approach for Eiffel [17] has a notion of *subsystems*. A subsystem in SCOOP is a thread and a set of objects handled by that thread. Brooke and Paige [3] suggest marking an object as "dead" when the processing of an asynchronous incoming call fails, causing subsequent calls to fail immediately. SCOOP subsystems cannot be nested.

Other related work Garcia et al. [9] provide a survey of exception mechanisms. However, the authors do not discuss the dependency safety issue. In fact, most modern imperative and/or object-oriented languages have inherited the exception mechanism of CLU [15] and therefore suffer from the problems addressed by our approach.

Class-handlers, as proposed by Dony [4] and others, are exception handlers associated with classes rather than blocks of statements; they apply to all methods of the class. They would facilitate manually guarding dependent code. For example, a class-handler on the *Database* class could set a *failed* field to *true* when an unchecked exception is caught and then re-throw the exception. The field would still need to be checked manually on entry to each method.

Weimer and Necula [22] propose *compensation stacks* to make it easier to write effective cleanup code. However, they do not address the safety issues identified in Section 5.

Fetzer et al. [5] assume the viewpoint that "exception handling is only effective if the premature termination of a method due to an exception does not leave an object in an inconsistent state". The paper proposes techniques to detect and "mask" *non-atomic exception handling*, i.e. violations against *failure atomicity*. The paper assumes that after catching an exception, the entire application should be in a consistent state, whereas we allow *failed failboxes* to remain in an inconsistent state, while preventing control from entering a failed failbox. The authors find a large number of Java methods that are not failure atomic. This would strengthen the case for failboxes, because it indicates that exceptions do indeed commonly leave objects in an inconsistent state.

An alternative way to deal with failures is to roll the state of the objects involved back to a consistent state, through the use of transactions (e.g. Shavit and Touitou [19], Welc et al. [23], Fetzer et al. [5]). However, this has a greater performance overhead; also, it presents problems when the computation that failed

performed I/O. Our failboxes approach is more conservative from a semantic and performance point of view.

This work was inspired by our research in program verification for Java-like languages that is sound in the presence of failures. To the best of our knowledge, no existing program verifiers for Java-like languages (including ESC/Java [6] and Spec# [2]) have this property. In Jacobs et al. [13], we propose a verification approach for Java programs where the programmer manually guards dependent code using flag variables that track an object's consistency. The present work addresses the programming overhead of that approach.

9 Conclusion

We propose a language extension, called *failboxes*, that facilitates writing sequential or multithreaded programs that provably preserve intended safety properties and that do not leak resources, even in the presence of failure, and that perform safe cancellation of computations. To the best of our knowledge, it is the first such extension of a Java-like language that combines low programming, performance, and reasoning overhead, and that is compositional.

Future work includes gaining experience with our prototype implementation, mainly to assess the applicability and the usability of the approach. We anticipate the possible need to facilitate the placement of enter blocks, perhaps through annotations on methods, classes, or packages, or through some inference scheme. Other work includes applying the failboxes idea to the problem of exception handling in asynchronous and callback patterns.

Acknowledgements

The authors would like to thank Jan Smans, Marko van Dooren, Scott Owens, and the Cambridge programming languages group for their helpful comments. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy.

Bibliography

[1] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. FMCO 2005*, volume 4111 of *LNCS*. Springer, 2006.

[3] Phillip J. Brooke and Richard F. Paige. Exceptions in Concurrent Eiffel. *Journal of Object Technology*, 6(10):111–126, November 2007.

[4] Christophe Dony. Exception handling and object-oriented programming: a synthesis. In *Proc. OOPSLA*, 1990.

[5] Christof Fetzer, Karin Högstedt, and Pascal Felber. Automatic detection and masking of non-atomic exception handling. In *Proc. Intl. Conf. Dependable Systems and Networks (DSN)*, 2003.
[6] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245, 2002.
[7] Matthew Flatt and Robert Bruce Findler. Kill-safe synchronization abstractions. In *Proc. PLDI*, 2004.
[8] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems (or Revenge of the son of the Lisp machine). In *Proc. Intl. Conf. on Functional Programming (ICFP)*, 1999.
[9] Alessandro F. Garcia, Cecília M. F. Rubira, Alexander B. Romanovsky, and Jie Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59 (2):197–222, 2001.
[10] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Prentice Hall PTR, 2005.
[11] Chris Hawblitzel and Thorsten von Eicken. Luna: a flexible Java protection system. In *Proc. OSDI*, 2002.
[12] Bart Jacobs and Frank Piessens. Failboxes: Prototype implementations, prototype verifier, machine-checked metatheory. http://www.cs.kuleuven.be/~bartj/failboxes, July 2008.
[13] Bart Jacobs, Peter Müller, and Frank Piessens. Sound reasoning about unchecked exceptions. In *Proc. ICFEM*, 2007.
[14] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999. Online at http://java.sun.com/docs/books/jvms/.
[15] Barbara Liskov and Alan Snyder. Exception handling in CLU. *IEEE Trans. Software Eng.*, 5(6):546–558, 1979.
[16] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. In *Proc. PLDI*, 2001.
[17] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
[18] Algis Rudys, John Clements, and Dan S. Wallach. Termination in language-based systems. In *Network and Distributed System Security Symposium (NDSS)*, February 2001.
[19] N. Shavit and D. Touitou. Software transactional memory. In *Proc. PODC*, pages 204–213, 1995.
[20] Jim Shore. Fail fast. *IEEE Software*, September 2004.
[21] Stephen Toub. Keep your code running with the reliability features of the .NET Framework. *MSDN Magazine*, October 2005.
[22] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *Proc. OOPSLA*, pages 419–431, October 2004.
[23] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *Proc. ECOOP*, 2004.

Verification of Unloadable C Modules—Status Report

Bart Jacobs *Jan Smans* *Frank Piessens*

Report CW 567, October 2009

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Verification of Unloadable C Modules—Status Report

Bart Jacobs *Jan Smans* *Frank Piessens*

Report CW 567, October 2009

Department of Computer Science, K.U.Leuven

Abstract

C programs may dynamically load and unload modules. For example, some operating system kernels support dynamic loading and unloading of device drivers. This causes specific difficulties in the verification of such programs and modules; in particular, it must be verified that no functions or global variables from the module are used after the module is unloaded.

We propose a separation-logic-based approach for the verification of such programs and modules. We propose proof rules for loading and unloading modules, and for dealing with pointers to functions in unloadable modules, that ensure soundness while imposing minimal verification overhead. We offer a formalization and we report on verifying a small kernel-like program using a prototype implementation of the approach in our verifier, VeriFast. To the best of our knowledge, ours is the first approach for sound modular verification of unloadable modules.

Verification of Unloadable C Modules Status Report

Bart Jacobs*, Jan Smans, and Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{bart.jacobs,jan.smans,frank.piessens}@cs.kuleuven.be

Abstract. C programs may dynamically load and unload modules. For example, some operating system kernels support dynamic loading and unloading of device drivers. This causes specific difficulties in the verification of such programs and modules; in particular, it must be verified that no functions or global variables from the module are used after the module is unloaded.
We propose a separation-logic-based approach for the verification of such programs and modules. We propose proof rules for loading and unloading modules, and for dealing with pointers to functions in unloadable modules, that ensure soundness while imposing minimal verification overhead. We offer a formalization and we report on verifying a small kernel-like program using a prototype implementation of the approach in our verifier, VeriFast. To the best of our knowledge, ours is the first approach for sound modular verification of unloadable modules.

1 Introduction

In statically typed safe programming languages (including Java, C#, the ML family, and Haskell), code is immutable and permanent.[1] That is, both statically bound and dynamically bound routine calls always succeed and are bound to code that satisfies the static type of the call. Also, if an object reference or function value satisfies a given contract at one point in time, it continues to do so forever.

This is not the case in dynamically typed languages like LISP, Scheme, JavaScript, Ruby, or Python, and in unsafe languages like C and C++. In C, if at one point during execution a function pointer points to a function that satisfies a given contract, this does not mean it always will. The module (DLL, shared object, ...) containing the function may be unloaded, or the function's code may reside on the stack or in a malloc'ed piece of memory.

Existing verification approaches for C programs (VCC [4], Frama-C [1], HAVOC [5], Smallfoot [2], our own verifier VeriFast [7], Jahob [9]) assume that the program is unchanging and is not part of the mutable state. As a result, these

* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).
[1] Some of these languages may garbage collect code that is unreachable, but this is unobservable.

approaches cannot be used for sound verification of programs that involve the unloading of code.

In this paper, we propose a separation-logic-based approach for extending a verification approach for C programs to enable verification of programs involving code unloading. The approach is simple: execution of unloadable code at a given address requires a permission to read at the address and a proof that the code at this address has the expected behavior. Specifically, execution of the body of a function in an unloadable module requires an abstract permission that states that the module is currently loaded.

The conventional approach for verifying dynamically bound calls in separation logic is through predicate families [8], i.e. abstract predicates indexed by the target function. VeriFast, for example, has used predicate families indexed by function pointer to verify function pointer calls. However, this is not applicable to unloadable code since the function pointer no longer immutably identifies a specific function. Therefore, in our approach, we drop the use of predicate families and instead we use parameterized function types combined with higher-order predicates.

We implemented the approach in our prototype verifier, VeriFast, and we verified a small server that allows clients to load modules, unload modules, and use services provided by the modules, mimicking operating system kernels that may dynamically load and unload device drivers. The implementation and the example are online at http://www.cs.kuleuven.be/~bartj/verifast/.

The rest of the paper is structured as follows. First, in Section 2, we show how programs involving dynamic code loading, but not unloading, may be verified in VeriFast. Then, in Section 3, we present the extensions required to enable verification of code unloading. In Section 4 we further extend the example to get a fully abstract treatment. We provide a formal treatment in Section 5. We discuss the implementation in Section 6 and related work in Section 8. We conclude in Section 9.

2 Dynamically loaded code

The C program in Figure 1 illustrates dynamic code loading. It is safe, and VeriFast can confirm this thanks to the annotations shown on a gray background.

Execution proceeds as follows. The main program, a.c, dynamically loads the shared object b.so, generated from b.c, and looks up the getIncr function in that object. If the function is not found, the program aborts. Otherwise, it calls the function to retrieve another function, which it then calls as well.

The soundness of the verification approach used in this example, is based on the assumption that the mapping from function names to function types (with their associated contract), as specified in b.spec in the example, is globally unique. This seems realistic, especially if sufficiently long and distinctive names are chosen, and if the origin of the code is trusted, e.g. through code signing. An alternative approach would be to submit the module to a verification tool at load time.

```
// dlfcn.h
void *dlopen(char *name);
  requires string(name);
  ensures string(name) * lib(result);

// b.h
typedef int incrType(int x);
  requires true;
  ensures result - 1 = x;

typedef incrType *getIncrType();
  requires true;
  ensures is_incrType(result);

// a.c
#include "dlfcn.h"
#include "b.h"
#include "b_proxy.h"

void main()
  requires true;
  ensures true;
{
  void *l = dlopen("b.so");
  getIncrType *f = dlsym_getIncr(l);
  if (f = 0) abort();
  incrType *g = f();
  int y = g(41);
  assert(y = 42);
}
```

```
// b.spec
getIncr : b.h#getIncrType

// b_proxy.h  (generated from b.spec)
#include "b.h"

getIncrType *dlsym_getIncr(void *l);
  requires lib(l);
  ensures lib(l) ∧ (result = 0 ? true :
    is_getIncrType(result));

// b_proxy.c  (generated from b.spec)
getIncrType *dlsym_getIncr(void *l)
{ return dlsym(l,"getIncr"); }

// b.c
#include "b.h"

int myIncr(int x) : incrType
{ return x + 1; }

incrType *getIncr()
  : getIncrType
{ return myIncr; }
```

Fig. 1. An example of dynamic loading, but not unloading, in C. VeriFast annotations are shown on a gray background

3 Code unloading

Suppose we added the following contract to dlfcn.h in Figure 1:

void *dlclose*(**void** $*\ell$);
requires *lib*(ℓ);
ensures true;

Clearly, this would be unsound, since this contract would allow the main program to unload the library and then perform calls through function pointers that point into the library.

To enable unloading, we extend the approach with four ingredients. Firstly, we add *module names* as first-class values inside annotations. For example, the module name of b.c is b and this name may be used as an expression in annotations. Secondly, we add a built-in predicate *module*, that takes a module name and states that this module is currently loaded. Thirdly, a module may declare itself to be **unloadable**. The precondition of each function of an unloadable module m must imply *module*(m), and this permission is unavailable for the duration of the function's execution. Also, each module that exports symbols for dynamic linking must be unloadable.[2] Finally, we allow function types to be parameterized by arbitrary values, including module names.

In Figure 2, we show the example of Figure 1, modified so that the main program now unloads the module after performing the function calls. The contracts have been updated to include the *module* permission.

4 Abstraction

The example of Figure 2 is lacking in abstraction. For example, suppose we want to write a module that logs calls to the *incr* function. This is not possible given the precondition of *incrType* in b.h, which provides access only to the module image.

To enable this implementation, we parameterize *incrType* not by a module name, but by an arbitrary predicate P. Further, in order to allow the logging infrastructure to be torn down after the client is done using the module, the *getIncr* function returns not just a pointer to the *incr* function, but a struct containing both a pointer to the *incr* function and a pointer to a *dispose* function.

The implementation, with annotations, is shown in Figure 3. The implementation now uses global variables. To support these, we extend the verification approach with three new ingredients. Firstly, we add another built-in predicate *code*, which takes a module name and represents the code of the named module. Secondly, the *module* predicate now represents not just the module's code but its global variables as well. That is, it is the separate conjunction of the code and the globals. Thirdly, we introduce a variant of the *module* predicate,

[2] This may be relaxed by specifying in the spec file whether the module is unloadable or not, and adapting the proxy contract accordingly

```
// dlfcn.h
void *dlopen(char *name);
  requires string(name);
  ensures string(name) *
    ∃m • lib(result, m) * module(m);

void dlclose(void *ℓ);
  requires ∃m • lib(ℓ, m) * module(m);
  ensures true;

// b.h
typedef int incrType (m) (int x);
  requires module(m);
  ensures module(m) ∧ result − 1 = x;

typedef incrType *getIncrType (m) ();
  requires module(m);
  ensures module(m) ∧
    is_incrType(result, m);

// a.c
#include "dlfcn.h"
#include "b.h"
#include "b_proxy.h"

void main()
  requires true;
  ensures true;
{
  void *ℓ = dlopen("b.so");
  getIncrType *f = dlsym_getIncr(ℓ);
  if (f = 0) abort();
  incrType *g = f();
  int y = g(41);
  dlclose(ℓ);
  assert(y = 42);
}
```

```
// b.spec
getIncr : b.h#getIncrType

// b_proxy.h (generated from b.spec)
#include "b.h"

getIncrType *dlsym_getIncr(void *ℓ);
  forall m;
  requires lib(ℓ, m);
  ensures lib(ℓ, m) ∧ (result = 0 ∨
    is_getIncrType(result, m));

// b.c
#include "b.h"
unloadable;

int myIncr(int x) : incrType(b)
{ return x + 1; }

incrType *getIncr()
  : getIncrType(b)
{ return myIncr; }
```

Fig. 2. The example of Figure 1, with unloading added, but lacking in abstraction

named $module_0$, which represents the same state, but additionally states that the globals hold their initial value. The contract of *dlopen* is updated to provide $module_0$.

5 Formal System

In this section, we formalize the approach and state the soundness theorem. A detailed soundness proof is future work.

The rest of this section is structured as follows. In Section 5.1, we introduce the formal programming language and execution model, and we illustrate it using an example program and module. In Section 5.2, we define the syntax and the semantics of the specification language, and we show the specification for the example module. Finally, in Section 5.3, we introduce the proof system, we state its soundness theorem, and we show a proof outline for the example module.

5.1 Language Syntax and Semantics

The formal programming language is an extension of the standard separation logic language with function pointer call and module load and unload commands, and with function values L. The latter are used to represent pieces of code in the heap. Its syntax is as follows:

$$
\begin{array}{rl}
 & n \in \mathbb{Z}, x \in \text{Vars}, \tau \in \text{FunTypes} \\
e ::= & n \mid x \mid e + e \mid e - e \\
b ::= & e = e \mid e < e \\
c ::= & x := \mathbf{cons}(\overline{e}) \mid x := [e] \mid [e] := e \mid \mathbf{dispose}(e) \mid x := e \\
 & \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c \mid (c; c) \mid x := \mathbf{call}\ e(\overline{e}) \\
 & \mid x := \mathbf{load}\ e\ \mathbf{as}\ \tau \mid \mathbf{unload}(e) \\
L ::= & \mathbf{lambda}\ (\overline{x})\ c
\end{array}
$$

We adopt the standard run-time state of separation logic, consisting of a store s, a total function that maps program variable names to integers, and a heap h, a partial function that maps positive integer addresses to integer values. In order to be able to store function values in the heap, we assume a one-to-one encoding $\lfloor \cdot \rfloor$ of function values into integers.

To model the loading and unloading of modules, we assume the existence of a module repository *Modules*, which is a finite map from module names to module definitions. A module definition consists of a module contract and a module image. We assume each module has a single entry point; therefore, the module contract is simply the function contract for the single entry point. We assume a set *FunTypes* of function type names, and a mapping from function type names to contracts. The run-time semantics is concerned only with the names and not the meanings of the function types. It uses them to perform a run-time type check. The module image is simply a tuple of one or more integers. The first element of the tuple is the encoded code for the module's entry point; the other

```
// b.h
typedef int incrType (P) (int x);
  requires P;
  ensures P ∧ result − 1 = x;

typedef void disposeType (o, P, m) ();
  requires obj(o, _, _) ∗ P;
  ensures module(m);

struct obj {
  incrType ∗incr;
  disposeType ∗dispose;
};

predicate obj(o, d, p) =
  o→incr ↦ p ∗ o→dispose ↦ d;

typedef struct obj ∗
  getIncrType (m) ();
  requires module0(m);
  ensures
    ∃P, d, p • obj(result, P, m, d, p) ∗ P
    ∧ is_incrType(p, P)
    ∧ is_disposeType(d, result, P, m);

// a.c
#include "dlfcn.h"
#include "b.h"
#include "b_proxy.h"

void main()
  requires true; ensures true;
{
  void ∗ℓ = dlopen("b.so");
  getIncrType ∗f =
    dlsym_getIncr(ℓ);
  if (f = 0) abort();
  struct obj ∗o = f();
  int y = o→incr(41); assert(y = 42);
  o→dispose(); dlclose(ℓ)
}
```

```
// dlfcn.h
void ∗dlopen(char ∗name);
  requires string(name);
  ensures string(name) ∗
    ∃m • lib(result, m) ∗ module0(m);
void dlclose(void ∗ℓ);
  requires ∃m • lib(ℓ, m) ∗ module(m);
  ensures true;

// b.spec
getIncr : b.h#getIncrType

// b_proxy.h (generated from b.spec)
#include "b.h"

getIncrType ∗dlsym_getIncr(void ∗ℓ);
  forall m;
  requires lib(ℓ, m);
  ensures lib(ℓ, m) ∧ (result = 0 ∨
    is_getIncrType(result, m));

// b.c
#include "b.h"
#include "logging.h"
unloadable;

struct logchannel ∗chan;

predicate Q = code(b) ∗
  ∃c • chan ↦ c ∗ logchannel(c);

int myIncr(int x) : incrType(Q)
{ logCall(chan); return x + 1; }

struct obj o = {myIncr, myDispose};

void myDispose() : disposeType(o, Q, b)
{ disposeLogChannel(chan); }

struct obj ∗getIncr()
  : getIncrType(b)
{ chan = allocLogChannel(); return o; }
```

Fig. 3. The example of Figure 2, with better abstraction

elements may be code or data. We assume a one-to-one encoding $\lfloor \cdot \rfloor$ of module names to integers.

The dynamic semantics is given by the step rules in Figure 4. A run-time configuration consists of a state (i.e. a store and a heap) and a continuation. This is either a done continuation, denoting the successful termination of the program, a return continuation, denoting a caller stack frame, or a command continuation:

$$\kappa ::= \mathbf{done} \mid \mathbf{ret}(x, s, \kappa) \mid c; \kappa$$

A return continuation specifies the variable x that will receive the return value, and the saved store.

Most rules are standard. A function pointer call $x := \mathbf{call}\ e(\overline{e})$ looks for an encoded function value $\mathbf{lambda}\ (\overline{x})\ c$ at the address given by e. It gets stuck if the address is not allocated, if the value at the address does not encode a function value, or if the length of the argument list does not match the length of the parameter list. Otherwise, it pushes a return continuation and executes the body of the function value under a store that binds the parameters to the corresponding arguments and the variable ip (for *instruction pointer*) to the address given by e.

When the body of a function value completes, the caller's store is restored, and the return value of the function value, written by convention into variable result, is assigned to the target variable of the call.

A command $x := \mathbf{load}\ e\ \mathbf{as}\ \tau$, where τ is a function type name, fails if there is no module whose name is encoded by the value of e, or if there is one but its contract is not τ. In that case, the command returns zero. Otherwise, it allocates $m + 1$ consecutive memory locations, where m is the size of the module image. It writes the size itself into the first location, and the module image into the subsequent locations. It returns the address of the first location. As a result, the module's entry point is at the returned address plus one.

Command $\mathbf{unload}(e)$ gets stuck if the memory location at the address given by e is not allocated, holds a negative value, or holds a value m such that some of the m next memory locations are not allocated. Otherwise, it deallocates these $m + 1$ memory locations.

Figure 5 shows an example main program and module repository holding a single module, called myIncrLib. The main program starts by loading the module whose name is given by the initial value of variable *libName*, which is assumed to be provided by the user; i.e., it is arbitrary. It then calls the loaded module's entry point, which it assumes returns a pointer to a struct that has two fields, the first of which points to a function that returns its argument incremented by one, and the second one serves to clean up any resources used by the module. It first calls the incrementor function, and asserts that it indeed incremented its argument. (It performs a null pointer dereference if it did not.) It then calls the dispose function, and finally unloads the module.

The example module's image consists of six values, the first three of which are encoded function values, and the latter will serve as global variables. To illustrate that our approach performs strong abstraction and allows the module

Cons

$$\frac{0 < \ell \qquad \mathrm{dom}(h) \cap \{\ell, \ldots, \ell + m - 1\} = \emptyset}{\langle s, h, x := \mathbf{cons}(e_1, \ldots, e_m); \kappa\rangle \rightsquigarrow \langle s[x := \ell], h[\ell := [\![e_1]\!]_s, \ldots, \ell + m - 1 := [\![e_m]\!]_s], \kappa\rangle}$$

Lookup

$$\frac{([\![e]\!]_s, v) \in h}{\langle s, h, x := [e]; \kappa\rangle \rightsquigarrow \langle s[x := v], h, \kappa\rangle}$$

Mutate

$$\frac{[\![e]\!]_s \in \mathrm{dom}(h)}{\langle s, h, [e] := e'; \kappa\rangle \rightsquigarrow \langle s, h[[\![e]\!]_s := [\![e']\!]_s], \kappa\rangle}$$

Dispose

$$\frac{([\![e]\!]_s, v) \in h}{\langle s, h, \mathbf{dispose}(e); \kappa\rangle \rightsquigarrow \langle s, h \setminus \{([\![e]\!]_s, v)\}, \kappa\rangle}$$

Assign

$$\langle s, h, x := e; \kappa\rangle \rightsquigarrow \langle s[x := [\![e]\!]_s, h, \kappa\rangle$$

IfTrue

$$\frac{[\![b]\!]_s = \mathbf{true}}{\langle s, h, \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'; \kappa\rangle \rightsquigarrow \langle s, h, c; \kappa\rangle}$$

IfFalse

$$\frac{[\![b]\!]_s = \mathbf{false}}{\langle s, h, \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'; \kappa\rangle \rightsquigarrow \langle s, h, c'; \kappa\rangle}$$

Seq

$$\langle s, h, (c; c'); \kappa\rangle \rightsquigarrow \langle s, h, c; (c'; \kappa)\rangle$$

Call

$$\frac{([\![e]\!]_s, \lfloor \mathbf{lambda}\ (\overline{x})\ c \rfloor) \in h \qquad |\overline{x}| = |\overline{e}|}{\langle s, h, x := \mathbf{call}\ e(\overline{e}); \kappa\rangle \rightsquigarrow \langle (\lambda\_ \bullet 0)[\mathsf{ip} := [\![e]\!]_s][\overline{x} := \overline{e}], h, c; \mathbf{ret}(s, x, \kappa)\rangle}$$

Return

$$\langle s, h, \mathbf{ret}(s', x, \kappa)\rangle \rightsquigarrow \langle s'[x := s(\mathsf{result})], h, \kappa\rangle$$

LoadSuccess

$$\frac{([\![e]\!]_s, (\tau, (v_1, \ldots, v_m))) \in \mathrm{Modules} \qquad 0 < \ell \qquad \mathrm{dom}(h) \cap \{\ell, \ldots, \ell + m\} = \emptyset}{\langle s, h, x := \mathbf{load}\ e\ \mathbf{as}\ \tau; \kappa\rangle \rightsquigarrow \langle s[x := \ell], h[\ell := m, \ell + 1 := v_1, \ldots, \ell + m := v_m], \kappa\rangle}$$

LoadFail

$$\frac{\neg \exists \overline{v} \bullet ([\![e]\!]_s, (\tau, (\overline{v}))) \in \mathrm{Modules}}{\langle s, h, x := \mathbf{load}\ e\ \mathbf{as}\ \tau; \kappa\rangle \rightsquigarrow \langle s[x := 0], h, \kappa\rangle}$$

Unload

$$\frac{[\![e]\!]_s = \ell \qquad \{(\ell, m), (\ell + 1, v_1), \ldots, (\ell + m, v_m)\} \subseteq h}{\langle s, h, \mathbf{unload}(e); \kappa\rangle \rightsquigarrow \langle s, h \setminus \{(\ell, m), (\ell + 1, v_1), \ldots, (\ell + m, v_m)\}, \kappa\rangle}$$

Fig. 4. Step rules

flexibility in implementation, the module's incrementor function does not simply increment its argument, but additionally tracks the number of calls made to it in a dynamically allocated cell. The cell is allocated by the module's entry point, and its address is stored in the third global variable. The first two global variables will serve as the struct that is returned to the client by the module's entry point. The entry point writes a pointer to the incrementor function and the dispose function into the struct, allocates the counter cell, and then returns the address of the struct. The incrementor function, whose code resides in the second element of the module image, increments the counter cell and then returns its argument plus one. The dispose function, which resides in the third position, simply disposes the counter cell.

```
Modules = {
  (myIncrLib, (getIncrType, (
    ⌊lambda () [ip + 3] := ip + 1;
        [ip + 4] := ip + 2; x := cons(0); [ip + 5] := x; result := ip + 3⌋,
    ⌊lambda (x) c := [ip + 4]; n := [c]; [c] := n + 1; result := x + 1⌋,
    ⌊lambda () x := [ip + 3]; dispose(x)⌋,
    0,
    0,
    0
  )))
}
main =
  ℓ := load libName as getIncrType;
  if ℓ = 0 then x := x else (
    getIncr := ℓ + 1; o := call getIncr();
    incr := [o]; r := call incr(42);
    if r = 43 then x := x else [0] := 0;    // assert(r = 43)
    dispose := [o + 1]; dummy := call dispose();
    unload ℓ
  )
```

Fig. 5. Example program in the formal language

5.2 Specification Language

The specification language is an extension of separation logic with special-purpose predicates **lib**, **module**$_0$, and **module**, for dealing with module loading and unloading, as well as with parameterized function types and assertion closures for dealing with dynamic code more generally.

Expressions in assertions may be constants, program variables, logical variables, operator applications, and assertion closure expressions. Evaluation of an

assertion closure expression yields an assertion closure C, which records the interpretation I of the free logical variables of the assertion and the store s that provides the value of the free program variables of the assertion.

The assertion $E : \tau(\overline{E})$ asserts that the address denoted by E satisfies the function type τ instantiated with arguments $\overline{E}$. The assertion closure application expression $E(\overline{E})$ applies arguments $\overline{E}$ to the assertion closure denoted by E. $\mathbf{lib}(E, E')$ denotes the management information (i.e. the size field) for the loaded module named E' at address E. $\mathbf{module}_0(E, E')$ asserts that the module image of the module named E' is at address $E + 1$ and following, and it is in its initial state. $\mathbf{module}(E, E')$ asserts that there are m allocated memory locations at addresses $E + 1$ and following, where m is the size of the module named E'.

Assertions are interpreted in the context of a set of function type declarations and named predicate declarations. A function type declaration specifies a function type name, a list of function type parameters, a list of function parameters, a precondition, and a postcondition. A predicate declaration specifies a predicate name, a list of parameters, and a body.

$$
\begin{array}{rl}
 & n \in \mathbb{Z}, x \in \text{ProgVars}, y \in \text{LogVars}, op \in \text{Operators} \\
E ::= & n \mid x \mid y \mid op(\overline{E}) \mid \mathbf{lambda}\ (\overline{y})\ A \\
C ::= & \mathbf{lambda}_{I,s}\ (\overline{y})\ A \\
A ::= & \mathbf{emp} \mid E \mapsto E \mid E = E \mid A \wedge A \mid A \vee A \mid A * A \mid \exists y \bullet A \\
 & \mid E : \tau(\overline{E}) \mid p(\overline{E}) \mid E(\overline{E}) \\
 & \mid \mathbf{lib}(E, E) \mid \mathbf{module}_0(E, E) \mid \mathbf{module}(E, E) \\
\mathit{ftdecl} ::= & \mathbf{funtype}\ \tau(\overline{y})(\overline{x})\ \mathbf{req}\ A\ \mathbf{ens}\ A \\
\mathit{pdecl} ::= & \mathbf{predicate}\ p(\overline{y}) = A
\end{array}
$$

The semantics of assertion expressions, assertions, and function type judgments is given in Figure 6. The figure uses the following auxiliary notions derived from the semantics of the programming language.

$$
\begin{array}{l}
\mathsf{Stuck} = \{\gamma \mid \neg\exists\gamma' \bullet \gamma \rightsquigarrow \gamma'\} \\
\mathsf{Bad} = \{\gamma \mid \exists\gamma' \in \mathsf{Stuck} \bullet \gamma \rightsquigarrow^* \gamma'\} \\
\mathsf{Pre}(c) = \{(s, h) \mid \langle s, h, c; \mathbf{done}\rangle \notin \mathsf{Bad}\} \\
\mathsf{Post}(c) = \{((s, h), (s', h')) \mid \langle s, h, c; \mathbf{done}\rangle \rightsquigarrow^* \langle s', h', \mathbf{done}\rangle\}
\end{array}
$$

We consider a configuration γ to be bad if it can lead to a stuck configuration. The precondition of a command is the set of initial states that will not lead to a stuck configuration. The postcondition relates the pre-states and the post-states.

A function type judgment $\ell : \tau(\overline{v})$ holds, i.e., an address ℓ satisfies function type τ instantiated with arguments $\overline{v}$, if and only if the instantiated precondition of the function type implies that there is a function value at address ℓ that satisfies the function type's instantiated contract.

In general, the equations of Figure 6 do not qualify as definitions, since they may have multiple solutions, or no solution at all. To avoid this, we impose the following restrictions. We impose an order on function types and predicates, so that each of these may mention only earlier ones in its definition. Further, we do not allow assertion closure applications inside assertion closures. We believe

$$
\begin{array}{l}
[\![x]\!]_{I,s} = s(x) \\
[\![y]\!]_{I,s} = I(y) \\
[\![op(\overline{E})]\!]_{I,s} = [\![op]\!]([\![\overline{E}]\!]_{I,s}) \\
[\![\mathbf{lambda}\ (\overline{y})\ A]\!]_{I,s} = \lfloor \mathbf{lambda}_{I|_{\mathrm{freeLogVar}(A)\setminus\{\overline{y}\}}, s|_{\mathrm{freeProgVar}(A)}}\ (\overline{y})\ A \rfloor
\end{array}
$$

$$
\begin{array}{lll}
I,s,h \vDash \mathbf{emp} & \Leftrightarrow h = \emptyset \\
I,s,h \vDash E \mapsto E' & \Leftrightarrow h = \{([\![E]\!]_{I,s}, [\![E']\!]_{I,s})\} \\
I,s,h \vDash E = E' & \Leftrightarrow [\![E]\!]_{I,s} = [\![E']\!]_{I,s} \\
I,s,h \vDash A \wedge A' & \Leftrightarrow I,s,h \vDash A \wedge I,s,h \vDash A' \\
I,s,h \vDash A \vee A' & \Leftrightarrow I,s,h \vDash A \vee I,s,h \vDash A' \\
I,s,h \vDash A * A' & \Leftrightarrow \exists h_1, h_2 \bullet h = h_1 \uplus h_2 \wedge I,s,h_1 \vDash A \wedge I,s,h_2 \vDash A' \\
I,s,h \vDash \exists y \bullet A & \Leftrightarrow \exists n \in \mathbb{Z} \bullet I[y := n], s, h \vDash A \\
I,s,h \vDash E : \tau(\overline{E}) & \Leftrightarrow [\![E]\!]_{I,s} : \tau([\![\overline{E}]\!]_{I,s}) \\
I,s,h \vDash p(\overline{E}) & \Leftrightarrow \exists \overline{y}, A \bullet (\mathbf{predicate}\ p(\overline{y}) = A) \\
 & \qquad \wedge\ (\lambda\_ \bullet 0)[\overline{y} := [\![\overline{E}]\!]_{I,s}], (\lambda\_ \bullet 0), h \vDash A \\
I,s,h \vDash E(E_1,\ldots,E_m) & \Leftrightarrow \exists I', s', y_1,\ldots,y_m, A \bullet [\![E]\!]_{I,s} = \lfloor \mathbf{lambda}_{I',s'}\ (y_1,\ldots,y_m)\ A \rfloor \\
 & \qquad \wedge\ I'[y_1 := [\![E_1]\!]_{I,s}] \cdots [y_m := [\![E_m]\!]_{I,s}], s', h \vDash A \\
I,s,h \vDash \mathbf{lib}(E,E') & \Leftrightarrow \exists (M, (\tau, (v_1,\ldots,v_m))) \in \mathrm{Modules} \bullet \\
 & \qquad [\![E']\!]_{I,s} = \lfloor M \rfloor \wedge h = \{([\![E]\!]_{I,s}, m)\} \\
I,s,h \vDash \mathbf{module}_0(E,E') & \Leftrightarrow \exists (M, (\tau, (v_1,\ldots,v_m))) \in \mathrm{Modules}, \ell \in \mathbb{Z} \bullet \\
 & \qquad [\![E]\!]_{I,s} = \ell \wedge [\![E']\!]_{I,s} = \lfloor M \rfloor \wedge \\
 & \qquad h = \{(\ell+1, v_1), \ldots, (\ell+m, v_m)\} \\
I,s,h \vDash \mathbf{module}(E,E') & \Leftrightarrow \exists (M, (\tau, (v_1,\ldots,v_m))) \in \mathrm{Modules}, \ell, v'_1,\ldots,v'_m \in \mathbb{Z} \bullet \\
 & \qquad [\![E]\!]_{I,s} = \ell \wedge [\![E']\!]_{I,s} = \lfloor M \rfloor \wedge \\
 & \qquad h = \{(\ell+1, v'_1), \ldots, (\ell+m, v'_m)\}
\end{array}
$$

$$
\begin{array}{l}
\ell : \tau(\overline{v}) \ \Leftrightarrow \\
\quad \forall I,s,h \bullet I(\overline{y}) = \overline{v} \wedge I,s,h \vDash P \Rightarrow \\
\qquad \exists L, c \bullet (\ell, \lfloor L \rfloor) \in h \wedge L \approx \mathbf{lambda}\ (\overline{x})\ c \\
\qquad\quad \wedge\ (s,h) \in \mathsf{Pre}(c) \\
\qquad\quad \wedge\ \forall s', h' \bullet ((s,h),(s',h')) \in \mathsf{Post}(c) \Rightarrow \\
\qquad\qquad I, s[\mathsf{result} := s'(\mathsf{result})], h' \vDash Q \\
\text{where } \mathbf{funtype}\ \tau(\overline{y})(\overline{x})\ \mathbf{req}\ P\ \mathbf{ens}\ Q
\end{array}
$$

Fig. 6. Semantics of assertions and function type judgments. $\approx$ denotes equality of function values up to alpha conversion.

this ensures well-definedness of the semantics. Furthermore, they are sufficient for the programs we considered. Relaxing these restrictions is future work.

Figure 7 shows the definition of the contract of the example module of Figure 5. The entry point's contract getIncrType requires the module's image in its initial state, and returns the struct containing the pointer to the incrementor function and the dispose function, as well as the remainder of the module's state in the form of existentially quantified assertion closure value P. The incrementor function type incrType simply requires and ensures the module state P. The dispose function type disposeType takes back the struct with the function pointers and the module state, and returns the module image, ready for unloading.

funtype incrType$(P)(x)$
  req P
  ens $P \wedge \mathsf{result} = x + 1$

funtype disposeType$(o, P, \ell, m)()$
  req $o \mapsto \_ * o + 1 \mapsto \_ * P$
  ens **module**(ℓ, m)

funtype getIncrType$(\ell, m)()$
  req **module**$_0(\ell, m)$
  ens $\exists P, p, d \bullet \mathsf{result} \mapsto p * \mathsf{result} + 1 \mapsto d * P$
    $\wedge\ p : \mathsf{incrType}(P) \wedge d : \mathsf{disposeType}(o, P, \ell, m)$

Fig. 7. Definition of function type getIncrType

5.3 Proof System

The proof system is an extension of the proof rules of separation logic with proof rules for deriving function type judgments, and for verifying function pointer call and module load and unload commands, and for folding and unfolding the module assertions. The new proof rules are shown in Figure 8.

Rule C-FunType allows one to derive a function type judgment guarded by a pure boolean expression ϕ. The latter serves to express constraints on the free logical variables that appear in the judgment. The other rules are straightforward.

We can now define validity of a module.

Definition 1. *A module is valid if its entry point satisfies the function type given by its contract, instantiated with the address where the module is loaded and the module name.*

$$(M, (\tau, \overline{v})) \in \mathit{Modules} \Rightarrow (\mathsf{valid}(M) \Leftrightarrow \vdash \forall \ell \bullet \ell + 1 : \tau(\ell, M))$$

C-FunType

$$\frac{\begin{array}{c}\textbf{funtype}\ \tau(\overline{y})(\overline{x})\ \textbf{req}\ P\ \textbf{ens}\ Q \\ \text{freeLogVars}(\phi, E, \overline{E}) \cap \text{freeLogVars}(\overline{y}, P, Q) = \emptyset \qquad \text{freeProgVars}(\phi, E, \overline{E}) = \emptyset \\ \phi \wedge P[\overline{E}/\overline{y}] \Rightarrow E \mapsto \lfloor \textbf{lambda}\ (\overline{x})\ c \rfloor * \textbf{true} \qquad \{\phi \wedge P[\overline{E}/\overline{y}] \wedge \mathsf{ip} = E\}\ c\ \{Q\}\end{array}}{\phi \Rightarrow E : \tau(\overline{E})}$$

C-Call

$$\frac{\textbf{funtype}\ \tau(\overline{y})(\overline{x})\ \textbf{req}\ P\ \textbf{ens}\ Q}{\{e : \tau(\overline{y}) \wedge \overline{e} = \overline{y'} \wedge P[\overline{y'}/\overline{x}]\}\ x := \textbf{call}\ e(\overline{e})\ \{Q[\overline{y'}/\overline{x}, x/\mathsf{result}]\}}$$

C-Load

$$\frac{}{\begin{array}{l}\{\textbf{emp} \wedge e = y\} \\ x := \textbf{load}\ e\ \textbf{as}\ \tau \\ \{x = 0 \wedge \textbf{emp} \vee 0 < x \wedge \textbf{lib}(x, y) * \textbf{module}_0(x, y) \wedge x + 1 : \tau(x, y)\}\end{array}}$$

C-Module-Unfold

$$\frac{(M, (\tau, (v_1, \ldots, v_m))) \in \text{Modules}}{\textbf{module}_0(y, M) \Rightarrow y + 1 \mapsto v_1 * \ldots * y + m \mapsto v_m}$$

C-Module-Fold

$$\frac{(M, (\tau, (v_1, \ldots, v_m))) \in \text{Modules}}{y + 1 \mapsto \_ * \cdots * y + m \mapsto \_ \Rightarrow \textbf{module}(y, M)}$$

C-Unload

$$\frac{}{\{\textbf{lib}(e, y) * \textbf{module}(e, y)\}\ \textbf{unload}(e)\ \{\textbf{emp}\}}$$

Fig. 8. Proof rules

The proof of the example module is straightforward. Figure 9 gives an outline. The assertion closure parameter P is instantiated with predicate Q applied to the address where the module is loaded. This predicate encompasses the module image, plus the counter cell, minus the struct containing the function pointers.

We can now state the soundness theorem of our approach. It uses the notion of a valid command.

Definition 2 (Valid Command). *A command c is valid if is satisfies the following triple:*

$$\vdash \{\textbf{emp}\}\ c\ \{\textbf{true}\}$$

Theorem 1 (Soundness). *If each module in the module table is valid, and the main program is valid, then execution does not get stuck.*

A soundness proof is future work.

predicate $\mathsf{Q}(\ell)$ =
$\ell + 1 \mapsto \lfloor$**lambda** () $[\mathsf{ip} + 3] := \mathsf{ip} + 1;$
$[\mathsf{ip} + 4] := \mathsf{ip} + 2; x := \mathbf{cons}(0); [\mathsf{ip} + 5] := x; \mathsf{result} := \mathsf{ip} + 3\rfloor *$
$\ell + 2 \mapsto \lfloor$**lambda** (x) $c := [\mathsf{ip} + 4]; n := [c]; [c] := n + 1; \mathsf{result} := x + 1\rfloor *$
$\ell + 3 \mapsto \lfloor$**lambda** () $x := [\mathsf{ip} + 3]; \mathbf{dispose}(x)\rfloor *$
$\exists c \bullet \ell + 6 \mapsto c * c \mapsto \_$

$\forall \ell \bullet \ell + 2 : \mathsf{incrType}($**lambda** () $\mathsf{Q}(\ell))$ by C-FunType (1)
$\forall \ell \bullet \ell + 3 : \mathsf{disposeType}(\ell + 4,$ **lambda** () $\mathsf{Q}(\ell), \ell, \mathsf{myIncrLib})$ by C-FunType, C-Module-Fold (2)
$\forall \ell \bullet \ell + 1 : \mathsf{getIncrType}(\ell, \mathsf{myIncrLib})$ by C-FunType,C-Module-Unfold,(1),(2)

Fig. 9. Proof outline of the validity of module myIncrLib

6 Verification Tool

We implemented the approach in our prototype verifier, VeriFast, and we verified a small server that allows clients to load modules, unload modules, and use services provided by the modules, mimicking operating system kernels that may dynamically load and unload device drivers. The implementation and the example are online at http://www.cs.kuleuven.be/~bartj/verifast/.

7 Future work

The approach, as presented in this paper, is directly applicable to some important instances of dynamic code loading and unloading, notably loadable device drivers in the Linux kernel. However, in certain other cases, there are additional complexities that are not yet addressed by the approach.

Specifically, the user-space shared library mechanisms of most operating systems (notably Unix-like or Windows operating systems) are complicated by the fact that loading a module might yield a new reference to an existing instance of the module, rather than a fresh instance of the module. Therefore, in this context it is not sound for the proof rule for loading a library to grant full permission to the module's code and global variables.

Another item of future work is to add support for libraries or programs consisting of multiple modules. We envisage extending the specification language to allow a module B to *import* another module A. A module B's **module** predicate would then denote not just B's code and global variables, but would include A's **module** predicate as well. A link-time check would ensure that there is no cycle in the module import graph.

A third item of future work is to investigate the interaction between the previous two: how to verify a program or library that specifies a load-time dependency on another library? Here, too, the library sharing issue arises: if two libraries B and C specify a dependency on some library A, then B and C will be linked at load time against the same instance of library A.

8 Related Work

The approach presented in this paper builds on and extends separation logic. Existing verification systems for separation logic include Smallfoot [2], jStar [6] and YNOT [3]. However, to the best of our knowledge, none of these systems can be used for verifying programs that use unloadable code.

9 Conclusion

Existing verification approaches for C programs (VCC [4], Frama-C [1], HAVOC [5], Smallfoot [2], our own verifier VeriFast [7], Jahob [9]) cannot be used for verifying programs that involve unloading of code as they assume that the code is unchanging and is not part of the mutable state. In this paper, we propose a novel separation-logic-based approach for extending a verification approach for C programs to enable verification of programs involving code unloading.

Bibliography

[1] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C specification language.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2006.

[3] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, 2009.

[4] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, number 5674 in LNCS, 2009. To appear.

[5] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *POPL*, 2009.

[6] Dino Distefano and Matthew Parkinson. jStar: Towards practical verification for java. In *OOPSLA*, 2008.

[7] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008.

[8] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, 2005.

[9] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *PLDI*, 2008.