Ph.D. DISSERTATION

RUSC and CompCertM: A new foundation for modular verification and its application to compiler verification

RUSC와 CompCertM: 나눠서 검증하기 위한 새로운 이론적 토대와 컴파일러 검증에의 적용

BY

Youngju Song

FEBRUARY 2021

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

RUSC and CompCertM: A new foundation for modular verification and its application to compiler verification

RUSC와 CompCertM: 나눠서 검증하기 위한 새로운 이론적 토대와 컴파일러 검증에의 적용

BY

Youngju Song

FEBRUARY 2021

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

RUSC and CompCertM: A new foundation for modular verification and its application to compiler verification

RUSC와 CompCertM: 나눠서 검증하기 위한 새로운 이론적 토대와 컴파일러 검증에의 적용

지도교수 허 충 길

이 논문을 공학박사 학위논문으로 제출함

2021 년 1 월

서울대학교 대학원

전기 컴퓨터 공학부

송용주

Youngju Song의 공학박사 학위논문을 인준함

2021 년 1 월

위 원 장	이광근
부위원장	허충길
위 원	이우석
위 원	허기홍
위 원	강지훈

Abstract

Modern software systems are complex. To verify such a system, it is critically important to have a modular verification technique. However, none of the existing approaches are satisfactory. In this dissertation, we develop a novel modular verification technique, called RUSC (Refinement Under Self-related Contexts), and demonstrate its usefulness by applying it to both compiler verification and program verification.

On the one hand, RUSC advances the state-of-the-art on compiler verification. Specifically, we develop CompCertM, a full extension of CompCert supporting multi-language linking without any restrictions but still with low verification overhead, thus surpassing the two state-of-the-arts, CompCertX and Compositional CompCert. On the other hand, RUSC as a program verification technique is in its early stage but shows considerable potential. Compared to higher-order separation logic, our approach provides simpler specifications and stronger results, but its verification overhead is much higher and does not support advanced features yet.

Keywords: C, Modularity, Compositional Compiler Verification, CompCert, Separate Compilation, Multi-Language Linking
Student Number: 2015-21244

Contents

List of Figures

List of Tables

Chapter I

Overview

Modern software systems are complex, and modularity is the crucial tool for coping with such complexity. That is, software systems are typically broke up into multiple modules so that software developers can focus on a single module at a time. Moreover, instead of directly writing each module in assembly, they are mostly written in high-level languages such as C. Then compilers translate such high-level languages into assembly via multiple translation passes.

To verify such a system, it is critically important to have a modular verification technique. Specifically, the verification can be divided into two parts, compiler verification and program verification. The former is about proving the generated assembly code behaves as specified by the source program's semantics, and the latter is about proving desired properties about the behaviors of the source program. In compiler verification, we want to focus on a single compiler and its single translation at a time, even though each module is written in different languages and compiled with different compilers. Similarly, in program verification, we want to focus on a single module and its single abstraction at a time, assuming interfaces of other modules.

However, none of the existing approaches are satisfactory. For compiler verification, there are two state-of-the-art frameworks, CompCertX [7, 28] and Compositional CompCert (shortly, CompComp) [4, 27], supporting modular verification of multi-language systems. The former simplifies the problem by

imposing restrictions that the source modules should have no mutual dependence and be verified against certain well-behaved specifications. On the other hand, the latter develops a new verification technique that directly solves the problem but at the expense of significantly increasing the verification cost. For program verification, higher-order separation logic (such as VST[1]) has shown great success, but it also has shortcomings: its underlying model (and thus soundness result) is esoteric, it proves partial correctness instead of total correctness, and its soundness theorem is applicable only when the whole program is verified with the same logic.

In this dissertation, we develop a novel modular verification technique, called RUSC (Refinement Under Self-related Contexts) (Chapter II), and demonstrate its usefulness by applying it to both compiler verification (Chapter III) and program verification (Chapter IV). Specifically, we develop CompCertM, a full extension of CompCert supporting multi-language linking, and show how RUSC enables modular compiler verification without any restrictions but still with low verification overhead. Moreover, we verify interesting programs using RUSC as a program logic and demonstrate that it does not suffer from the drawbacks above of higher-order separation logic. Although the result shows potential, RUSC-as-a-program-logic is still in its early stage, and we discuss future research directions.

This dissertation draws heavily on the work and writing in the following paper: [25]

1 Compiler Verification

CompCert [16, 17], the first *verified optimizing* compiler for *the C programming language*, has served as a backend in end-to-end verified software [2]. Specifically, CompCert compiles programs written in (a large subset of) C down to assembly code via various translation passes including a number of common optimizations. Moreover, it is formally verified in Coq that every translation of CompCert preserves the semantics: the generated assembly code behaves as specified by the semantics of the source program. Therefore, CompCert has been used to transform verification results about the source C program into those about the compiled assembly code in various projects such as CertiKOS [6, 8] and VST [1].

There is, however, a limitation in the original CompCert that restricts its application to a more wide range of software verification—namely the lack of

support for handwritten assembly. This limitation can be serious in verification of *real-world* software because handwritten assembly is often crucial for writing low-level system software or library code.

To overcome this limitation, two extensions of CompCert, namely CompCertX [7, 28] and Compositional CompCert (shortly, CompComp) [4, 27], have been developed. Interestingly, they take different approaches to *two key challenges*:

1. how to modularly verify each translation of each module using a different relational memory invariant (shortly, memory relation) and compose the proofs all together; and

2. how to deal with illegal interference from arbitrary (handwritten) assembly modules that can invalidate compiler translations of C modules (*e.g.*, not preserving the callee-save register values).

We elaborate more on the first, more fundamental, challenge. CompCert uses three different memory relations called memory *identity*, *extension* and *injection* (in the order of complexity and generality) for a proof engineering purpose: it uses a simpler relation whenever possible to simplify the correctness proof. The challenge occurs in an open setting where a translation of an open module is verified separately. In a closed setting as in CompCert where the whole closed program (*i.e.*, all the modules) is compiled by the same translation pass thereby being verified as a whole, verification of such a closed program using a simpler relation essentially implies that using a more general one. However, in an open setting (*i.e.*, for verification of an open module), that implication does not hold because such verification assumes that the unknown contexts also preserve the same memory relation. In other words, using a simpler relation, the verification guarantees a stronger property on its own module but assumes a stronger property on the context modules. Therefore, verification of open modules using different memory relations cannot be compared, which makes composition of such verifications hard.

CompCertX's Approach CompCertX is developed as a backend compiler for the verified OS kernel CertiKOS [6, 8] and thus specialized for this purpose. Specifically, CompCertX simplifies the two challenges by making two assumptions that (*i*) there are no mutual dependencies among the input modules and (*ii*) each

input module is verified against a well-behaved specification, called *Certified Abstraction Layer (CAL)*.

First, these assumptions enable CompCertX to use *closed* simulations, the simple verification technique used by the original CompCert. The simulations are closed in the sense that they relate known source and target functions under the condition that all invoked unknown functions have independent good behaviors. Specifically, the unknown functions (*i*) provide full end-to-end behaviors regardless of who the caller is (*i.e.*, whether it is the source or the target); and (*ii*) those behaviors satisfy a certain good-behavior property. Note that these two requirements for closed simulations directly follow from the two assumptions of CompCertX above, respectively. Then proving compositionality between closed simulations using the three different types of memory relations is straightforward as discussed above (*i.e.*, verification using a simpler relation implies that using a more general one). As a result, the correctness proofs of all compiler passes using closed simulations in CompCertX are only 15.51% larger than those in the original CompCert 3.0.1 in terms of significant lines of code (SLOC)[1], and the metatheory (*i.e.*, all the rest) is 47.65% larger.

Second, thanks to the assumptions of CompCertX, interference from assembly modules is also handled simply. The assumption that handwritten assembly modules are verified against CAL specifications implies that those modules do not cause any illegal interference (*i.e.*, well-behaved).

CompComp's Approach CompComp establishes a more general correctness result without the restrictions of CompCertX but at the expense of using a more heavyweight verification technique of its own, called *structured simulations*. They are in the form of *open* simulations in the sense that they allow invoked unknown functions to depend on their callers (*e.g.*, via mutual recursion). Since this openness technically makes compositionality proofs much harder as discussed above, to simplify them CompComp uses a single memory relation, called *structured injection*. For this reason, the verification technique is less flexible. Specifically, the proofs of the whole compiler passes using the structured injection deviate quite far from the original proofs in CompCert and require significantly more efforts: the correctness proofs of all compiler passes are 145.77% larger than those in the original CompCert 2.1, and the metatheory is 81.77% larger.

[1] we counted SLOC using `coqwc`.

Also, CompComp handles interference from assembly modules more generally without assuming the good-behavior property for input modules. Since such interference only occurs via the register file and the function arguments area of the stack (*i.e.*, the shared resources that exist in assembly but not in C), the *interaction semantics* of CompComp, which gives a logical semantics to programs consisting of multi-language modules, duplicates those resources for each invocation of an assembly module and does not propagate any illegal effects outside the module.

However, the treatment comes with no adequacy proof with respect to the physical semantics. Indeed, interaction semantics is not adequate: due to the logical isolation of illegal effects, the interaction semantics of linked assembly modules deviates from their *physical* semantics (*i.e.*, the assembly semantics of CompCert) when one of the modules indeed causes illegal interference, for example, by not preserving the callee-save register values. Note that this problem was also observed and discussed in the PhD thesis of [26] (see Section 14 for comparison).

Finally, there is another difference between CompComp and CompCertX: CompComp only supports C-style calling conventions, while CompCertX additionally supports assembly-style calling conventions (*i.e.*, imposing no conditions except on the return address) between assembly modules.

Our Approach In this dissertation, we develop a new framework achieving both the flexibility of CompCertX and the generality of CompComp. We demonstrate its power as a compiler verification framework by applying it to CompCert. Specifically, we develop:

- Open (Mixed) Simulations: a simpler version of structured simulations, (*i*) allowing arbitrary memory relations including memory identity, extension and injection, and (*ii*) supporting mixed forward-backward simulation;
- RUSC (Refinement Under Self-related Contexts): our new lightweight theory for composing arbitrary open simulations together, which is the highlight of our theoretical contribution;
- Repaired Interaction Semantics: providing adequacy w.r.t. the physical semantics and additionally supporting assembly-style calling conventions;
- CompCertM: CompCert v3.5 fully extended with the repaired interaction semantics and open simulations to support multi-language linking (18.73%

larger in the correctness proofs of all compiler passes, and 32.59% larger in the metatheory);

- `Unreadglob`: a new optimization pass we added that eliminates all unread static variables and instructions writing to them, whose verification for *open* modules requires a new kind of memory relation, *memory injection with module-local invariants*;

The key theory enabling all these results is RUSC, which takes a set of (almost arbitrary) open simulations $\mathcal{R}$ and lifts them to a larger relation $\succcurlyeq_{\mathcal{R}}$ that is fully compositional. The idea is inspired by the situation where the transitivity problem of logical relations is avoided by proving their inclusion in the contextual refinement, which is trivially transitive. To increase its applicability, RUSC simply generalizes the notion of contextual refinement (CR) by parameterizing over a set of program relations $\mathcal{R}$. Specifically, we say that $p \succcurlyeq_{\mathcal{R}} q$ if for any context C that is related to itself by every relation in $\mathcal{R}$, the observable behaviors of $C[p]$ are refined by those of $C[q]$. The key idea is to give the notion of well-behaved contexts w.r.t. a set of program relations $\mathcal{R}$ as those that are self-related by every relation in $\mathcal{R}$. The intuition behind it is that a context self-related by a program relation R preserves all the invariants of the relation R. The merits of RUSC are that RUSC is (*i*) unlike CR, applicable even in the presence of ill-behaved contexts, which is the case in our setting, and (*ii*) fully compositional like CR. By setting $\mathcal{R}$ as the set of open simulations with four kinds of memory relations—the three relations used by CompCert and our new relation, memory injection with module-local invariants—we can freely choose one of them in verification of a compiler pass, or a program against its specification.

Also, to generally support forward simulation in the presence of nondeterminism, we implement the notion of mixed forward-backward simulation from [19] with a slight generalization needed for CompCert (see Section 8.3).

We repair the interaction semantics of CompComp by defining those behaviors causing illegal interference as *undefined behaviors* (UBs)[2], which, however, required a few nontrivial ideas. First, we identify the sources of inadequacy of interaction semantics as those behaviors violating three assumptions—seen as a part of the official calling convention—made by standard compilers such as GCC and LLVM with concrete counterexamples. Second, to make those

[2]UBs can be understood as forbidden behaviors, so that compilers are licensed to translate them into *any* behaviors.

illegal behaviors UBs, we strengthened only the interaction part of interaction semantics without changing the underlying language semantics of CompCert, which indeed is quite nontrivial as discussed in Section 7. Finally, we prove two adequacy results: (*i*) the interaction semantics of linked assembly modules is refined by their physical semantics, and (*ii*) the physical semantics (*i.e.*, the language semantics of CompCert) of linked (typed-checked) C modules is refined by their interaction semantics. These results mean that the repaired interaction semantics does not give too few behaviors to assembly programs (*e.g.*, missing physically observable behaviors), nor does it give too many behaviors to well-typed C programs (*e.g.*, giving UB to them).

CompCertM is a full extension of CompCert 3.5 without missing any translation pass and without changing the underlying semantics, which is developed in two steps. First, we refactored the proofs of the original CompCert to get CompCertR, where the main parts of the correctness proof of each pass is separated out as a main lemma that can be later used for both closed and open simulation proofs. CompCertR gives exactly the same results as CompCert with only 4.41% increase in the correctness proofs of all passes and 2.74% increase in the metatheory. Then, on top of CompCertR, we developed an add-on package, CompCertM pack, supporting interaction semantics and multi-language linking. CompCertM reuses all the main lemmas of CompCertR and adds (*i*) additional proofs to reason about the interaction parts of interaction semantics in the correctness proofs of all passes, which amount to 14.32% of the original proofs in CompCert, and (*ii*) additional metatheory including interaction semantics and RUSC, which amounts to 29.85% of the original metatheory in CompCert.

Finally, a newly added optimization, `Unreadglob`, shows the flexibility of our framework: allowing arbitrary memory relations.

2 Program Verification

Program verification is about proving the source program behaves according to mathematical specifications. Together with the compiler verification, this constitutes an end-to-end verification of a system. Like compiler verification, it is critically important to have a modular reasoning principle that allows one to verify each module against its specification by a series of gradual abstractions.

However, program verification is distinct from compiler verification in few ways. First, we need a way to describe mathematical specifications. Several existing works [18, 12, 1, 7] take different approaches on this with pros and

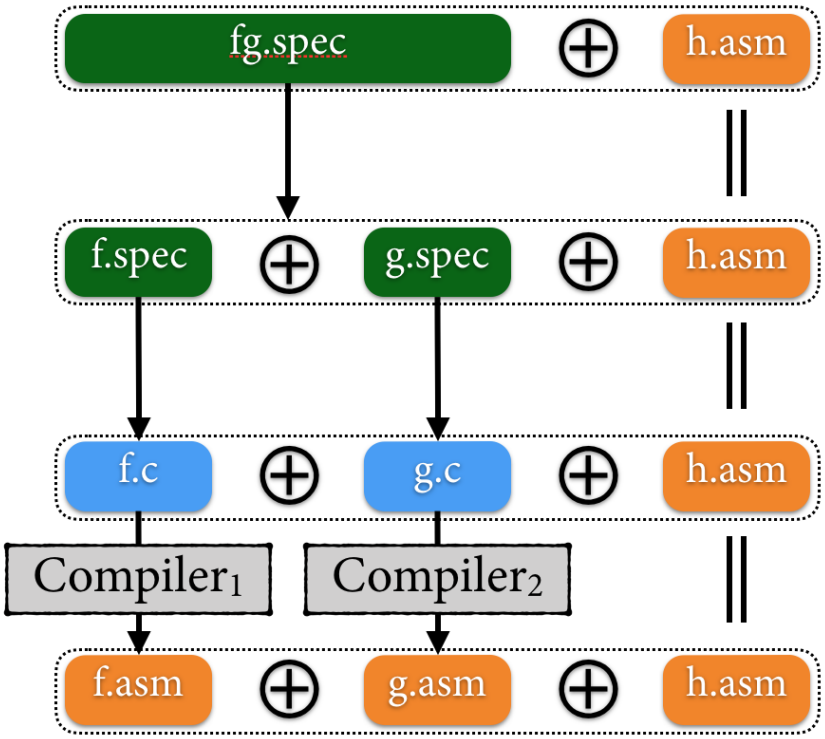

Figure I.1: An end-to-end verification scenario

cons. The second difference is that compilers are *conservative*; when a compiler translates a module, it does not assume any property about external function's implementation, and thus the translation is sound under arbitrary C context. However, in program verification, we need to assume the external function's behavior, and that function should meet the assumption. For example, suppose there is a source program that uses a sort function from an external library. Then, the module is correct (behaves according to a mathematical specification) only when the sort function behaves as expected. In other words, in program verification, it is crucial to support *cooperation* between different modules naturally. Third, compilers are general, which means that they are designed to serve arbitrary C code, so their translations are relatively more straightforward. However, in program verification, we write specifications for each C module, so each abstraction is unique, requiring a deeper understanding of its algorithmic nature.

Now we explain how we can apply RUSC to program verification with an example (Figure I.1) where two mutually recursive modules, `f.c` and `g.c`, are

verified against its specifications even in the presence of an unknown assembly module `h.asm`.

First, we write a specification as a *module*, not in C but an abstract mathematical state transition system. For instance, if we have a C module computing Fibonacci number employing dynamic programming technique, its specification module directly returns Fib(`n`) for an argument `n` where Fib is a Gallina function. Existing works[18, 12, 1, 7] often take different choices with us, where they write specifications in a specific language, as a *Hoare triple*, or as a CAL. We use a state transition system because it is the most general form, and fortunately, it is already supported in interaction semantics.

Then, each implementation module (written in C or assembly) is verified against its specification module using open simulations, thus implying RUSC relation. The challenge here is: How to utilize the other modules' specifications even though what we are proving is RUSC, which quantifies over an arbitrary context? Our key idea is that, instead of directly assuming the specifications of other modules, we carefully massage each specification module with UB in order to give an illusion as if we are assuming specifications of other modules. Specifically, `f.spec` calls `g` – which can be an arbitrary function because we are proving under an arbitrary context – and then check if `g` returns the expected value. If so, it will proceed, but if not, it will trigger UB. As a result, when verifying `f.spec` $\succcurlyeq_{\mathcal{R}}$ `f.c` one needs to proceed with the simulation proof only when `g` behaves as expected because otherwise, the proof is trivial. After each module's verification is over, we can *merge* these modules to form `fg.spec` and remove the potential source of UB. This way, we can handle cooperation among the modules naturally.

Finally, recall that RUSC lifts (almost) arbitrary open simulations. Therefore, even though verifying `f.c` against `f.spec` requires a specialized simulation, RUSC can ably support them.

We demonstrate RUSC's potential as a program verification framework with interesting examples, for which we write mathematical specifications as abstract modules in interaction semantics and prove refinement between the examples and their specification modules. Specifically, we develop:

- Verification of `utod`: providing a correctness proof against its specification module using an open simulation, where `utod` is a handwritten assembly function casting unsigned long to double, whose correctness against its specification is axiomatized in CompCert but not any more in CompCertM;

- `mutual-sum`: an example consisting of (*i*) C and handwritten assembly modules that mutually recursively compute summation up to a given integer, performing memoization using module-local static variables, and (*ii*) correctness proofs against their specification modules using open simulations with the new memory relation, memory injection with module-local invariants.

Our approach is promising – it does not suffer from a few drawbacks of higher-order separation logic – but is still in early-stage, and further research is needed. Specifically, we give clean and understandable specifications, prove total correctness, and our results are sound under arbitrary context, but we do not support advanced features (*e.g.*, concurrency) yet, and our verification overhead is relatively high. We will give a more comprehensive comparison and discuss future research directions in Section 17 and 18.

All our results are formalized in Coq, and it is available at:

`https://sf.snu.ac.kr/compcertm`

Chapter II

RUSC

We first give background on compiler verification techniques including the notions of closed and open simulations (Section 3), discuss the problems with open simulations (Section 4) and present our solution (Section 5).

3 Background

CompCert's Verification CompCert's correctness establishes *behavioral refinement* (also called *semantics preservation*) saying that the set of all observable behaviors of a source program P, denoted $\texttt{Beh}(P)$ (seen as a specification), includes that of its compiled target program Q, *i.e.*, $\texttt{Beh}(Q)$ (seen as an implementation). Here an observable behavior of a program (either in C, assembly, or an intermediate language) is a (finite or infinite) trace of observable events (typically, invocation of system calls) occurring in a sequence of execution steps according to the language semantics.

The semantics of a language $\mathbb{L}$ is given by a loading function $\uparrow \in \mathrm{Prog}(\mathbb{L}) \to \mathrm{Mem} \times \mathrm{State}(\mathbb{L})$ from programs to *machine states* consisting of a memory and a *language state*, and a step relation $\hookrightarrow \subseteq (\mathrm{Mem} \times \mathrm{State}(\mathbb{L})) \times \mathrm{Event} \times (\mathrm{Mem} \times \mathrm{State}(\mathbb{L}))$ between machine states producing an event. Specifically, $\uparrow P$ denotes the initial machine state after loading the program P, and $(m, s) \overset{e}{\hookrightarrow} (m', s')$ denotes that the machine state (m, s) can transition to (m', s') producing an (observable or

silent) event e in a single step of execution.

CompCert is a multi-pass compiler and the whole verification is performed modularly by composing independent verification of each pass. Specifically, verification of a pass proves that the source and target programs of every translation performed by the pass are related by a certain relation, called (closed) *simulation*, to be described below. Since simulation relations are closed under composition, every end-to-end translation, which is a composition of translations of all passes, is also related by a simulation relation. Finally, CompCert's correctness follows from the fact that every simulation relation implies behavioral refinement between the related programs.

In fact, there are two versions of simulations, *forward* and *backward*. The former is more convenient for compiler verification but implies behavioral refinement only when the target language is deterministic[1]. Since CompCert mostly uses forward simulations, we will also focus on forward ones throughout the dissertation and discuss how to mix forward and backward simulations to support forward reasoning even when the target language is not deterministic in Section 8.3.

We say a translation of a program P into Q is related by a relation R between machine states if the loaded initial states $\uparrow P$ and $\uparrow Q$ are related by R. Then R is called a (closed forward) simulation if for any pair of machine states $(ms_{\mathtt{src}}, ms_{\mathtt{tgt}})$ related by R, the target state $ms_{\mathtt{tgt}}$ simulates one step execution of the source state $ms_{\mathtt{src}}$ (up to silent steps, denoted τ) and the resulting states are again related by R (slightly simplified for presentation purposes):

$$\begin{aligned} \forall (ms_{\mathtt{src}}, ms_{\mathtt{tgt}}) \in R,\ \forall e, ms'_{\mathtt{src}}, \quad & ms_{\mathtt{src}} \overset{e}{\hookrightarrow} ms'_{\mathtt{src}} \implies \\ & \exists ms'_{\mathtt{tgt}}, \quad ms_{\mathtt{tgt}} \overset{\tau}{\hookrightarrow}^{*} \overset{e}{\hookrightarrow} \overset{\tau}{\hookrightarrow}^{*} ms'_{\mathtt{tgt}} \wedge (ms'_{\mathtt{src}}, ms'_{\mathtt{tgt}}) \in R \ . \end{aligned}$$

CompComp's Verification The interaction semantics of CompComp gives a way to execute an *open* module M (*i.e.*, invoking external functions defined outside M) in isolation by providing a logical mechanism to reflect possible interference from external function calls. More specifically, the semantics provides two meta-level functions `at_external` and `after_external`. First, `at_external` s = `Some` $(f, \vec{v})$ denotes that at language state s, an external function pointed to by a function pointer f is called with arguments $\vec{v}$. Second,

[1] CompCert uses a slightly different condition, namely that the source language is *receptive* and the target is *determinate*.

1: $\forall w,\ \forall (m_{\mathtt{src}}, m_{\mathtt{tgt}}) \in \mathtt{mrel}(w),\ \forall s_{\mathtt{src}}, s_{\mathtt{tgt}},\ ((m_{\mathtt{src}}, s_{\mathtt{src}}), (m_{\mathtt{tgt}}, s_{\mathtt{tgt}})) \in R(w) \implies$

2: **match** `at_external` $s_{\mathtt{src}}$ **with**

3: | `Some` $(f_{\mathtt{src}}, \vec{v}_{\mathtt{src}}) \Rightarrow$

4: $\exists f_{\mathtt{tgt}}, \vec{v}_{\mathtt{tgt}},$ `at_external` $s_{\mathtt{tgt}} =$ `Some` $(f_{\mathtt{tgt}}, \vec{v}_{\mathtt{tgt}}) \wedge$

5: $\boxed{(f_{\mathtt{src}}, f_{\mathtt{tgt}}) \in \mathtt{vrel}(w) \wedge (\vec{v}_{\mathtt{src}}, \vec{v}_{\mathtt{tgt}}) \in \overrightarrow{\mathtt{vrel}(w)}} \wedge$

6: $\boxed{\forall w' \sqsupseteq w,\ \forall (m'_{\mathtt{src}}, m'_{\mathtt{tgt}}) \in \mathtt{mrel}(w'),}\ \boxed{\forall (r_{\mathtt{src}}, r_{\mathtt{tgt}}) \in \mathtt{vrel}(w'),}$

7: $((m'_{\mathtt{src}}, \mathtt{after\_external}\ r_{\mathtt{src}}\ s_{\mathtt{src}}), (m'_{\mathtt{tgt}}, \mathtt{after\_external}\ r_{\mathtt{tgt}}\ s_{\mathtt{tgt}})) \in R(w')$

8: | `None` $\Rightarrow$

9: $\forall e, m'_{\mathtt{src}}, s'_{\mathtt{src}},\ (m_{\mathtt{src}}, s_{\mathtt{src}}) \overset{e}{\hookrightarrow} (m'_{\mathtt{src}}, s'_{\mathtt{src}}) \implies$

10: $\exists m'_{\mathtt{tgt}}, s'_{\mathtt{tgt}},\ (m_{\mathtt{tgt}}, s_{\mathtt{tgt}}) \overset{\tau}{\hookrightarrow}^{*} \overset{e}{\hookrightarrow} \overset{\tau}{\hookrightarrow}^{*} (m'_{\mathtt{tgt}}, s'_{\mathtt{tgt}}) \wedge$

11: $\boxed{\exists w' \sqsupseteq w,\ (m'_{\mathtt{src}}, m'_{\mathtt{tgt}}) \in \mathrm{mrel}(w')} \wedge$

12: $((m'_{\mathtt{src}}, s'_{\mathtt{src}}), (m'_{\mathtt{tgt}}, s'_{\mathtt{tgt}})) \in R(w')$

13: **end**

Figure II.1: Structured (or, open) simulations (simplified for presentation purposes)

`after_external` r s denotes the language state after the external function call at s, assuming the call returned a value r.

Using interaction semantics, CompComp defines *structured simulations* relating two open modules. Here we briefly review the key ideas behind them, which also occurred elsewhere, *e.g.*, in [10, 19, 14]. First, unlike the closed simulations above, structured simulations explicitly specify value and memory relations (evolving over time) because values and memory are shared with external modules. Specifically, such relations are defined using Kripke-style possible worlds, called *structured injections* (see Section 6.1 for more details), by giving (i) a future world relation $\sqsupseteq$ for which $w' \sqsupseteq w$ denotes that w' is a future world of w; and (ii) value and memory relations at each world w, denoted $\mathtt{vrel}(w)$ and $\mathtt{mrel}(w)$. Then, a structured simulation R gives a relation between machine states at each world w, denoted $R(w)$, and should satisfy the *open* simulation property (simplified for presentation purposes) given in Figure II.1.

Here the simulation involves rely-guarantee reasoning and is split into two cases: one for interactions with external modules and the other for internal steps (omitting two more cases for function start and end, for presentation purposes). Specifically, given any world w, related memories at w and machine states related by the simulation relation R at w (line 1), we check whether the

source state is invoking an external function or taking an internal step (line 2). In the former case (line 3), the target state should also be invoking an external function (line 4) and the invoked functions and arguments should be related by the value relation at the world w (line 5), which is a guarantee condition to the external modules. Then we assume that the invoked external functions proceed to a future world w' yielding related memories and related return values at w' (line 6), which is a rely condition from the external modules. Under the assumption, the machine states after the external calls should also be related by the simulation relation R at the future world w' (line 7). In the latter case (line 8), for any internal step from the source state (line 9), there should be corresponding internal steps from the target state (line 10). Then the resulting memories after the steps should be related at some future world w' (line 11), which is a guarantee condition to the external modules. Finally, the machine states after the steps should also be related by the simulation relation R at the future world w' (line 12).

At high level, this simulation property specifies that internal executions of the source and target modules should be related in lockstep satisfying the guarantee conditions to the external modules, assuming that the rely conditions from them hold after each external function call. Note that the rely and guarantee conditions on memory (at lines 6 and 11) are matched and also those on values (at lines 5 and 6) will be matched if we include the omitted cases for function start and end. This matching—in addition to the fact that the same rely/guarantee conditions are used globally (*i.e.*, for verification of every module)—is crucial for proving preservation of the simulation property after linking modules because otherwise what one module assumes about the other modules will not match with what the other modules guarantee.

To use structured simulations for compiler verification, CompComp proves the following three key properties, where we say a source module M simulates a target module M' if there exists a structured simulation that relates M and M':

- **(*Vertical Compositionality*)** If M simulates M', which simulates M'', then M simulates M''.
- **(*Horizontal Compositionality*)** If M_1 and M_2 simulate M_1' and M_2' respectively, then the linked source module $M_1 \oplus M_2$ simulates the linked target module $M_1' \oplus M_2'$.
- **(*Adequacy*)** If M simulates M', then $\texttt{Beh}(M) \supseteq \texttt{Beh}(M')$.

4 Problems

As discussed in the introduction, verification using structured simulations is significantly more costly than that using closed simulations. The reasons are twofold.

First, while closed simulations freely allow arbitrary memory relations—therefore CompCert uses three different kinds of memory relations to simply proofs—structured simulations only allow a single type of memory relations called *structured injections* due to horizontal compositionality. The reason is that, as discussed above, allowing different memory relations would introduce different rely/guarantee conditions thereby breaking simulation after linking (*i.e.*, horizontal compositionality) due to the mismatch between different rely/guarantee conditions.

Second, proving vertical compositionality for *open* simulations is in general very technical and involved [21, 19]. Indeed the proof for structured simulations is about 5,000 SLOC in Coq. Moreover, vertical compositionality also introduces unnecessary complexities in structured simulations of CompComp such as *effect annotations* and *closedness under restriction* [27].

To sum up, although it is quite straightforward to prove horizontal compositionality and adequacy for a single relation (*i.e.*, with the same rely/guarantee conditions), it is challenging to prove (*i*) vertical compositionality even for a single relation and (*ii*) horizontal compositionality between different relations (*i.e.*, with different rely/guarantee conditions).

5 Our Solution

Our solution is twofold. First, we develop a general and abstract theory, called *Refinement Under Self-related Contexts (RUSC)*, which is inspired by the standard notion of contextual refinement and the notion of self-related context from [27] (see Section 14 for comparison). Specifically, given a set of (arbitrary and independent) relations each of which is horizontally compositional and adequate, RUSC completes the relations by giving a super-relation (*i.e.*, including all of them) that is horizontally and vertically compositional and also adequate. Second, we prove that our version of structured simulations, called *open simulations*, with almost arbitrary memory relations are horizontally compositional and adequate, so that we can apply RUSC to open simulations with any chosen set of memory relations.

Theory of RUSC RUSC can be defined abstractly for any linking algebra, which consists of a set of modules, Module, with a notion of behavior[2], denoted $\texttt{Beh}(p)$ for $p \in \text{Module}$, a linking operation $\oplus$ between modules that is associative[3], and the identity (*i.e.*, empty) module $\texttt{id} \in \text{Module}$:

$$\begin{gathered}\oplus : \text{Module} \times \text{Module} \to \text{Module} \\ \forall p, q, r \in \text{Module},\ p \oplus (q \oplus r) = (p \oplus q) \oplus r \\ \forall p \in \text{Module},\ \texttt{id} \oplus p = p \oplus \texttt{id} = p\end{gathered}$$

Note that RUSC can be applied to interaction semantics because it allows linking between arbitrary modules sharing the same notions of value and memory (see Section 6.2 and Section 11 for details).

To define RUSC, let $\mathcal{R}$ be a set of module relations each of which is horizontally compositional and adequate: for any $R \in \mathcal{R}$ and $p, p', q, q' \in \text{Module}$,

$$\begin{aligned}(p, p'), (q, q') \in R &\implies (p \oplus q, p' \oplus q') \in R && (\textit{HorComp}) \\ (p, p') \in R &\implies \texttt{Beh}(p) \supseteq \texttt{Beh}(p') && (\textit{Adequacy})\end{aligned}$$

Then the RUSC relation for $\mathcal{R}$, denoted $\succcurlyeq_{\mathcal{R}}$, is defined as follows:

$$\begin{aligned}p \succcurlyeq_{\mathcal{R}} p' \quad &\textit{iff} \quad \forall c_1, c_2 \in \text{Self}(\mathcal{R}),\ \texttt{Beh}(c_1 \oplus p \oplus c_2) \supseteq \texttt{Beh}(c_1 \oplus p' \oplus c_2) \\ \text{Self}(\mathcal{R}) \quad &\stackrel{\text{def}}{=} \quad \{\, c \in \text{Module} \mid \forall R \in \mathcal{R},\ (c, c) \in R \,\}\end{aligned}$$

The definition is simple: p is RUSC-related to p' if the behaviors of p' refine those of p under arbitrary contexts that are related to themselves by every relation in $\mathcal{R}$.

Theorem 1 (Properties of RUSC)**.** The RUSC relation $\succcurlyeq_{\mathcal{R}}$ satisfies the following key properties.

$$\begin{aligned}&\text{(Inclusion)} && \forall R \in \mathcal{R}, && R \subseteq \succcurlyeq_{\mathcal{R}} \\ &\text{(Adequacy)} && \forall p, p' \in \text{Module}, && p \succcurlyeq_{\mathcal{R}} p' \implies \texttt{Beh}(p) \supseteq \texttt{Beh}(p') \\ &\text{(VerComp)} && \forall p, p', p'' \in \text{Module}, && p \succcurlyeq_{\mathcal{R}} p' \wedge p' \succcurlyeq_{\mathcal{R}} p'' \implies p \succcurlyeq_{\mathcal{R}} p'' \\ &\text{(HorComp)} && \forall p, p', q, q' \in \text{Self}(\mathcal{R}), && p \succcurlyeq_{\mathcal{R}} p' \wedge q \succcurlyeq_{\mathcal{R}} q' \implies p \oplus q \succcurlyeq_{\mathcal{R}} p' \oplus q' \\ &\text{(SelfComp)} && \forall p, q \in \text{Self}(\mathcal{R}), && p \oplus q \in \text{Self}(\mathcal{R})\end{aligned}$$

Note that horizontal compositionality holds only for self-related modules, which, however, is not a big deal in practice as we will discuss below.

[2]Behaviors just need to be defined for closed programs. Technically, we can give undefined behavior (UB) to open modules.

[3]Commutativity does not hold for linking of CompCert modules because changes in the order of global variables affect the initial memory after loading due to CompCert's deterministic memory allocation.

Proof. The proof of the theorem is simple. The inclusion $R \subseteq \succcurlyeq_{\mathcal{R}}$ trivially follows from the horizontal compositionality and adequacy of R. Adequacy of $\succcurlyeq_{\mathcal{R}}$ directly follows from the definition of $\succcurlyeq_{\mathcal{R}}$ by taking the empty context. Vertical compositionality (*i.e.*, transitivity) of $\succcurlyeq_{\mathcal{R}}$ holds trivially by definition. Horizontal compositionality, $p \oplus q \succcurlyeq_{\mathcal{R}} p' \oplus q'$, is proven in two steps using transitivity: (i) $p \oplus q \succcurlyeq_{\mathcal{R}} p' \oplus q$, which follows from the definition of $p \succcurlyeq_{\mathcal{R}} p'$ since $q \in \mathrm{Self}(\mathcal{R})$, and (ii) $p' \oplus q \succcurlyeq_{\mathcal{R}} p' \oplus q'$, which follows similarly since $q \succcurlyeq_{\mathcal{R}} q'$ and $p' \in \mathrm{Self}(\mathcal{R})$. Finally, self-relatedness is closed under composition because every relation in $\mathcal{R}$ is horizontally compositional. □

The reason why vertical compositionality is easily proven for RUSC is that we essentially prove it for *closed* programs by closing an open module with contexts. Indeed, the technical difficulties with vertical compositionality for open simulations arise from the openness: it is difficult to set up a setting properly with arbitrary future memories given after an external function call.

The reason why horizontal compositionality holds between different relations is interesting. Directly composing two simulations $(p, p') \in R_1$ and $(q, q') \in R_2$ with different relations R_1 and R_2 does not work in general. However, each simulation can be easily extended with identical contexts because a pair of identical modules usually respects any sensible relational principles. Therefore, we have $(p \oplus q, p' \oplus q) \in R_1$ and $(p' \oplus q, p' \oplus q') \in R_2$, which can be transitively composed by vertical compositionality just as discussed above.

To sum up, RUSC provides a general condition for composing different relational proofs: each proof just needs to be compatible with its context modules in terms of self-relatedness, not necessarily with their relational proofs.

Open Simulations Since we can obtain vertical and horizontal compositionality using RUSC, we can use open simulations with almost arbitrary memory relations. More specifically, we prove that open simulations with *any* Kripke-style memory/value relation satisfying certain minimal conditions (see Section 12.2 for details) are horizontally compositional and adequate. Since the required conditions are so minimal, they are satisfied by the three memory relations—memory identity, extension and injection—and also by a more powerful relation, called *memory injection with module-local invariants*. This new memory relation is needed to verify a new pass we added, called `Unreadglob`, which requires reasoning about module-local states enabled by *static* variables of C (see Section 9 for details).

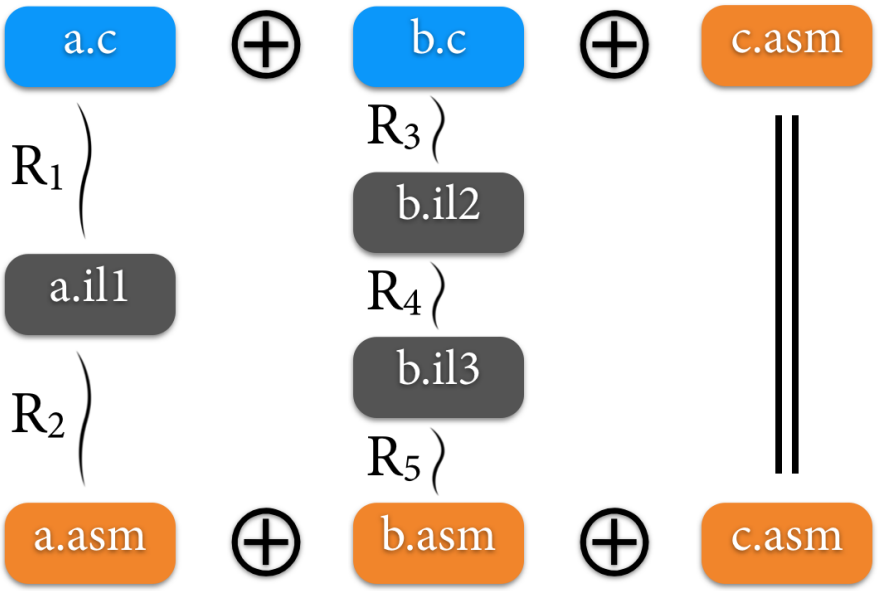

Figure II.2: A compiler verification scenario with RUSC

Note that unlike CompCert 2.1 on which CompComp is based, CompCert 3.5 implements a static analyzer performing value analysis, which is used by several passes. In order to support independent modular verification of such analyzers, we also parameterize open simulations with memory predicates—representing the analysis results of such analyzers—and prove their horizontal compositionality and adequacy (See Section 12 for details).

Applications We use RUSC in two situations: compiler and program verification.

First, we give an abstract example (presented in Figure II.2) for compiler verification. Suppose our source program is written in three modules, `a.c`, `b.c` and `c.asm`, and compiled via multiple passes: `a.c` $\to$ `a.il1` $\to$ `a.asm` and `b.c` $\to$ `b.il2` $\to$ `b.il3` $\to$ `b.asm`, each of which is verified using a different relation R_1 to R_5. Then as long as the *end modules*, `a.c`, `b.c`, `a.asm`, `b.asm`, `c.asm`, are self-related by the relations $R_1, \ldots, R_5$, using RUSC we can obtain the following behavioral refinement:

$$\texttt{Beh}(\texttt{a.c} \oplus \texttt{b.c} \oplus \texttt{c.asm}) \supseteq \texttt{Beh}(\texttt{a.asm} \oplus \texttt{b.asm} \oplus \texttt{c.asm})$$

The underlying reasoning is simple: for $\mathcal{R} = \{ R_1, \ldots, R_5 \}$, we get

- $\texttt{a.c} \succcurlyeq_{\mathcal{R}} \texttt{a.asm}$ and $\texttt{b.c} \succcurlyeq_{\mathcal{R}} \texttt{b.asm}$ by Inclusion and VerComp of Theorem 1;
- $\texttt{c.asm} \succcurlyeq_{\mathcal{R}} \texttt{c.asm}$ since $(\texttt{c.asm}, \texttt{c.asm}) \in R_1 \subseteq \succcurlyeq_{\mathcal{R}}$ by Inclusion of Theorem 1;
- $\texttt{a.c} \oplus \texttt{b.c} \oplus \texttt{c.asm} \succcurlyeq_{\mathcal{R}} \texttt{a.asm} \oplus \texttt{b.asm} \oplus \texttt{c.asm}$ by HorComp of Theorem 1;
- $\texttt{Beh}(\texttt{a.c} \oplus \texttt{b.c} \oplus \texttt{c.asm}) \supseteq \texttt{Beh}(\texttt{a.asm} \oplus \texttt{b.asm} \oplus \texttt{c.asm})$ by Adequacy of Theorem 1.

Note that we need to prove the self-relatedness only for the end modules because we only link those, not the intermediate ones like `a.il1`, `b.il2`, `c.il3`. Moreover, proving self-relatedness by a relation is typically straightforward as long as the relation is sensibly defined. Indeed, we could easily prove that *all* Clight[4] and assembly programs are self-related by all the relations used by CompCertM (*i.e.*, open simulations with memory identity, extension, and injection with or without module-local invariants).

Second, we demonstrate, via small but interesting examples (see Section 17), that our framework can be used to verify program modules against (open) mathematical specification modules, written in Coq's Gallina language. In the above example, for instance, we can prove

$$\texttt{a.spec} \succcurlyeq_{\mathcal{R}} \texttt{a.c} \quad \texttt{b.spec} \succcurlyeq_{\mathcal{R}} \texttt{b.c} \quad \texttt{c.spec} \succcurlyeq_{\mathcal{R}} \texttt{c.asm}$$
$$\texttt{abc.spec} \succcurlyeq_{\mathcal{R}} \texttt{a.spec} \oplus \texttt{b.spec} \oplus \texttt{c.spec}$$

and link them together with the compiler correctness results above to get

$$\texttt{Beh}(\texttt{abc.spec}) \supseteq \texttt{Beh}(\texttt{a.asm} \oplus \texttt{b.asm} \oplus \texttt{c.asm})$$

as long as the mathematical specification modules `a.spec`, `b.spec`, `c.spec`, `abc.spec` are in $\text{Self}(\mathcal{R})$, which is usually straightforward to prove.

Comparison to Contextual Refinement As one can easily see, RUSC refines the standard notion of contextual refinement: instead of quantifying over *all* contexts, RUSC quantifies over only *self-related* contexts. The main difference is that RUSC gives the notion of well-behaved context w.r.t. a given set of program relations (*i.e.*, reasoning principles) in terms of contexts self-related by them. This is particularly useful when not all contexts are well behaved. For example, in the interaction semantics allowing mathematical specification

[4]Clight is taken as the source language in most verification projects using CompCert such as VST [1], CertiKOS and even CompComp. However, we also prove behavioral refinement w.r.t. the C source language (see Section 10).

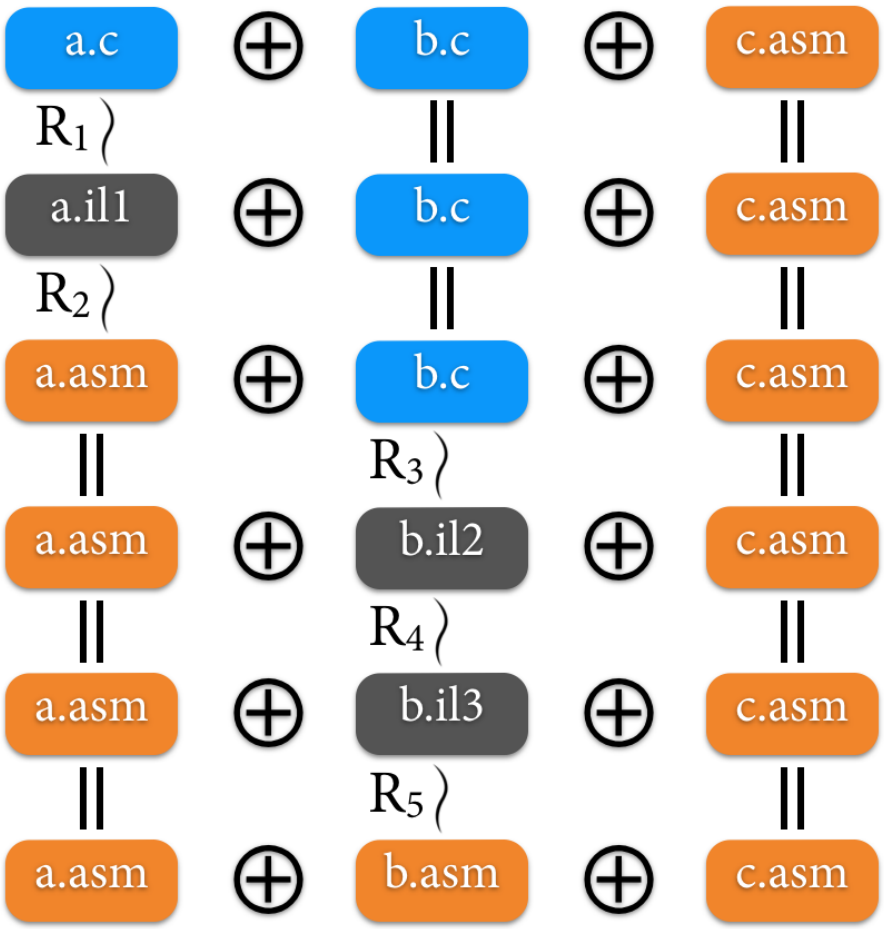

Figure II.3: An expanded version of the verification scenario described Figure II.2

modules as above, one can easily write a specification module that arbitrarily changes the whole memory including other modules' private memory. Under the presence of such ill-behaved contexts, the contextual refinement will end up being too strong preventing any reasoning about private memory such as functions' stack frames. On the other hand, RUSC w.r.t. a set of sensible relations will rule out such bad contexts and give us a sensible (better) relation.

Comparison to SepCompCert If we "unfold" the notion of RUSC in the verification scenario above (Figure II.2), what is happening under the hood is this: Figure II.3. Here, the key insight is that by inserting *dummy passes* – which does not modify anything – *only one* module is actually translated between

each row. Then, self-relatedness of *end modules* and *HorComp* and *Adequacy* of each relation R_i implies behavioral refinement between each row. For instance, between the first and second row:

- $(\texttt{a.c}, \texttt{a.il1}) \in R_1$ is given;
- $(\texttt{b.c}, \texttt{b.c}) \in R_1$ and $(\texttt{c.asm}, \texttt{c.asm}) \in R_1$ by self-relatedness of end modules;
- $(\texttt{a.c} \oplus \texttt{b.c} \oplus \texttt{c.asm},\ \texttt{a.il1} \oplus \texttt{b.c} \oplus \texttt{c.asm}) \in R_1$ by *HorComp* of R_1;
- $\texttt{Beh}(\texttt{a.c} \oplus \texttt{b.c} \oplus \texttt{c.asm}) \supseteq \texttt{Beh}(\texttt{a.il1} \oplus \texttt{b.c} \oplus \texttt{c.asm})$ by *Adequacy* of R_1.

Actually, the idea of inserting dummy passes has already been employed in SepCompCert's "Level B" correctness, but their technique is akin to contextual refinement, and thus RUSC can be seen as a generalization of their technique. Also, explicitly having a composable relation (RUSC) is both cognitively and technically useful. Other minor differences are that in their setting, they used the syntactic linking operator $\circ$ concatenating modules of the *same* language only, where our linking algebra allows language-independent linking. Also, due to this restriction, SepCompCert was able to use closed simulation while we use open simulation.

Chapter III

Compiler Verification

We first give background on CompCert and CompComp's interaction semantics (Section 6), discuss the problems with interaction semantics (Section 7) and present how we fixed it (Section 8). We show the flexibility of our framework by adding an advanced optimization (Section 9), and report our overall results in (Section 10). In Section 11 to 13, we flesh out formal details of our development. Finally, we discuss related works in Section 14.

6 Background

In this section, we briefly review CompCert's memory model and memory relations (Section 6.1) and interaction semantics of CompComp (Section 6.2).

6.1 CompCert's Memory Model and Memory Relations

Undefined Behavior UBs can be understood as erroneous (forbidden) operations so that compilers are licensed to translate them into arbitrary behaviors. A typical example of such an erroneous operation is a buffer overrun, which may spoil the stack frame and invalidate the compiler's basic assumptions. Note that compilers are not required to detect UB – it is impossible to detect it statically, and doing so dynamically will give too much overhead. Instead, programmers are obligated to avoid UB, and compilers can optimize, assuming the source

program is free of UB.

To capture those intuitions, UB is formally defined as a set of all possible behaviors. Recall that the semantics preservation (Section 3) property states that behaviors of the source program includes that of the target. By defining UB this way, compilers are indeed licensed to translate UB into arbitrary behaviors. Also, programmers are indeed obligated to avoid UB, because if UB is executed, anything can happen (*e.g.*, leaking private data).

CompCert Memory Model The *memory model* determines, for each memory operation, how the memory is changed and what are the return values.

The simplest way to define a memory model is to represent memory as a single large map, just as in hardware. This model is often called a *flat* (or *concrete*) memory model. Formally, the model is defined as follows.

$$\begin{aligned} \text{Mem} &\stackrel{\text{def}}{=} \texttt{uint32} \to \texttt{Option Val} \\ \text{Val} &\stackrel{\text{def}}{=} \texttt{uint32} \end{aligned}$$

Note that both pointer and integer values are represented as a 32-bit integer in Val. $\text{Mem}(p) = \texttt{Some}\ v$ means that the location p is allocated, and it contains a value v. On the other hand, $\text{Mem}(p) = \texttt{None}$ means it is not allocated yet.

However, the flat memory model does not properly model buffer overrun as UB, and thus it cannot validate essential compiler translations. Specifically, see the following example:

```
int main() {              int main {              int f() {
  int* x = malloc(4);       int* x = malloc(4);     int* y = malloc(4);
  x[0] = 42;                x[0] = 42;              y[1] = 43;
  f();                 -->  f();                  }
  out(x[0]);                out(42);
}                         }
```

Figure III.1: A counterexample showing the problem with the flat memory model

In this example, if the address of x and y+4 happens to be the same, the translation is invalid because the source prints 43 while the target prints 42. The root cause of the problem is that there is no way to distinguish two pointers x and y+4 where the former should be allowed to access x[0] while the latter should not.

To address this problem, CompCert uses a different memory model that is also called *logical* memory model. Conceptually, in this model memory consists of a finite set of blocks where each block is an array of finite size. It is formally defined as follows (simplified for presentation purpose):

$$
\begin{aligned}
\text{Mem} &\stackrel{\text{def}}{=} \{(m, nb) \in (\mathit{Block} \to \mathbb{Z} \to (\text{Perm} \times \text{Val})) \times \text{Block} \mid \\
&\qquad \forall\, b, o, v, m(b, o) = (\text{Valid}, v) \implies b < nb\} \\
\text{Block} &\stackrel{\text{def}}{=} \texttt{uint32} \\
\text{Val} &\stackrel{\text{def}}{=} \texttt{uint32} \uplus (\text{Block} \times \mathbb{Z}) \uplus \texttt{undef} \\
\text{Perm} &\stackrel{\text{def}}{=} \text{Valid} \uplus \text{Invalid}
\end{aligned}
$$

If a block's permission (Perm) is Valid, it is an allocated block and accessible. If it is Invalid, it is not allocated yet or already freed; thus, accessing it is considered UB. Values are either a 32-bit integer, a pointer composed of a block id and an offset inside it, or an `undef`value, which can be understood as an uninitialized value. In Mem, for a given address (b, o), m returns permission, and the value contained in that address. nb (which is an abbreviation of next block) always points to the smallest fresh block id. Therefore, whenever a memory block is allocated, it increments nb by one and returns $(nb, 0)$.

With a logical memory model, the above translation is valid. Note that x and y originate in different allocations, so they have different block id. Therefore, one cannot access x with y no matter the offset. Actually, accessing $y[1]$ executes UB because its permission is Invalid, so the semantic preservation holds trivially.

Memory Relations of CompCert As previously mentioned, CompCert uses three memory relations: identity, extension, and injection. The memory identity imposes that the source and target memories are identical; and the extension that the two memories contain identical block ids and each target block extends the corresponding source block with more space and any values in it at the end.

A memory injection injects a subset of the source blocks into target blocks without overlap. More precisely, a (selected) whole source block is injected into a single target block while allowing multiple source blocks to be injected into the same target block without overlap. This injection map specifies the *public* areas of the source and target memories and the correspondence between them. In other words, the corresponding addresses by the injection map are treated as *equivalent* (public) pointer values, so that at those corresponding addresses,

only equivalent[1] values (*i.e.*, equivalent non-pointer values or corresponding addresses) should be stored . All the areas that are not on the injection map are considered as *private* areas of the source and target memories.

Memory Relations of CompCertM CompCertM uses the original memory identity and extension of CompCert (Section 6.1) and mildly strengthens the original memory injection to reason about dynamically allocated local memory such as a function's stack frame for *open* modules, which can be compared to the structured injection of CompComp (Section 9). Moreover, we generalize it further to reason about statically allocated local memory such as static variables of C by allowing module-local invariants on those static variables (Section 9).

6.2 Interaction Semantics

We give a brief overview of interaction semantics of CompComp, which interactively executes modules equipped with their own independent module semantics. Each module semantics M provides a set of module states (also called *cores*) $\mathrm{State}(M)$ with the following operations:

- `init_core`: given a function f with arguments $\vec{v}$, gives the initial module state $s \in \mathrm{State}(M)$ executing the invoked function f with $\vec{v}$.
- `at_external`: given $s \in \mathrm{State}(M)$, checks if an external function f is called with arguments $\vec{v}$.
- `after_external`: given $s \in \mathrm{State}(M)$ where an external function is called, and a return value r, gives the module state s' after the function call returns r.
- `halted`: given $s \in \mathrm{State}(M)$, checks if the module execution is halted with a return value r.
- `corestep`: given $s \in \mathrm{State}(M)$ and memory m, takes a local step producing an event e and the next state s' with updated memory m'.

We explain how interaction semantics works using an example in Figure III.2, where the whole machine state consists of a memory, say m, and a stack of module states (called *core stack*), say $[s_2; s_1]$. Then, interaction semantics checks whether

[1]Technically speaking, CompCert allow more undefined values in the source because it proves refinement rather than equivalence between the source and target programs.

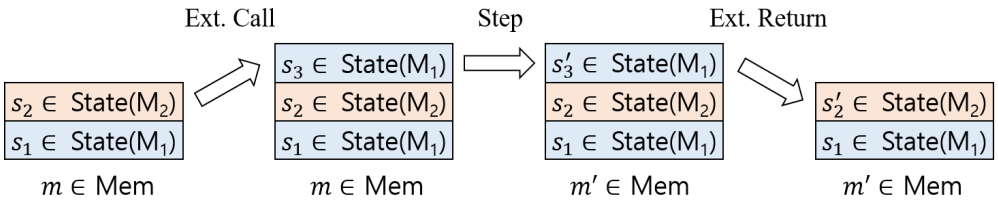

Figure III.2: An execution of interaction semantics

the stack-top module s_2 is invoking an external function using `at_external`, and if so, pushes the invoked module's initial state, say s_3, obtained by `init_core`. Note here that the same module M_1 can have multiple module states s_1 and s_3 in the stack. Then the new top module s_3 takes a local step to s_3' with updated memory m' according to its `corestep`, and if s_3' is a halted state with a return value r (checked with `halted`), the top module s_3' is popped and returned to the next module s_2, which is then updated to s_2' given by `after_external` with the return value r.

Finally, note that the language semantics of C, assembly and intermediate languages can be lifted to give a module semantics by defining `corestep` to be the same as the execution step of the language's semantics, and the other module operations to reflect the calling conventions. Note also that all language-specific resources (*i.e.*, other than the memory) such as the register-file of assembly reside inside the module state, and thus are duplicated at each invocation of a module.

7 Problems

The problems with the interaction semantics of CompComp are that it does not satisfy two adequacy properties. First, the adequacy w.r.t. C says that for any C modules $M_1, \ldots, M_n$, the behaviors of the linked program according to interaction semantics $\texttt{Beh}(M_1 \oplus \ldots \oplus M_n)$ should *be included in* those according to the physical semantics $\texttt{Beh}(M_1 \circ \ldots \circ M_n)$. The reason for failure was quite simple and we could easily fix it: unlike CompComp, we allow passing the `undef` value to an external module since the C semantics does so, while we turn ill-typed values into `undef` when they are passed to an external module.

Second, the failure of the adequacy w.r.t. assembly is more serious. Adequacy says that for any assembly modules $M_1, \ldots, M_n$, the behaviors of the linked program according to interaction semantics $\texttt{Beh}(M_1 \oplus \ldots \oplus M_n)$ should *include*

those according to the physical semantics $\texttt{Beh}(M_1 \circ \ldots \circ M_n)$. Note that the direction is opposite since assembly is the target language. As discussed before, the reason for failure is that the interaction semantics of CompComp does not have a mechanism to detect illegal interference and make it undefined behavior (UB).

8 Our Solution

We identify the sources of inadequacy w.r.t. assembly as violations of three assumptions made by standard compilers: two on the registers and one on the stack. We discuss why they cause problems with counterexamples and show how to semantically handle them without changing the underlying language semantics.

8.1 Assumptions on the Registers

The two problematic assumptions on the registers are that an invoked assembly function (*i*) should preserve the initial values of the callee-save registers, and (*ii*) should not access the memory via the leftover pointer values remaining in those registers that are not involved in passing meaningful information to the callee, which we henceforth call *non-info-passing* registers.

Counterexamples The example in Figure III.3 shows how violations of the two assumptions can invalidate correct compiler translations. The code in the left box (a) shows a standard translation of C code into assembly (written in pseudocode) performed by mainstream compilers like GCC and LLVM, where the accesses to the array `x` are translated into accesses via the register `%rbx` assuming that `%rbx` is set to contain the address of `x`. An important point here is that the compiler assumes that (*i*) the value of `%rbx` is unchanged across the function call `f()` since it is a callee-save register, and also (*ii*) the values in the array pointed to by `%rbx` are unchanged across `f()` since the array's addresses do not escape except via non-info-passing registers like `%rbx`. Therefore, the compiler expects that `out(*(%rbx))` in the target code correctly outputs `0`.

The right box (b) presents an example of handwritten assembly (written in pseudocode) for function `f` that violates the above two assumptions of the compiler. The code either increments `%rbx` by `4` or writes `1` to `*(%rbx)` depending on the result of call to `g`. Now if we link the assembly code in (a) and that in (b)

(a)

```
int main()   {          main:
  int* x = malloc(8);     ...
  x[0] = 0;               *(%rbx) = 0;
  x[1] = 1;               *(%rbx + 4) = 1;
  f();               -->  f();
  out(x[0]);              out(*(%rbx));
  ...                     ...
}
```

⊕

(b)

```
f:
  if (g(%rbx))
    %rbx = %rbx + 4;
  else
    *(%rbx) = 1;
```

Figure III.3: A counterexample showing the problem with the assumptions on the registers

together, one can easily see that `out(*(%rbx))` incorrectly outputs `1` instead of `0` in either case: in the former case, `%rbx` points to the second element of the array `x`, which contains `1`; in the latter case, the value of `*(%rbx)` is directly updated to `1`. Therefore, it makes sense to define those illegal behaviors of (b) as undefined behavior (UB).

Our Model We present our model making the illegal behaviors UBs in stages, explaining at each stage why naive models do not work.

First, in order to enforce the preservation of callee-save register values, we store the initial values of the callee-save registers at the `init-core` step of assembly modules; and check, at the `halted` step, whether the final values of those registers are equal to the stored initial values and if not, raise UB. Here, the question is, when a new core with a fresh register file is pushed into the core stack, what values should be set as initial values of the non-info-passing registers including all of the callee-save registers. Since the registers may contain arbitrary values in the physical assembly semantics, a natural choice would be to initially set them to contain the `undef` value, which is an abstract value representing all possible values. Indeed, this is the choice of CompComp. However, there is a serious problem. Since, for instance, `undef + 4` results in `undef`, checking

whether the final values of callee-save registers are equal to the initial values, *i.e.*, `undef`, is not sufficient. Specifically, the assembly code in (b) above does not raise UB in this new semantics in case `g(%rbx)` returns `1` because the initial and final values of `%rbx` are both `undef` and thus equal even though the callee-save register `%rbx` is incremented by `4` in the physical semantics.

Second, another natural solution would be to initially set the non-info-passing registers to nondeterministically contain arbitrary values including `undef`. Though this model is more flexible, it still has a problem. For instance, in the above example, to simulate the physical behaviors of the assembly function `f` in interaction semantics, one can set the initial value of `%rbx` to be either (*i*) `undef` (*i.e.*, a more abstract value than the physical one), or (*ii*) a pointer to the array `x` (*i.e.*, a value equivalent to the physical one): other values cannot be used since they are not refined by the value of `%rbx` in the target, which is required since the value is passed to an unknown function `g`. In the former case, if `g(%rbx)` returns `1`, we have the same problem with callee-save checking as shown above. In the latter case, if `g(%rbx)` returns `0`, the function `f` successfully updates the array `x` thereby invaliding the translation in (a) as illustrated above.

We solve this problem by further revising the second model: nondeterministically allocating an arbitrary number of *junk blocks* (*i.e.*, blocks of size zero) and then initializing the non-info-passing registers with arbitrary non-pointer values or *junk pointers* (*i.e.*, pointers to the junk blocks). Then we can simulate the physical behaviors by initializing each register r (*i*) with the same non-pointer value if the physical value of r is a non-pointer value; and (*ii*) otherwise with a fresh junk pointer. The high-level idea is that, like `undef`, a junk pointer is more abstract (*i.e.*, causing more UBs) than any pointer but, unlike `undef`, sufficiently distinguishable. For instance, in the previous example, if `g(%rbx)` returns `1`, the initial and final values of `%rbx` (*i.e.*, p and $p+4$ for a junk pointer p) are distinguished thereby raising UB by the callee-save checking; if `g(%rbx)` returns `0`, the memory access `*(%rbx) = 1` raises UB because `%rbx` points to a junk block of size zero.

Finally, note that introducing nondeterminism as above is not a showstopper thanks to the mixed simulation, as discussed in Section 8.3: we can do forward simulation everywhere except for the `init_core` step of assembly modules, where we should do backward simulation.

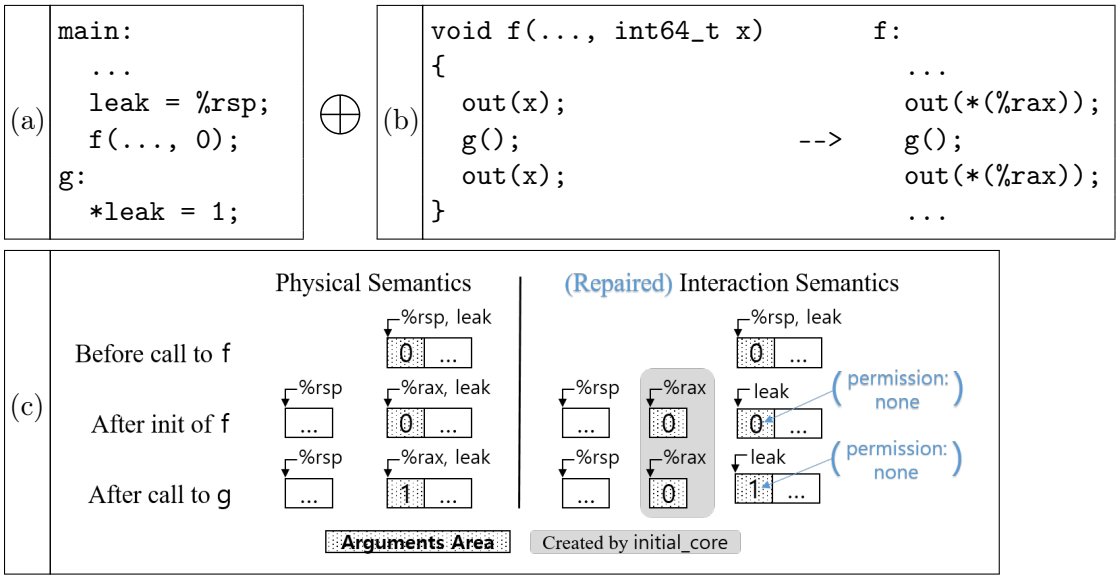

Figure III.4: A counterexample showing the problem with the assumption on the stack

8.2 Assumptions on the Stack

The problematic assumption on the stack is that the *outgoing arguments area* of a caller's stack (*i.e.*, where overflowing function arguments are stored) should be fully *owned* by the callee. In other words, the callee can assume that the arguments area is never modified by others unless its addresses are revealed to the public by the callee itself.

Counterexamples The example in Figure III.4 shows how violations of the assumption can invalidate correct compiler translations. The box (a) shows handwritten assembly code implementing two functions `main` and `g`; the box (b) shows a standard translation of C code into assembly essentially performed by `gcc -O0`; and the left-hand side (LHS) of the box (c) depicts the shape of the stack during execution in the physical semantics. The function `main` stores the address of the outgoing arguments area (*i.e.*, `%rsp` as depicted in LHS of (c)) in the global variable `leak` and invokes the function `f`, where the last argument `0` is stored in the arguments area of the stack. Then the function `f` makes three function calls, `out(x)`, `g()` and `out(x)`, where the argument `x` is directly read from the arguments area pointed to by `%rax` in the assembly, as depicted in LHS of (c), and `out(x)` outputs the read value. Finally, the function `g` updates the

arguments area pointed to by `leak` with `1`, as depicted in LHS of (c), between the two function calls `out(x)`.

An important point here is that the compiler assumes that the arguments area (*i.e.*, `%rax`) is unchanged across the function call `g()` since it is fully owned by `f`. Therefore, the compiler expects that both calls `out(*(%rax))` in the target code correctly output `0`. However, since the function `g` updates the arguments area with `1` via `leak`, the two calls incorrectly output `0` and `1`. We confirmed this incorrectness by compiling `f` with `gcc -O2`, which eliminates the second load `*(%rax)` by propagating the result of the first load across `g()` thereby outputting `0` twice.

Our Model In order to solve the problem, we have to distinguish accesses to the arguments area via the caller from those via the callee and define the former as UB. Though making such distinction is difficult in the physical semantics, fortunately it is already made in interaction semantics due to the language-independent design. For example, consider the interaction semantics of the above example, depicted in the right-hand-side (RHS) of Figure III.4 (c). The difference is that when the assembly function `f` is invoked, the initialization process (*i.e.*, `init_core`) of the module semantics newly constructs the arguments area of the stack from the given logical arguments in order to make an environment needed to execute the assembly function `f`. This is essentially needed because the caller may not be an assembly module so that it may not have its own stack at all. Then the callee sees the new arguments area created by `init_core` while the caller (in assembly) sees the original arguments area.

Although the original interaction semantics does not prevent access to the arguments area via the caller, we can easily fix it. We simply (*i*) turn off the access permission of the original arguments area in the `at_external` step of the caller module, and (*ii*) turn it back on in the `after_external` step. Note that the notion of permission already exists in the CompCert semantics, so that we do not need to strengthen it. In the above example again, the update by `g` will raise UB since the original argument area pointed to by `leak` has no access permission.

8.3 Mixed Simulation

While the target language of CompCert is deterministic (more precisely, the source is receptive and the target is determinate) thereby mostly using forward

simulations, the repaired interaction semantics of CompCertM is inherently nondeterministic to handle illegal interference from assembly modules (Section 8.1) thus preventing the use of forward simulation.

In order to recover the ability to use forward simulation in the occasional presence of nondeterminism, we adopt the idea of *mixed (forward-backward) simulation* from [19]. The key observation is that the requirement for using forward simulations (*i.e.*, determinism of the target) is a per-state property, not a per-language property: as long as a particular target machine state is *locally deterministic* (*i.e.*, its next state is unique), one can do forward simulation at that state. Based on this observation, mixed simulations selectively allow forward simulation when the target is locally deterministic, in addition to the default backward simulation. Specifically, we say that a relation R is a (closed) mixed simulation if for all $(ms_{\mathtt{src}}, ms_{\mathtt{tgt}}) \in R$,

1. $\forall e, ms'_{\mathtt{tgt}},\ ms_{\mathtt{tgt}} \overset{e}{\hookrightarrow} ms'_{\mathtt{tgt}} \implies$
   $\exists ms'_{\mathtt{src}},\ ms_{\mathtt{src}} \overset{\tau}{\hookrightarrow}^{*} \overset{e}{\hookrightarrow} \overset{\tau}{\hookrightarrow}^{*} ms'_{\mathtt{src}} \wedge (ms'_{\mathtt{src}}, ms'_{\mathtt{tgt}}) \in R$; or

2. $\forall e, ms'_{\mathtt{src}},\ ms_{\mathtt{src}} \overset{e}{\hookrightarrow} ms'_{\mathtt{src}} \implies$
   $\exists ms'_{\mathtt{tgt}},\ ms_{\mathtt{tgt}} \overset{\tau}{\circ\!\!\rightarrow}^{*} \overset{e}{\circ\!\!\rightarrow} \overset{\tau}{\circ\!\!\rightarrow}^{*} ms'_{\mathtt{tgt}} \wedge (ms'_{\mathtt{src}}, ms'_{\mathtt{tgt}}) \in R$

where $ms \overset{e}{\circ\!\!\rightarrow} ms'$ denotes that ms is locally deterministic and $ms \overset{e}{\hookrightarrow} ms'$.

Figure III.5 visualizes this formulation of mixed simulation, where solid and dotted arrows represent universally and existentially quantified steps, respectively, and double circles represent locally deterministic target states. In this figure, since the first three target machine states are deterministic, we can do forward simulation as shown in the figure; then, since the following target state is nondeterministic, we should do backward simulation as shown in the figure.

Note that the repaired interaction semantics is nondeterministic only at the initial step of a module invocation, so that we can do forward simulation everywhere else using mixed simulations.

In order to support CompCert's condition for forward simulation, we also add the following to the above formulation of mixed simulation:

3. or, $ms_{\mathtt{src}}$ is receptive and
   $\forall e, ms'_{\mathtt{src}},\ ms_{\mathtt{src}} \overset{e}{\hookrightarrow} ms'_{\mathtt{src}} \implies$
   $\exists ms'_{\mathtt{tgt}},\ ms_{\mathtt{tgt}} \overset{\tau}{\bullet\!\!\rightarrow}^{*} \overset{e}{\bullet\!\!\rightarrow} \overset{\tau}{\bullet\!\!\rightarrow}^{*} ms'_{\mathtt{tgt}} \wedge (ms'_{\mathtt{src}}, ms'_{\mathtt{tgt}}) \in R$
   where $ms \overset{e}{\bullet\!\!\rightarrow} ms'$ denotes that ms is locally determinate and $ms \overset{e}{\hookrightarrow} ms'$.

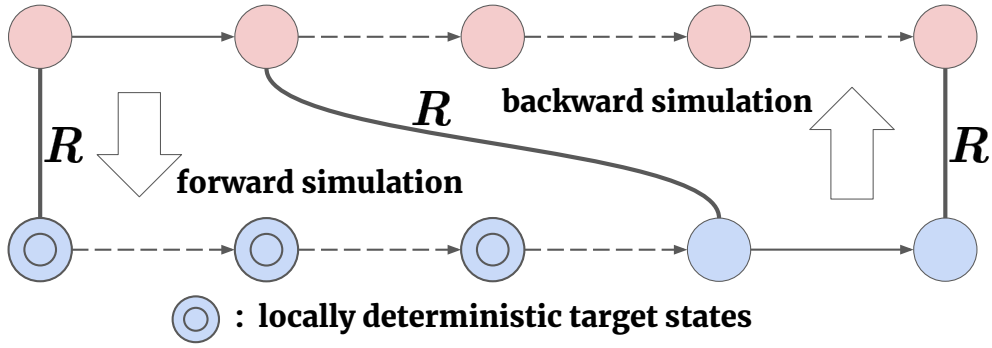

Figure III.5: A visualized example of mixed simulations

Also we apply this mechanism of mixed simulation to our open simulations.

9 Advanced Optimizations with Module-Local Invariants

To demonstrate the flexibility of our framework – allowing arbitrary memory relations – we added a new optimization, `Unreadglob`, whose verification requires a new memory relation. We flesh out the details in the following order: first we present *enriched memory injection*, a mildly strengthened version of CompCert's original injection (Section 9), then the new memory relation (Section 9), and finally `Unreadglob` optimization (Section 9).

Enriched Memory Injection For open modules, reasoning about dynamically allocated local memory such as a function's stack frame requires to strengthen the original memory injection due to the presence of unknown modules. The reason is because when reasoning about a module M, we have to assume that an unknown function invoked by M does not change the dynamic local memory of M and also guarantee that a function of M invoked by an unknown module does not change the caller's dynamic local memory.

For this purpose, CompComp introduces *structured injections* that enrich the original memory injections with ownership information (*i.e.*, whether owned by the current module or others) for all memory blocks including public ones. Using them, structured simulations impose fine-grained invariants subject to the ownership information and a concrete leakage protocol based on reachability from pointers.

Unlike CompComp, CompCertM generalizes open simulations and memory

injections in a more abstract way following [5, 10].

First, we generalize the external call case of the open simulation in Figure II.1 by allowing *private transitions*, denoted $\sqsupseteq_{\texttt{prv}}$, as follows (in red color):

$$\begin{array}{ll}
5: & \exists w' \sqsupseteq_{\texttt{prv}} w,\ (f_{\texttt{src}}, f_{\texttt{tgt}}) \in \texttt{vrel}(w') \wedge (\vec{v}_{\texttt{src}}, \vec{v}_{\texttt{tgt}}) \in \overrightarrow{\texttt{vrel}(w')} \wedge \\
6: & \forall w'' \sqsupseteq w',\ \forall (m'_{\texttt{src}}, m'_{\texttt{tgt}}) \in \texttt{mrel}(w''),\ \forall (r_{\texttt{src}}, r_{\texttt{tgt}}) \in \texttt{vrel}(w''), \\
7: & \exists w''' \sqsupseteq_{\texttt{prv}} w'',\ w''' \sqsupseteq w \wedge \\
& ((m'_{\texttt{src}}, \texttt{after_external}\ r_{\texttt{src}}\ s_{\texttt{src}}), (m'_{\texttt{tgt}}, \texttt{after_external}\ r_{\texttt{tgt}}\ s_{\texttt{tgt}})) \in R(w''')
\end{array}$$

Though private transitions are allowed before and after an external function call (*i.e.*, $w' \sqsupseteq_{\texttt{prv}} w$ and $w''' \sqsupseteq_{\texttt{prv}} w''$), the overall transition should be *public* (*i.e.*, $w''' \sqsupseteq w$) assuming the external call also makes a public transition (*i.e.*, $w'' \sqsupseteq w'$).[2]

Second, we extend memory injections to specify others' dynamic local memories in the source and target that should be unchanged by the current module. Specifically, an (enriched) memory injection $(\iota, m^{\texttt{prv}}_{\texttt{src}}, m^{\texttt{prv}}_{\texttt{tgt}})$ consists of an original memory injection ι mapping the source public blocks into target blocks; and additionally a private (*i.e.*, dynamic local) memory of the source $m^{\texttt{prv}}_{\texttt{src}}$ and that of the target $m^{\texttt{prv}}_{\texttt{tgt}}$ where $m^{\texttt{prv}}_{\texttt{src}}$ and $m^{\texttt{prv}}_{\texttt{tgt}}$ should be disjoint from the public memories specified by ι. Then, private transitions from $(\iota, m^{\texttt{prv}}_{\texttt{src}}, m^{\texttt{prv}}_{\texttt{tgt}})$ to $(\iota', m'^{\texttt{prv}}_{\texttt{src}}, m'^{\texttt{prv}}_{\texttt{tgt}})$ only require that ι' should extend ι, while public transitions additionally require that private memories should be unchanged (*i.e.*, $m^{\texttt{prv}}_{\texttt{src}} = m'^{\texttt{prv}}_{\texttt{src}}$ and $m^{\texttt{prv}}_{\texttt{tgt}} = m'^{\texttt{prv}}_{\texttt{tgt}}$). Note that all the areas of the source and target memories that are not on $m^{\texttt{prv}}_{\texttt{src}}$, $m^{\texttt{prv}}_{\texttt{tgt}}$ or the injection map ι are considered as *private* (*i.e.*, dynamic local) memory of the current module.

```
   int f() {              int f() {
1:   int a0;                int a[2];
2:   reg a1 = 0;   -->      a[1] = 0;
3:   g(&a0);                g(&a[0]);
4:   return a1;             return a[1];
   }                      }
```

To show how it works, we give an example mimicking register spilling in the presence of address-taken stack variables. Consider the transformation on the right, where in the source a memory block for `a0` and a function-local register for `a1` are allocated and the address of `a0` escapes to `g`, while in the target a single

[2] We only allow private transitions just before and after external calls for simplicity. See Section 14 for comparison with [5, 10].

block for both `a[0]` and `a[1]` is allocated and the address of the block escapes to `g`. Here `a0` can be seen as an address-taken stack variable and `a1` a spilled register. The key difference is that, in the source, `a1` cannot be accessed by `g` since it is a function-local register while, in the target, `a[1]` can be accessed via the address of `a[0]`.

We now show how the target `f` simulates the source `f` by logically protecting `a[1]` from `g`. Though we give an informal description here to help understanding, the formal definition of an open simulation R can be easily derived from the description. At line `1`, any world w_0 and memories $(m_{\mathtt{src}}, m_{\mathtt{tgt}})$ related at w_0 are given. We take a step to line `2` by extending $w_0.\iota$ (*i.e.*, the public injection of w_0) to map `a0` to `a[0]`, say w_1, which is a public transition. At line `2`, we take a step to line `3` without changing the world w_1. At line `3`, we first make a private transition from w_1 to w_2 by extending $w_1.m^{\mathtt{prv}}_{\mathtt{tgt}}$ to include the memory chunk `a[1]` $= 0$. Then we assume that `g` makes a public transition from w_2 to w_3 returning any memories related at w_3. Thanks to $w_2.m^{\mathtt{prv}}_{\mathtt{tgt}} = w_3.m^{\mathtt{prv}}_{\mathtt{tgt}}$, we know that the chunk `a[1]` $= 0$ remains the same. Then we make a private transition from w_3 to w_4 by dropping the chunk `a[1]` $= 0$ from $w_3.m^{\mathtt{prv}}_{\mathtt{tgt}}$. Since $w_4.m^{\mathtt{prv}}_{\mathtt{tgt}} = w_1.m^{\mathtt{prv}}_{\mathtt{tgt}}$, we have a public transition from w_1 to w_4. Finally, at line `4`, we know that both the register `a1` and the memory-allocated variable `a[1]` contain `0` and thus the same value `0` is returned.

It is important to note that the (others') private memories $w.m^{\mathtt{prv}}_{\mathtt{src}}$ and $w.m^{\mathtt{prv}}_{\mathtt{tgt}}$ of a memory injection w are preserved as long as a function accesses (i) the memory via public addresses, or (ii) its own private memory. In the former case, since a public block of the source is fully injected into a block of the target, whenever a pointer offset goes beyond the public area mapped by the injection $w.\iota$, the source program accesses an unallocated area thereby raising UB. In the example above, if `g` in the target accesses `*(&a[0]+1)`, then in the source it accesses `*(&a0+1)`, which raises UB. In the latter case, since the function's own private memory is disjoint from all the memories specified by w, accessing it does not affect w. In the example above, at line `2` in the target, the assignment `a[1] = 0` preserves $w_1.m^{\mathtt{prv}}_{\mathtt{tgt}}$ (and also the target public memory of w_1) because we know that the current private memory `a[1]` is disjoint from the area specified by w_1 by construction.

Also note that any part of the public memories cannot be converted to a private one since the injection map is only extended at each step; and any part of the others' private memories (*i.e.*, $m^{\mathtt{prv}}_{\mathtt{src}}$ and $m^{\mathtt{prv}}_{\mathtt{tgt}}$) cannot be converted to

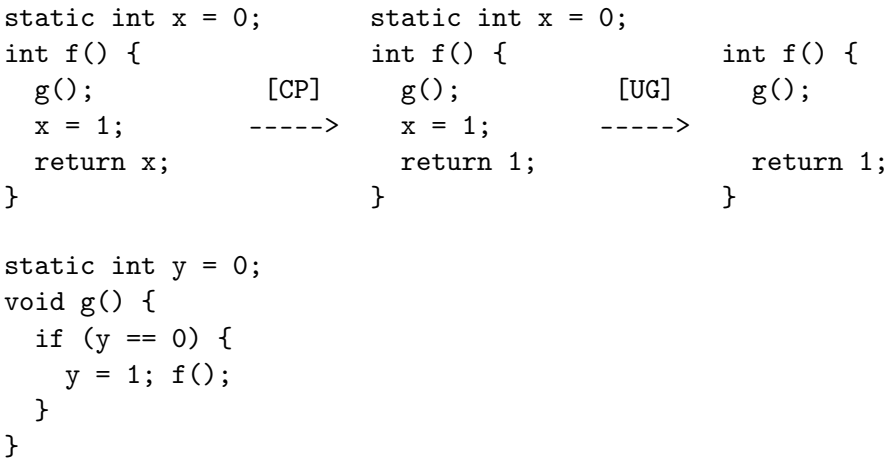

Figure III.6: An example of `Unreadglob` optimization

the current module's private one since all *proper* steps (*i.e.*, local steps or steps across an external call) only allow public transitions (*i.e.*, preserving $m^{\mathtt{prv}}_{\mathtt{src}}$ and $m^{\mathtt{prv}}_{\mathtt{tgt}}$).

Memory Injection with Module-Local Invariants For open modules, reasoning about statically allocated local memory such as static variables of C requires a further generalization. The problem is that when an open module M invokes an unknown function f, one cannot assume that the static memory of M is unchanged during the call because f may call back a function from M, which may change the static memory. However, since the static memory is only accessible to the known functions in M, one can find a certain invariant on the static memory by analyzing all the functions of M and expect that an external call preserves the invariant although the static memory can be changed. Enabling such reasoning is simple: CompCertM just adds another component in a memory injection w that globally imposes a given invariant on selected static variables disjoint from $w.m^{\mathtt{prv}}_{\mathtt{src}}$, $w.m^{\mathtt{prv}}_{\mathtt{tgt}}$ and $w.\iota$. We give examples using module-local invariants in Section 9 and 17.

Unreadglob Optimization We developed a new optimization `Unreadglob` eliminating all unread static variables and instructions writing to them. Figure III.6 shows an example optimization, where (*i*) the first program is optimized to the second one by constant propagation (CP) replacing `return x` by `return 1`; and (*ii*) the second one is optimized to the third one by `Unreadglob` (UG)

eliminating the unread static variable `x` and the command `x = 1`. It is important to note that across the function call `g()`, the static variable `x` may be updated from `0` to `1` because the function `g` can indirectly update it by calling `f` as shown in the fourth program in Figure III.6.

In verification of the optimization `UG` above, we have to use memory injections w with module-local invariants introduced in Section 9. The reason is that the static variable `x` in the source cannot reside (i) in the injection map $w.\iota$ since `x` does not exist in the target; or (ii) in the source private $w.m^{\mathtt{prv}}_{\mathtt{src}}$ since `x` can be modified during the external call `g()`. To verify `UG` above, we can impose the trivial invariant `Top` on the eliminated static variable `x`, meaning that `x` can be modified arbitrarily, which is sufficient because `x` is unread.

Note that CompCertX may be able to verify `Unreadglob` using memory injections because it assumes no mutual dependency among modules, so that no static variables can be accessed via external function calls, unlike the above example with mutual recursion.

10 CompCertM

Based on the theories we presented so far, we develop CompCertM, an extension of CompCert with the repaired interaction semantics and open simulations to support multi-language linking. We state CompCertM's compositional correctness results (Section 10.1) and evaluate its verification efforts (Section 10.2). CompCertM currently supports the x86 backend only. We do not currently see any technical problem with supporting other architectures.

10.1 Compositional Correctness

CompCertM uses open simulations with three parameters: memory relations, symbol relations and memory predicates (see Section 12.3 for details). It supports (i) the memory relations discussed in Section 6.1: identity, extension and (enriched) injections with no or any given module-local invariant; (ii) two symbol relations: one for keeping identical symbols in the source and target and the other for allowing elimination of global variables in the target (only allowed for memory injections), needed for `Unusedglob` and `Unreadglob`; (iii) two memory predicates: one for no analysis and the other for the value analysis of CompCert.

Let $\mathcal{R}$ be the set of open simulations with all possible parameters. To apply RUSC, we prove that the CompCertM compiler $\mathcal{C}$ transforms the source module

with a series of passes that are independently verified using open simulations in $\mathcal{R}$.

Lemma 2 (Pass Correctness)**.** For any Clight module S and Asm module T, if $\mathcal{C}(S) = T$, then there exist intermediate modules $M_0, M_1, \cdots, M_n$ such that:

1. $M_0 = S$ and $M_n = T$; and
2. $\forall i \in [0, n),\ \exists R \in \mathcal{R},\ (M_i, M_{i+1}) \in R$.

We also prove all Clight and Asm modules are self-related.

Lemma 3 (Self-Relatedness)**.** For any Clight or Asm module M, we have $M \in \mathrm{Self}(\mathcal{R})$.

Note that since we define illegal interference from Asm (*i.e.*, causing different behaviors in the source and target) as undefined behaviors (UBs) as shown in Section 8, every Asm module can be self-related.

From Lemmas 2 and 3, the RUSC relation for the compiler follows.

Theorem 4 (Modular Correctness)**.** For any Clight module S and Asm module T, if $\mathcal{C}(S) = T$:

$$S \succcurlyeq_{\mathcal{R}} T \quad \text{with} \quad S, T \in \mathrm{Self}(\mathcal{R}) \ .$$

This theorem provides a truly compositional correctness thanks to the compositionality of RUSC (Theorem 1): the relation can be freely (*i.e.*, vertically or horizontally) composed with any verification using RUSC including that against mathematical specifications. As an example, the following compositional correctness follows.

Corollary 5 (Compositional Correctness 1)**.** Let $(S_1, T_1), \ldots, (S_n, T_n)$ be pairs of source and target modules. If each pair is either compiled (*i.e.*, $\mathcal{C}(S_i) = T_i$ with S_i Clight and T_i Asm), or a self-related context (*i.e.*, $S_i = T_i \in \mathrm{Self}(\mathcal{R})$), then

$$\mathtt{Beh}(S_1 \oplus \cdots \oplus S_n) \supseteq \mathtt{Beh}(T_1 \oplus \cdots \oplus T_n) \ .$$

This correctness theorem is compositional in the sense that behavior is refined in the presence of any self-related contexts such as arbitrary Clight and Asm modules (Lemma 3).

Note that Clight, not CompCert C, is the source language in the above theorems. One of the reasons is that Clight is the source language for most verification frameworks based on CompCert, such as VST [1], CompComp, and

CompCertX. More importantly, we found that CompCert C is incompatible with memory injections. Specifically, CompCert C imposes a strict alignment requirement on memory blocks of size zero, which, however, is not preserved by memory injections. In other words, CompCert C modules are not always self-related by memory injections.[3]

Supporting CompCert C However, we can still prove a compositional correctness (not modular correctness as in Theorem 4) for CompCert C following SepCompCert's *Level A* technique [15], which exploits the fact that all CompCert C modules are transformed to Clight modules by the same two passes. Specifically, the first pass is verified using an open simulation with the memory identity and the second pass with memory injections, as done in the original CompCert. Then the following lemma follows from horizontal compositionality and adequacy of open simulations (with memory identity and injection) and transitivity of behavioral refinement.

Lemma 6 (ClightGen Correctness)**.** Let $(S_1, T_1), \ldots, (S_n, T_n)$ be pairs of source and target modules. If each pair is either translated (*i.e.*, $\mathrm{ClightGen}(S_i) = T_i$ with S_i CompCert C and T_i Clight), or a self-related context (*i.e.*, $S_i = T_i \in \mathrm{Self}(\mathcal{R})$), then

$$\mathtt{Beh}(S_1 \oplus \cdots \oplus S_n) \supseteq \mathtt{Beh}(T_1 \oplus \cdots \oplus T_n) \ .$$

By composing Corollary 5, Lemma 6 and Lemma 3, we have the following theorem.

Theorem 7 (Compositional Correctness 2)**.** Let $(S_1, T_1), \ldots, (S_n, T_n)$ be pairs of source and target modules. If each pair is either compiled (*i.e.*, $\mathcal{C}(S_i) = T_i$ with S_i CompCert C or Clight and T_i Asm), or a self-related context (*i.e.*, $S_i = T_i \in \mathrm{Self}(\mathcal{R})$), then

$$\mathtt{Beh}(S_1 \oplus \cdots \oplus S_n) \supseteq \mathtt{Beh}(T_1 \oplus \cdots \oplus T_n) \ .$$

Adequacy w.r.t. Physical Semantics We show that the repaired interaction semantics is adequate w.r.t. the physical semantics of CompCert, where the former uses the language-independent linking $\oplus$ and the latter the syntactic linking $\circ$ concatenating modules of the same language.

[3]This problem would be solved if one strengthens memory injections with more strict alignment requirements.

We prove that the physical semantics refines the repaired interaction semantics for Asm modules using a closed simulation of CompCert with memory injections.

Theorem 8 (Adequacy w.r.t. Assembly)**.** Let $M_1, \cdots, M_n$ be Asm modules. We have:

$$\texttt{Beh}(M_1 \oplus \ldots \oplus M_n) \supseteq \texttt{Beh}(M_1 \circ \ldots \circ M_n) \ .$$

This theorem allows us to carry verification results on the interaction semantics such as Theorem 7 down to CompCert's Asm semantics with syntactic linking.

Conversely, we prove that the repaired interaction semantics refines the physical semantics for CompCert C modules using a closed simulation of CompCert with memory identity. This result is useful because we want to allow separate compilation (of C modules) on the compiler side, and on the program verification side, we want to hide complexities from inter-module steps.

Theorem 9 (Adequacy w.r.t C)**.** Let $M_1, \cdots, M_n$ be well-typed CompCert C modules. We have:

$$\texttt{Beh}(M_1 \circ \ldots \circ M_n) \supseteq \texttt{Beh}(M_1 \oplus \ldots \oplus M_n) \ .$$

In some sense, the Theorems 7 to 9 together forms a strong stress-test for a language-independent linking, and our results show strong evidence that our repaired interaction semantics is indeed adequate (in a literal sense). Specifically, if one of the three desiderata is missing, it is trivial to find language-independent linking satisfying the others. Without Theorem 8, one can define interaction semantics to always execute UB; then, the other theorems become trivial. Without Theorem 9, one can define the behavior of interaction semantics to an empty set. Without Theorem 7, one can define $\oplus \stackrel{\text{def}}{=} \circ$.

Interestingly, by composing Theorems 7 to 9, we obtain the same separate compilation correctness result of SepCompCert [15]:

Corollary 10 (Separate Compilation Correctness)**.** Let $S_1, \ldots, S_n$ be CompCert C modules and $T_1, \ldots, T_n$ be Asm modules. If $\mathcal{C}(S_i) = T_i$ for each i, we have:

$$\texttt{Beh}(S_1 \circ \cdots \circ S_n) \supseteq \texttt{Beh}(T_1 \circ \cdots \circ T_n) \ .$$

10.2 Evaluation of Verification Efforts

To demonstrate that CompCertM is lightweight, we compare significant lines of code (SLOC) of CompCertM, CompComp, and CompCertX with those of their

Table III.1: SLOC of CompCertM and related works — compared to its baseline CompCert, respectively

Portion	CompCert 3.5	CompCertR 3.5	CompCertM pack	CompCert 2.1	CompComp
Pass Proofs	34,376	35,893 (+4.41%)	4,923(+14.32%)	21,215	52,140 (+145.77%)
The Rest	85,617	87,965 (+2.74%)	25,558(+29.85%)	59,365	107,910 (+81.77%)
Total	119,993	123,858 (+3.22%)	30,481(+25.40%)	80,580	160,050 (+98.62%)

Portion	CompCert 3.0	CompCertX
Pass Proofs	26,466	30,572 (+15.51%)
The Rest	82,312	121,532 (+47.65%)
Total	108,778	152,104 (+39.83%)

Table III.2: Breakdown of CompCertM pack

Portion	SLOC
Proofs about Intermodule Steps	4,923
Interaction Semantics/Properties	1,940
Language Semantics/Properties	1,701
Self Simulations	5,593
CompCert Metatheory Extension	4,688
CompCertM Metatheory	7,656
Mixed Simulation	1,090
Adequacy w.r.t. Asm	2,890

Table III.3: SLOC of additional developments

Portion	`Unreadglob` 3.5	`Unreadglob` pack	Adequacy w.r.t. C
Pass Proofs	1,842	338	-
The Rest	260	1,933	4,044
Total	2,102	2,271	4,044

baseline CompCert versions 3.5, 2.1, and 3.0, respectively. Overall, CompCertM adds less code to CompCert than CompComp and CompCertX do, and in particular significantly less code than CompComp for the proofs of compiler passes.[4] Also note that CompCertR uses the enriched memory injections of Section 9 instead of the original memory injections in order to give reusable main lemmas for both closed and open simulations. Since CompCertR's pass proofs are only 4.41% larger than CompCert's, the overhead due to handling the private memory components of enriched memory injections is, roughly speaking, at most 4.41%.

Table III.1 summarizes the comparison. For each compiler (*i.e.*, each column), the rows report SLOC for the proofs of all compiler passes (Pass Proofs), the rest of the development (The Rest), and their summation (Total). Note that CompCertM is split into CompCertR and CompCertM pack, for which the former is our refactoring of CompCert and the latter is an additional package to

[4]Note that CompComp allows horizontal compositionality between any intermediate languages (ILs) while CompCertM only between Clight and Asm since self-relatedness is proven only for the two. Though practically unnecessary, supporting linking between arbitrary ILs in CompCertM would increase SLOC to prove self-relatedness for the other ILs.

support multi-language linking. We counted SLOC reported by `coqwc`.[5] When counting SLOC, we excluded the following code for fair comparison: (*i*) code for other architectures than x86 because all three projects support only x86; (*ii*) code for the parser and type checker introduced in later versions of CompCert; and (*iii*) code for ClightGen, which is not supported by both CompCertX and CompComp. We also excluded CompComp's legacy proofs for the original compiler correctness. We used the latest development branches for the three projects.[6]

Table III.2 analyzes the 30,481 SLOC for CompCertM pack. The pass proofs consist of 4,923 SLOC for reasoning about intermodule steps, which is sometimes nontrivial since they perform the logical instrumentation presented in Section 8. Note that CompCertR provides proofs for intramodule steps as main lemmas, which are reused in CompCertM. The rest consists of 1,940 SLOC for the repaired interaction semantics and its properties; 1,701 SLOC for properties of each language such as determinism and receptiveness; 5,576 SLOC for self-relatedness (Lemma 3); 4,687 SLOC for extending the metatheory of CompCert; 7,569 SLOC for open simulations and other metatheory for CompCertM; 1,090 SLOC for mixed simulation; and 2,890 SLOC for adequacy w.r.t. assembly (Theorem 8).

Table III.3 shows SLOC for the new optimization pass. Note that `Unreadglob` 3.5 adds the optimization to CompCertR proving closed simulation and `Unreadglob` pack to CompCertM proving open simulation, which reuses the proof of `Unreadglob` 3.5 for intramodule steps.

11 Formal Semantics

In this section we give a few interesting details of formal semantics: the loading of interaction semantics (Section 11.1) and a few tweaks we made for module semantics (Section 11.2).

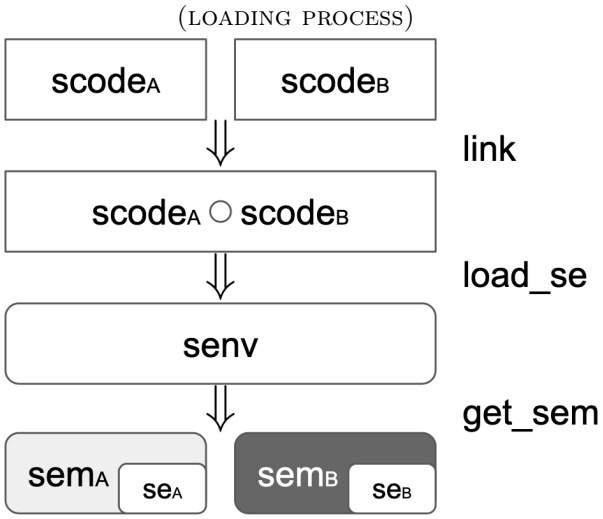

Figure III.7: Loading in Interaction Semantics

11.1 Loading in Interaction Semantics

Loading the initial states of multiple modules requires an interesting coordination of the modules, especially in the presence of module-local static variables. In essence, we should disallow accesses to a static variable from other modules than the defining one. For this, the loading of modules $M_1, \cdots, M_n$ proceeds as follows, which is illustrated for two modules in Figure III.7.

First, each module has *symbol code*, which consists of symbols (*i.e.*, global variables and functions) and their signatures (*e.g.*, `x: int`, `f: void(int)`). For each i, let M_i.`scode` be the symbol code of M_i. Crucially, symbol codes have the same type even if their modules are written in different languages.

Second, since symbol codes have the same type, we can calculate the physical linking $sc = M_1.\texttt{scode} \circ \cdots \circ M_n.\texttt{scode}$ of the symbol codes of modules. Now sc is the symbol code for entire program consisting of all the symbols and signatures. The physical linking is defined in [15].

Third, we load sc to get the initial memory *mem* (by load_mem) and the program's global *symbol environment se* (by load_se), which is the run-time information of symbols (*e.g.*, `x` points to `0x700` and `f` points to `0x800`). This

[5]Concretely, we counted "spec" and "proof" lines reported by `coqwc`. Because we use a different criteria for line numbers, they are different from those reported in prior work [27, 7, 28].

[6]Development as of November 8, 2019, available at: https://github.com/snu-sf/compcertr, https://github.com/snu-sf/compcertm, https://github.com/PrincetonUniversity/compcomp, https://github.com/DeepSpec/dsss17/tree/master/CAL

loading process follows the original CompCert's.

Fourth, we initialize module semantics for each module M_i with the program's global symbol environment se. In particular, we calculate M_i's local environment, which contains information of *only* those symbols defined in the module. Crucially, this prevents the other modules from accessing the static variables of M_i. Note that CompComp does not have local environments because it does not support static variables.

Finally, the initial memory and module semantics form the initial state for interaction semantics.

11.2 Module Semantics

We briefly discuss the notions of module and module semantics presented in Figure III.8. To support loading described in Section 11.1, a module M consists of M`.scode`, which is its symbol code, and M`.get_sem`, which returns a module semantics given a program's symbol environment. The local environment `senv` of the module semantics should coincide with the global environment restricted on M`.scode`.

The module semantics of CompCertM is slightly more general than that presented in Section 6.

- A module semantics has a symbol environment `senv` that determines whether a symbol belongs to the module or not.
- `init_core` is defined as a predicate rather than a function in order to allow such nondeterminism introduced in Section 8.
- Module operations other than `corestep` (denoted here $\hookrightarrow$) can also change the memory, which is needed to turn on and off the access permission of the arguments area as discussed in Section 8.
- Module semantics supports not only C-style but also assembly-style calling convention in the sense of CompCertX, where the former just passes argument and return values between the caller and callee while the latter the whole register file. Like CompCertX, only assembly functions are allowed to make assembly-style calls.

(MODULE)
$M \in \text{Module} = \{(\texttt{scode}, \texttt{get\_sem}) \in (\text{Scode} \times (\text{Senv} \to \text{ModSem})) \mid \forall se, \texttt{get\_sem}(se).\texttt{senv} = se|_{\texttt{scode}}\}$

(MODULE SEMANTICS)
$sem \in \text{ModSem} =$
$\{\, \texttt{state} \in \text{Set}, \texttt{senv} \in \text{Senv}, \hookrightarrow \in \mathcal{P}((\text{Mem} \times \texttt{state}) \times \text{Event} \times (\text{Mem} \times \texttt{state})),$
$\texttt{init\_core} \in \text{CallData} \to \mathcal{P}(\text{Mem} \times \texttt{state}),$
$\texttt{at\_external} \in (\text{Mem} \times \texttt{state}) \to \texttt{Option}\ \text{CallData},$
$\texttt{after\_external} \in (\texttt{state} \times \text{RetData}) \to \texttt{Option}\ (\text{Mem} \times \texttt{state}),$
$\texttt{halted} \in (\text{Mem} \times \texttt{state}) \to \texttt{Option}\ \text{RetData} \mid$
$\{ms \mid \exists e, ms',\ ms \overset{e}{\hookrightarrow} ms'\}, \{ms \mid \exists c, \texttt{at\_external}(ms) = \texttt{Some}\ c\}, \{ms \mid \exists r, \texttt{halted}(ms) = \texttt{Some}\ r\}$ disjoint $\}$
$\text{CallData} \overset{\text{def}}{=} \{(\texttt{m}, \texttt{f}, \texttt{vs}) \in (\text{Mem} \times \text{Val} \times \overrightarrow{\text{Val}})\} \uplus \{(\texttt{m}, \texttt{f}, \texttt{rs}) \in (\text{Mem} \times \text{Val} \times (\text{Reg} \to \text{Val}))\}$
$\text{RetData} \overset{\text{def}}{=} \{(\texttt{m}, \texttt{v}) \in (\text{Mem} \times \text{Val})\} \qquad \uplus \{(\texttt{m}, \texttt{rs}) \in (\text{Mem} \times (\text{Reg} \to \text{Val}))\}$

Figure III.8: Module and Module Semantics

12 Formalization of Verification Techniques

Now we present the formalization of our verification techniques. We parameterize the notion of open simulation presented in Section 3 with three parameters: memory relations, symbol relations, and memory predicates. We present formal details about mixed simulation (Section 12.1), the three parameters (Section 12.2), the parameterized open simulations (Section 12.3), and their horizontal compositionality and adequacy theorems (Section 12.4). Finally, we present some interesting instances for the three parameters (Section 12.5).

12.1 Mixed Simulation

In this section, we flesh out technical details about mixed simulation. Recall that our mixed simulation technique supports three modes; (1) backward, (2) forward with locally deterministic target states, and (3) forward with locally receptive source states and locally determinate target states. Technically, modes (2) and (3) are implemented as an instance of a more general mode (4) that subsumes the two. Therefore, we first define that "general" mode and then explain the others.

Intuitively, mode (2) imposes a strong restriction on the target and no restriction on the source, whereas mode (3) imposes a moderate restriction on both source and target. In mode (4), we introduce a parameter ST that controls

the amount of restriction imposed on the source and target.

$$\begin{aligned}\mathrm{ST} \in \mathrm{SimilarTraces} = \{ \sim \in \texttt{Option}\ \mathrm{Event} \times \texttt{Option}\ \mathrm{Event} \mid \\ (\forall e,\ \texttt{None} \sim e \implies e = \texttt{None}) \wedge (\forall e,\ e \sim \texttt{None} \implies e = \texttt{None})\}\end{aligned}$$

Then, depending on the parameter ST, we define the notion of locally receptive and locally determinate as follows.

$$\begin{aligned}&\mathrm{receptive\_at}(s : \texttt{state}) \stackrel{\text{def}}{=} \forall e_1, s_1, e_2, (s \overset{e_1}{\hookrightarrow} s_1 \wedge e_1 \sim e_2) \implies \exists s_2, s \overset{e_2}{\hookrightarrow} s_2 \\ &\mathrm{determinate\_at}(s : \texttt{state}) \stackrel{\text{def}}{=} \forall e_1, s_1, e_2, s_2, (s \overset{e_1}{\hookrightarrow} s_1 \wedge s \overset{e_2}{\hookrightarrow} s_2) \implies \\ &\qquad\qquad (e_1 \sim e_2 \wedge (e_1 = e_2 \implies s_1 = s_2))\end{aligned}$$

Finally, mode (4) is defined as follows:

4. There exists ST such that $\mathrm{receptive\_at}(ms_{\texttt{src}})$ holds and
   $\forall e, ms'_{\texttt{src}},\ ms_{\texttt{src}} \overset{e}{\hookrightarrow} ms'_{\texttt{src}} \implies$
   $\exists ms'_{\texttt{tgt}},\ ms_{\texttt{tgt}} \overset{\tau}{\bullet\!\!\rightarrow}{}^{*} \overset{e}{\bullet\!\!\rightarrow} \overset{\tau}{\bullet\!\!\rightarrow}{}^{*} ms'_{\texttt{tgt}} \wedge (ms'_{\texttt{src}}, ms'_{\texttt{tgt}}) \in R$
   where $ms \overset{e}{\bullet\!\!\rightarrow} ms'$ denotes that $\mathrm{determinate\_at}(ms)$ holds and $ms \overset{e}{\hookrightarrow} ms'$.

Proof. The proof is basically the same with forward to backward simulation proof in CompCert except that ST is parameterized. If the $ms_{\texttt{src}}$ cannot take any step, it is undefined behavior, so the proof is over. Otherwise, we have e_1 such that $ms_{\texttt{src}} \overset{e_1}{\hookrightarrow} ms'_{\texttt{src}}$. By applying mode (4) proof with e_1 and $ms'_{\texttt{src}}$, we have (τ steps omitted for brevity) $ms_{\texttt{tgt}} \overset{e_1}{\hookrightarrow} ms'_{\texttt{tgt}} \wedge (ms'_{\texttt{src}}, ms'_{\texttt{tgt}}) \in R$. Now, it suffices[7] to prove this:

$$\forall e_2, ms''_{\texttt{tgt}}, ms_{\texttt{tgt}} \overset{e_2}{\hookrightarrow} ms''_{\texttt{tgt}} \implies \exists ms'_{\texttt{src}}, ms_{\texttt{src}} \overset{e_2}{\hookrightarrow} ms'_{\texttt{src}} \wedge (ms'_{\texttt{src}}, ms''_{\texttt{tgt}}) \in R.$$

By determinate_at, we have $(e_1 \sim e_2 \wedge (e_1 = e_2 \implies ms'_{\texttt{tgt}} = ms''_{\texttt{tgt}}))$. If $e_1 = e_2$, we have $ms'_{\texttt{tgt}} = ms''_{\texttt{tgt}}$. We finish the proof by instantiating $\exists ms'_{\texttt{src}}$ with $ms'_{\texttt{src}}$. Otherwise, we still have $e_1 \sim e_2$. By receptive_at, we have $ms''_{\texttt{src}}$ such that $ms_{\texttt{src}} \overset{e_2}{\hookrightarrow} ms''_{\texttt{src}}$. Then, by applying mode (4) proof with e_2 and $ms''_{\texttt{src}}$, we get $ms'''_{\texttt{tgt}}$ such that $ms_{\texttt{tgt}} \overset{e_1}{\hookrightarrow} ms'''_{\texttt{tgt}} \wedge (ms''_{\texttt{src}}, ms'''_{\texttt{tgt}}) \in R$. By applying determinate_at with $e_2, ms''_{\texttt{tgt}}, e_2, ms'''_{\texttt{tgt}}$, we get $(e_2 \sim e_2 \wedge (e_2 = e_2 \implies ms''_{\texttt{tgt}} = ms'''_{\texttt{tgt}}))$. Hence, we have $ms''_{\texttt{tgt}} = ms'''_{\texttt{tgt}}$, and by simple rewriting, $(ms''_{\texttt{src}}, ms''_{\texttt{tgt}}) \in R$. We finish the proof by instantiating $\exists ms'_{\texttt{src}}$ with $ms''_{\texttt{src}}$. □

[7] We omit the details about coinductive reasoning here.

We get modes (2) and (3) by instantiating $\sim$ of (4) with proper instances; the former with equality, and the latter with a certain relation called "match_traces" from CompCert. Note that the above proof outline does not use conditions $(\forall e,\ \texttt{None} \sim e \implies e = \texttt{None}) \wedge (\forall e,\ e \sim \texttt{None} \implies e = \texttt{None})$ in ST. These are mainly added for technical reasons and might not be essential.

12.2 Parameters for Open Simulations

Figure III.9 presents the sets of three parameters for open simulations: the set of memory relations MR, the set of symbol relations SR, and the set of memory predicates MP.

Memory Relation The first parameter ranges over Kripke-style memory/-value relations in MR. Following [5, 10], we model the evolution of memory relations using *possible worlds* and *private and public transitions* over the worlds. Note that this parameter will be instantiated with the three memory relations used in CompCert—namely memory identity, extension, and injection—and the memory injection with module-local invariants we introduced.

A memory relation in MR consists of (i) a set $\texttt{t}$ of possible worlds; (ii) *public* and *private* transition relations $\sqsubseteq$ and $\sqsubseteq_{\texttt{prv}}$ over the worlds; and (iii) for each world $w \in \texttt{t}$, memory relation $\texttt{mrel}(w)$ and value relation $\texttt{vrel}(w)$. A world w represents an invariant on the memory, which can evolve over time according to the public/private transition relations, as we discussed in Section 6.1. There are four natural well-formedness conditions, which are self-explanatory. We can also straightforwardly extend the value/memory relation to relations on CallData and RetData, denoted $\succsim_w$.

Symbol Relation The second parameter ranges over symbol relations in SR that relate information about global symbols (*e.g.*, which block each global variable points to) in the source and target. This parameter is needed to verify optimizations like `Unusedglob`, `Unreadglob` that remove unnecessary static variables thereby having non-identical symbol information in the source and target.

A symbol relation in SR consists of (i) a set $\texttt{t}$ of symbol relation states; (ii) an extension relation $\sqsubseteq$ on the states; (iii) for each state d, a (compile-time) symbol code relation $\texttt{screl}(d)$; and (iv) for each state d and world $w \in \text{MR}.\texttt{t}$, (run-time) symbol environment relation $\texttt{serel}(d, w)$. There are seven well-formedness

(MEMORY RELATION)

$$\begin{aligned}
&\mathrm{MR} \in \mathrm{MemRel} = \\
&\quad \{\, (\mathtt{t}, \sqsubseteq, \sqsubseteq_{\mathrm{prv}}, \mathtt{mrel}, \mathtt{vrel}) \in (\mathrm{Set} \times \mathcal{P}(\mathtt{t} \times \mathtt{t}) \times \mathcal{P}(\mathtt{t} \times \mathtt{t}) \times (\mathtt{t} \to \mathcal{P}(\mathrm{Mem} \times \mathrm{Mem})) \times (\mathtt{t} \to \mathcal{P}(\mathrm{Val} \times \mathrm{Val}))) \mid \\
&\qquad (\sqsubseteq \text{ is preorder}) \ \wedge \ (\sqsubseteq \subseteq \sqsubseteq_{\mathrm{prv}}) \ \wedge \ (\forall w, w',\ w \sqsubseteq w' \implies \mathtt{vrel}(w) \subseteq \mathtt{vrel}(w')) \wedge \\
&\qquad (\forall w, i,\ (\mathrm{Vint}\ i, v_{\mathtt{tgt}}) \in \mathtt{vrel}(w) \implies v_{\mathtt{tgt}} = \mathrm{Vint}\ i)\, \} \\
&c_{\mathtt{src}} \succsim_w c_{\mathtt{tgt}} \stackrel{\mathrm{def}}{=} (c_{\mathtt{src}}.\mathtt{m}, c_{\mathtt{tgt}}.\mathtt{m}) \in \mathtt{mrel}(w) \wedge (c_{\mathtt{src}}.\mathtt{f}, c_{\mathtt{tgt}}.\mathtt{f}) \in \mathtt{vrel}(w) \wedge (c_{\mathtt{src}}.\mathtt{vs}, c_{\mathtt{tgt}}.\mathtt{vs}) \in \overrightarrow{\mathtt{vrel}(w)} \wedge \\
&\qquad\qquad (c_{\mathtt{src}}.\mathtt{rs}, c_{\mathtt{tgt}}.\mathtt{rs}) \in \overrightarrow{\mathtt{vrel}(w)} \\
&r_{\mathtt{src}} \succsim_w r_{\mathtt{tgt}} \stackrel{\mathrm{def}}{=} (r_{\mathtt{src}}.\mathtt{m}, r_{\mathtt{tgt}}.\mathtt{m}) \in \mathtt{mrel}(w) \wedge (r_{\mathtt{src}}.\mathtt{v}, r_{\mathtt{tgt}}.\mathtt{v}) \in \mathtt{vrel}(w) \wedge (r_{\mathtt{src}}.\mathtt{rs}, r_{\mathtt{tgt}}.\mathtt{rs}) \in \overrightarrow{\mathtt{vrel}(w)}
\end{aligned}$$

(SYMBOL RELATION)

$$\begin{aligned}
&\mathrm{SR} \in \mathrm{SymbRel} = \\
&\quad \{(\mathtt{t}, \sqsubseteq, \mathtt{screl}, \mathtt{serel}) \in (\mathrm{Set} \times \mathcal{P}(\mathtt{t} \times \mathtt{t}) \times (\mathtt{t} \to \mathcal{P}(\mathrm{Scode} \times \mathrm{Scode})) \times (\mathtt{t} \to \mathrm{MR}.\mathtt{t} \to \mathcal{P}(\mathrm{Senv} \times \mathrm{Senv}))) \mid \\
&\qquad (1)\ \sqsubseteq \text{ is preorder} \\
&\qquad (2)\ \forall sc_{\mathtt{src}}, sc'_{\mathtt{src}}, sc''_{\mathtt{src}}, sc_{\mathtt{tgt}}, sc'_{\mathtt{tgt}}, sc''_{\mathtt{tgt}},\ sc''_{\mathtt{src}} = sc_{\mathtt{src}} \circ sc'_{\mathtt{src}} \wedge sc''_{\mathtt{tgt}} = sc_{\mathtt{tgt}} \circ sc'_{\mathtt{tgt}} \implies \\
&\qquad\quad \forall d, d',\ (sc_{\mathtt{src}}, sc_{\mathtt{tgt}}) \in \mathtt{screl}(d) \wedge (sc'_{\mathtt{src}}\ sc'_{\mathtt{tgt}}) \in \mathtt{screl}(d') \implies \\
&\qquad\quad \exists d'',\ (sc''_{\mathtt{src}}, sc''_{\mathtt{tgt}}) \in \mathtt{screl}(d'') \wedge d \sqsubseteq d'' \wedge d' \sqsubseteq d'' \\
&\qquad (3)\ \forall sc_{\mathtt{src}}, sc_{\mathtt{tgt}}, d,\ (sc_{\mathtt{src}}, sc_{\mathtt{tgt}}) \in \mathtt{screl}(d) \implies \\
&\qquad\quad \exists w,\ (\mathrm{load_mem}(sc_{\mathtt{src}}), \mathrm{load_mem}(sc_{\mathtt{tgt}})) \in \mathtt{mrel}(w) \ \wedge \ (\mathrm{load_se}(sc_{\mathtt{src}}), \mathrm{load_se}(sc_{\mathtt{tgt}})) \in \mathtt{serel}(d, w) \\
&\qquad (4)\ \forall d, w, w',\ w \sqsubseteq_{\mathrm{prv}} w' \implies \mathtt{serel}(d, w) \subseteq \mathtt{serel}(d, w') \\
&\qquad (5)\ \forall d, w, se_{\mathtt{src}}, se_{\mathtt{tgt}},\ (se_{\mathtt{src}}, se_{\mathtt{tgt}}) \in \mathtt{serel}(d, w) \implies se_{\mathtt{src}}.\mathtt{pubs} = se_{\mathtt{tgt}}.\mathtt{pubs} \wedge \\
&\qquad\quad \forall (v_{\mathtt{src}}, v_{\mathtt{tgt}}) \in \mathrm{MR}.\mathtt{vrel}(w),\ v_{\mathtt{src}} \in \mathrm{ftns}(se_{\mathtt{src}}) \implies v_{\mathtt{tgt}} \in \mathrm{ftns}(se_{\mathtt{tgt}}) \\
&\qquad (6)\ \forall d, d', w, sc_{\mathtt{src}}, sc_{\mathtt{tgt}}, se_{\mathtt{src}}, se_{\mathtt{tgt}},\ d \sqsubseteq d' \wedge (sc_{\mathtt{src}}, sc_{\mathtt{tgt}}) \in \mathtt{screl}(d) \wedge (se_{\mathtt{src}}, se_{\mathtt{tgt}}) \in \mathtt{serel}(d', w) \implies \\
&\qquad\quad (se_{\mathtt{src}}|_{sc_{\mathtt{src}}}, se_{\mathtt{tgt}}|_{sc_{\mathtt{tgt}}}) \in \mathtt{serel}(d, w) \\
&\qquad (7)\ \forall d, w, se_{\mathtt{src}}, se_{\mathtt{tgt}}, c_{\mathtt{src}}, c_{\mathtt{tgt}},\ (se_{\mathtt{src}}, se_{\mathtt{tgt}}) \in \mathtt{serel}(d, w) \wedge c_{\mathtt{src}} \succsim_w c_{\mathtt{tgt}} \implies \\
&\qquad\quad \forall e, r_{\mathtt{src}},\ \text{external_call}\ se_{\mathtt{src}}\ c_{\mathtt{src}}\ e\ r_{\mathtt{src}} \implies \exists r_{\mathtt{tgt}},\ \text{external_call}\ se_{\mathtt{tgt}}\ c_{\mathtt{tgt}}\ e\ r_{\mathtt{tgt}} \wedge \exists w' \sqsupseteq w,\ r_{\mathtt{src}} \succsim_{w'} r_{\mathtt{tgt}}\, \}
\end{aligned}$$

(MEMORY PREDICATE)

$$\begin{aligned}
&\mathrm{MP} \in \mathrm{MemPred} = \\
&\quad \{\, (\mathtt{t}, \sqsubseteq, \sqsubseteq_{\mathrm{prv}}, \mathtt{mpred}, \mathtt{vpred}, \mathtt{sepred}) \in (\mathrm{Set} \times \mathcal{P}(\mathtt{t} \times \mathtt{t}) \times \mathcal{P}(\mathtt{t} \times \mathtt{t}) \times (\mathtt{t} \to \mathcal{P}(\mathrm{Mem})) \times (\mathtt{t} \to \mathcal{P}(\mathrm{Val})) \times (\mathtt{t} \to \mathcal{P}(\mathrm{Senv}))) \mid \\
&\qquad (\sqsubseteq \text{ is preorder}) \ \wedge \ (\sqsubseteq \subseteq \sqsubseteq_{\mathrm{prv}}) \ \wedge \ (\forall u, u',\ u \sqsubseteq u' \implies \mathtt{vpred}(u) \subseteq \mathtt{vpred}(u')) \wedge \\
&\qquad (\mathtt{sepred} \text{ should satisfy the unary version of } \mathtt{serel}\text{'s conditions where SR.}\sqsubseteq \text{ and } \mathtt{screl} \text{ are the total relations})\, \} \\
&\mathtt{cpred}(u) \stackrel{\mathrm{def}}{=} \{c \in \mathrm{CallData} \mid c.\mathtt{m} \in \mathtt{mpred}(u) \wedge c.\mathtt{f} \in \mathtt{vpred}(u) \wedge c.\mathtt{vs} \in \overrightarrow{\mathtt{vpred}(u)} \wedge c.\mathtt{rs} \in \overrightarrow{\mathtt{vpred}(u)}\} \\
&\mathtt{rpred}(u) \stackrel{\mathrm{def}}{=} \{r \in \mathrm{RetData} \mid r.\mathtt{m} \in \mathtt{mpred}(u) \wedge r.\mathtt{v} \in \mathtt{vpred}(u) \wedge r.\mathtt{rs} \in \overrightarrow{\mathtt{vpred}(u)}\}
\end{aligned}$$

Figure III.9: Three parameters for open simulations

conditions: (1) the extension relation $\sqsubseteq$ is transitive and reflexive; (2) `screl` is closed under the syntactic linking; (3) if symbol codes are related by `screl`, then the initial memories and symbol environments loaded by load_mem and load_se are related by `mrel` and `serel`, respectively; (4) `serel` is monotone w.r.t. private transitions; (5) for symbol environments related by `serel`, their public symbols are identical and their functions have the same signatures; (6) `serel` is compatible with $\sqsubseteq$: for $d \sqsubseteq d'$, $\mathtt{serel}(d')$ restricted on $\mathtt{screl}(d)$ should be in $\mathtt{serel}(d)$; and (7) the memory and symbol relations should be compatible

with CompCert's axiom about system calls (*i.e.*, external_call).

Memory Predicate The third parameter ranges over Kripke-style memory predicates in MP, which are needed to modularly verify CompCert's analysis engines such as value analysis (see Section 12.3). MP is essentially a unary version of MR combined with SR where SR.$\sqsubseteq$ and `screl` are taken as the total relations (*i.e.*, relating everything): it consists of (*i*) the set `t` of possible worlds; (*ii*) public and private transition relations $\sqsubseteq$ and $\sqsubseteq_{\texttt{prv}}$ over the worlds, respectively; and (*iii*) for each world $w \in \texttt{t}$, a memory predicate mpred(w), a value predicate vpred(w), and a symbol environment predicate sepred(w). The well-formedness conditions are self-explanatory.

12.3 Open Simulations with Parameters

Figure III.10 presents our parameterized open simulations, which are given in the form of forward simulation for simplicity though they are actually in the form of mixed simulation presented in Section 8.3. In this section, we omit MR, SR, and MP when clear from context (*e.g.*, `vrel`(w) for MR.`vrel`(w)). Also, $\boxed{\text{R}}$ and $\boxed{\text{G}}$ means rely and guarantee conditions for the external modules.

Simulation of Machine States A relation *match_states* on machine states is an *open simulation* if all related states either (*i*) transition to related states, (*ii*) invoke related external calls (hence the name "open" simulation), or (*iii*) halt with related return values and memories. Specifically, given source and target module semantics $sem_{\texttt{src}}$, $sem_{\texttt{tgt}}$ and a (source) soundness predicate *sound_state* (discussed later), the relation *match_states* over worlds is an open simulation if the relatedness of $ms_{\texttt{src}}$ and $ms_{\texttt{tgt}}$ at a world w with the soundness of $ms_{\texttt{src}}$ implies one of the followings.

- (STEP) The source and target states transition to related states. Specifically:

line 1: the source machine state takes intramodule steps, and

line 2: if the source machine state transitions to a next state emitting an event e,

line 3: then the target machine state is able to transition to a next state emitting the same event e, possibly with additional silent transitions, and

(SIM:STATES)

$match\_states \in \mathrm{open\_sim}(sem_{\mathrm{src}}, sem_{\mathrm{tgt}}, sound\_state) \stackrel{\mathrm{def}}{=}$

$\forall w, \forall((m_{\mathrm{src}}, s_{\mathrm{src}}), (m_{\mathrm{tgt}}, s_{\mathrm{tgt}})) \in match\_states(w), (\exists u,\ (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sound\_state(u)) \implies$

(STEP) $sem_{\mathrm{src}}.\mathtt{at\_external}(m_{\mathrm{src}}, s_{\mathrm{src}}) = \mathtt{None} \wedge sem_{\mathrm{src}}.\mathtt{halted}(m_{\mathrm{src}}, s_{\mathrm{src}}) = \mathtt{None} \wedge$
$\forall e, m'_{\mathrm{src}}, s'_{\mathrm{src}},\ (m_{\mathrm{src}}, s_{\mathrm{src}}) \overset{e}{\hookrightarrow} (m'_{\mathrm{src}}, s'_{\mathrm{src}}) \implies$
$\exists m'_{\mathrm{tgt}}, s'_{\mathrm{tgt}},\ (m_{\mathrm{tgt}}, s_{\mathrm{tgt}}) \overset{\tau}{\hookrightarrow}{}^{*} \overset{e}{\hookrightarrow} \overset{\tau}{\hookrightarrow}{}^{*} (m'_{\mathrm{tgt}}, s'_{\mathrm{tgt}}) \wedge$
$\exists w' \sqsupseteq w,\ ((m'_{\mathrm{src}}, s'_{\mathrm{src}}), (m'_{\mathrm{tgt}}, s'_{\mathrm{tgt}})) \in match\_states(w')$

$\vee$ (CALL) $\exists w' \sqsupseteq_{\mathrm{prv}} w,\ \exists c_{\mathrm{src}}, c_{\mathrm{tgt}},\ c_{\mathrm{src}} \succsim_{w'} c_{\mathrm{tgt}} \wedge$
$sem_{\mathrm{src}}.\mathtt{at\_external}(m_{\mathrm{src}}, s_{\mathrm{src}}) = \mathtt{Some}\ c_{\mathrm{src}} \wedge sem_{\mathrm{tgt}}.\mathtt{at\_external}(m_{\mathrm{tgt}}, s_{\mathrm{tgt}}) = \mathtt{Some}\ c_{\mathrm{tgt}} \wedge$
$\forall w'' \sqsupseteq w', \forall r_{\mathrm{src}}, r_{\mathrm{tgt}}, r_{\mathrm{src}} \succsim_{w''} r_{\mathrm{tgt}} \implies$
$\forall m'_{\mathrm{src}}, s'_{\mathrm{src}},\ sem_{\mathrm{src}}.\mathtt{after\_external}(s_{\mathrm{src}}, r_{\mathrm{src}}) = \mathtt{Some}\ (m'_{\mathrm{src}}, s'_{\mathrm{src}}) \implies$
$\exists m'_{\mathrm{tgt}}, s'_{\mathrm{tgt}},\ sem_{\mathrm{tgt}}.\mathtt{after\_external}(s_{\mathrm{tgt}}, r_{\mathrm{tgt}}) = \mathtt{Some}\ (m'_{\mathrm{tgt}}, s'_{\mathrm{tgt}}) \wedge$
$\exists w''' \sqsupseteq_{\mathrm{prv}} w'',\ w''' \sqsupseteq w \wedge ((m'_{\mathrm{src}}, s'_{\mathrm{src}}), (m'_{\mathrm{tgt}}, s'_{\mathrm{tgt}})) \in match\_states(w''')$

$\vee$ (RET) $\exists w' \sqsupseteq w, \exists r_{\mathrm{src}}, r_{\mathrm{tgt}}, r_{\mathrm{src}} \succsim_{w'} r_{\mathrm{tgt}} \wedge$
$sem_{\mathrm{src}}.\mathtt{halted}(m_{\mathrm{src}}, s_{\mathrm{src}}) = \mathtt{Some}\ r_{\mathrm{src}} \wedge sem_{\mathrm{tgt}}.\mathtt{halted}(m_{\mathrm{tgt}}, s_{\mathrm{tgt}}) = \mathtt{Some}\ r_{\mathrm{tgt}}$

(SIM:MODSEM)

$sem_{\mathrm{src}} \succsim_{d, sound\_state} sem_{\mathrm{tgt}} \stackrel{\mathrm{def}}{=} \exists match\_states \in \mathrm{open\_sim}(sem_{src}, sem_{tgt}, sound\_state),$

(INIT) $\forall w \in \mathrm{MR}.\mathtt{t},\ \forall c_{\mathrm{src}}, c_{\mathrm{tgt}},\ c_{\mathrm{src}} \succsim_{w} c_{\mathrm{tgt}} \implies$
$c_{\mathrm{src}}.\mathtt{f} \in \mathrm{ftns}(sem_{\mathrm{src}}.\mathtt{senv}) \wedge c_{\mathrm{tgt}}.\mathtt{f} \in \mathrm{ftns}(sem_{\mathrm{tgt}}.\mathtt{senv}) \implies$
$(sem_{src}.\mathtt{senv}, sem_{tgt}.\mathtt{senv}) \in \mathtt{serel}(d, w) \implies \forall (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sem_{\mathrm{src}}.\mathtt{init\_core}(c_{\mathrm{src}}),$
$\exists (m_{\mathrm{tgt}}, s_{\mathrm{tgt}}) \in sem_{\mathrm{tgt}}.\mathtt{init\_core}(c_{\mathrm{tgt}}),$
$\exists w' \sqsupseteq w,\ ((m_{\mathrm{src}}, s_{\mathrm{src}}), (m_{\mathrm{tgt}}, s_{\mathrm{tgt}})) \in match\_states(w')$

(SIM:MOD)

$M_{\mathrm{src}} \succsim M_{\mathrm{tgt}} \stackrel{\mathrm{def}}{=} \exists d \in \mathrm{SR}.\mathtt{t},\ \exists sound\_state : \mathrm{MP}.\mathtt{t} \to \mathcal{P}(\mathrm{Mem} \times M_{\mathrm{src}}.\mathtt{state}),$

(1) $(M_{\mathrm{src}}.\mathtt{scode}, M_{\mathrm{tgt}}.\mathtt{scode}) \in \mathtt{screl}(d)$

$\wedge$ (2) $\forall se_{\mathrm{src}},\ sound\_state \in \mathrm{open\_prsv}(M_{\mathrm{src}}.\mathtt{sem}\ se_{\mathrm{src}})$

$\wedge$ (3) $\forall d' \sqsupseteq d,\ \forall w,\ \forall (se_{\mathrm{src}}, se_{\mathrm{tgt}}) \in \mathtt{serel}(d', w),$
$M_{\mathrm{src}}.\mathtt{sem}\,(se_{\mathrm{src}}) \succsim_{d, sound\_state} M_{\mathrm{tgt}}.\mathtt{sem}\,(se_{\mathrm{tgt}})$

(SIM:PROG)

$\mathrm{Prog}_{\mathrm{src}} \succsim \mathrm{Prog}_{\mathrm{tgt}} \stackrel{\mathrm{def}}{=}$
$\forall i \in \mathbb{N},\ \mathrm{Prog}_{\mathrm{src}}[i] \succsim \mathrm{Prog}_{\mathrm{tgt}}[i]$

(PRESERVATION)

$sound\_state \in \mathrm{open\_prsv}(sem_{\mathrm{src}}) \stackrel{\mathrm{def}}{=}$

(INIT) $\forall u \in \mathrm{MP}.\mathtt{t}, \forall c_{\mathrm{src}} \in \mathtt{cpred}(u),\ sem_{\mathrm{src}}.\mathtt{senv} \in \mathtt{sepred}(u) \implies \forall (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sem_{\mathrm{src}}.\mathtt{init\_core}(c_{\mathrm{src}}),$
$\exists u' \sqsupseteq u,\ (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sound\_state(u')$

$\wedge$ (STEP) $\forall u,\ \forall (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sound\_state(u), \forall e, m'_{\mathrm{src}}, s'_{\mathrm{src}},\ (m_{\mathrm{src}}, s_{\mathrm{src}}) \overset{e}{\hookrightarrow} (m'_{\mathrm{src}}, s'_{\mathrm{src}}) \implies$
$\exists u' \sqsupseteq u,\ (m'_{\mathrm{src}}, s'_{\mathrm{src}}) \in sound\_state(u')$

$\wedge$ (CALL) $\forall u,\ \forall (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sound\_state(u), \forall c_{\mathrm{src}},\ sem_{\mathrm{src}}.\mathtt{at\_external}(m_{\mathrm{src}}, s_{\mathrm{src}}) = \mathtt{Some}\ c_{\mathrm{src}} \implies$
$\exists u' \sqsupseteq_{\mathrm{prv}} u,\ c_{\mathrm{src}} \in \mathtt{cpred}(u') \wedge$
$\forall u'' \sqsupseteq u',\ \forall r_{\mathrm{src}} \in \mathtt{rpred}(u''),\ \forall m'_{\mathrm{src}}, s'_{\mathrm{src}},\ sem_{\mathrm{src}}.\mathtt{after\_external}(s_{\mathrm{src}}, r_{\mathrm{src}}) = \mathtt{Some}\ (m'_{\mathrm{src}}, s'_{\mathrm{src}}) \implies$
$\exists u''' \sqsupseteq_{\mathrm{prv}} u'',\ u''' \sqsupseteq u \wedge \exists m'_{\mathrm{src}} \in \mathtt{mpred}(u''') \wedge (m'_{\mathrm{src}}, s'_{\mathrm{src}}) \in sound\_state(u''')$

$\wedge$ (RET) $\forall u,\ \forall (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sound\_state(u), \forall r_{\mathrm{src}},\ sem_{\mathrm{src}}.\mathtt{halted}(m_{\mathrm{src}}, s_{\mathrm{src}}) = \mathtt{Some}\ r_{\mathrm{src}} \implies$
$\exists u' \sqsupseteq u,\ r_{\mathrm{src}} \in \mathtt{rpred}(u')$

Figure III.10: Parameterized Open Simulations

line 4: the next states are related by $match\_states(w')$ for a public future world $w' \sqsupseteq w$.

- (CALL) The source and target states invoke related external calls. Specifically:

line 1: certain external functions and arguments in the source and target are related at a private future world $w' \sqsupseteq_{\texttt{prv}} w$, and

line 2: the source and target machine states invoke the related external functions with the related arguments, and

line 3: for any return values and memories related at any public future world $w'' \sqsupseteq w'$,

line 4: if the source safely returns from the external call,

line 5: then the target also safely returns from the external call, and

line 6: the states after return are related by $match\_states(w''')$ for a world w''' that is a private future of w'' and a public future of w.

- (RET) The source and target states halt with related values and memories. Specifically:

line 1: with return values and memories related at w' for a public future world $w' \sqsupseteq w$,

line 2: the source and target machine states halt.

Simulation of Module Semantics Module semantics are related if their initial machine states are related. Specifically, for a symbol relation $d \in \mathrm{SR}$ and a (source) soundness predicate $sound\_state$, a target module semantics $sem_{\texttt{tgt}}$ simulates a source one $sem_{\texttt{src}}$ if for an open simulation $match\_states$:

- (INIT) the initial machine states of $sem_{\texttt{src}}$ and $sem_{\texttt{tgt}}$ are related by $match\_states$. Specifically:

line 1: for any source and target call data related at any world $w \in \mathrm{MR}$,

line 2: if the functions of the source and target call data belong to the modules and

line 3: the symbol environments are related at d and w, then for any initial machine state of the source function call,

line 4: there exists an initial machine state of the target function call such that

line 5: the two initial machine states are related by $match\_states(w')$ for w' a public future of w.

Simulation of Modules Modules are related if their module semantics are related. Specifically, a target module $M_{\texttt{tgt}}$ simulates a source one $M_{\texttt{src}}$ if the following hold for a symbol relation $d \in$ SR and a soundness predicate $sound\_state$:

line 1: the source and target symbol codes are related at d,

line 2: $sound\_state$ satisfies the open preservation property (discussed below), and

line 3: for any symbol environments related at any symbol relation d' extending d and any world w,

line 4: the source and target module semantics for the related symbol environments are related at d and w.

Note that the symbol environments are related at d', which represents the possible symbol information after linking with an arbitrary module, while the module semantics are related at d, which represents the module's own symbol information.

Simulation of Programs Two programs each of which consists of a list of modules are simulated if each corresponding modules are simulated.

Open Preservation with Parameters CompCert's passes are categorized into two groups: analysis pass and translation (optimization) pass. The former computes useful information about the source program so that it can be used in translation passes. Such separation encourages code reuse as an analysis pass can be employed in multiple translation passes.

In verification, CompCert verifies the soundness of an analysis pass modularly, independent from its clients. Specifically, CompCert uses a relation $match\_states$ to prove correctness of a translation pass and a predicate $sound\_state$ to prove correctness of the analyzer such as value analysis, where $sound\_state$ specifies

those states where the analysis results hold. As we do for *match_states*, we perform a similar generalization from a closed setting to an open setting for *sound_state*. Specifically, we generalize the conditions for *sound_state* from preservation to open preservation (*cf.* from simulation to open simulation); and parameterize over memory predicates MP (*cf.* memory relations MR), which intuitively encodes the analysis results of the analyzer. Also, as we do for open simulation, we prove that all Clight and Asm modules satisfy open preservation with MP, which intuitively means that all those context modules preserve the analysis results of the analyzer. Note that the definition of open preservation, open_prsv, is essentially a unary version of that of open simulation, where the (INIT) case corresponds to that of the module semantics simulation and the (STEP), (CALL), and (RET) cases to those of the state simulation.

12.4 Horizontal Compositionality and Adequacy

To use open simulations in RUSC, we prove their horizontal compositionality and adequacy. Let P and Q be programs (*i.e.*, lists of modules) and we define $P \oplus Q$ to be the list concatenation of P and Q. Let $\mathrm{MR} \in \mathrm{MemRel}, \mathrm{SR} \in \mathrm{SymbRel}, \mathrm{MP} \in \mathrm{MemPred}$ be parameters, and $\gtrsim$ be the program simulation relation for the parameters, given in (SIM:PROG) of Figure III.10. Then we have:

Theorem 11 (HorComp)**.** For any programs $P_{\mathtt{src}}$, $P_{\mathtt{tgt}}$, $Q_{\mathtt{src}}$, $Q_{\mathtt{tgt}}$, if $P_{\mathtt{src}} \gtrsim P_{\mathtt{tgt}}$ and $Q_{\mathtt{src}} \gtrsim Q_{\mathtt{tgt}}$:

$$P_{\mathtt{src}} \oplus Q_{\mathtt{src}} \gtrsim P_{\mathtt{tgt}} \oplus Q_{\mathtt{tgt}} \ .$$

Proof. Immediate from the definition of $\oplus$ and (SIM:PROG). □

Theorem 12 (Adequacy)**.** For any programs $P_{\mathtt{src}}$ and $P_{\mathtt{tgt}}$, if $P_{\mathtt{src}} \gtrsim P_{\mathtt{tgt}}$:

$$\mathtt{Beh}(P_{\mathtt{src}}) \supseteq \mathtt{Beh}(P_{\mathtt{tgt}}) \ .$$

Proof. By "weaving" module simulations as in [10]. □

12.5 Instances of Parameters

This section presents intuition behind the three parameters (Section 12.5) with some interesting instances of them.

The virtue of parameterizing MR is that in the (CALL) case of Figure III.10, possible future worlds are restricted with $w'' \sqsupseteq w'$. Actually, all the guarantee conditions $w' \sqsupseteq w$ elsewhere are required to ensure this condition. The most

interesting instances of MR are already presented in Section 9. They are carefully designed so that each function guarantees others' private memories (*e.g.*, dynamic local memories) are unchanged ($w'' \sqsupseteq w'$), and in return, they are guaranteed that their own private memories are unchanged after an external function call.

SR is introduced to verify optimizations that change their symbol code, such as `Unusedglob`. Except for those optimizations, we always use the following trivial quadruple.

$$\begin{aligned}&(\text{Scode} \times \text{Scode},\\ &\ \texttt{fun}\ \_\ \_ \to \top,\\ &\ \texttt{fun}\ d\ sc_{\texttt{src}}\ sc_{\texttt{tgt}} \to d = (sc_{\texttt{src}}, sc_{\texttt{tgt}}),\\ &\ \texttt{fun}\ \_\ \_\ se_{\texttt{src}}\ se_{\texttt{tgt}} \to se_{\texttt{src}} = se_{\texttt{tgt}})\end{aligned}$$

For `Unusedglob`, we use the following instance.

$$\begin{aligned}&((\texttt{d}, \texttt{s}, \texttt{t}) \in \mathcal{P}(\text{Ident}) \times \text{Scode} \times \text{Scode}),\\ &\ (\texttt{fun}\ \ (d, s, t)\ (d', s', t') \to\\ &\qquad d \subseteq d' \wedge s \subseteq s' \wedge t \subseteq t' \wedge (\forall i \in \text{Ident}, i \in d' \wedge i \notin d \implies (i \notin s \wedge i \notin t))),\\ &\ (\texttt{fun}\ \ (d, s, t)\ sc_{\texttt{src}}\ sc_{\texttt{tgt}} \to\\ &\qquad s = sc_{\texttt{src}} \wedge t = sc_{\texttt{tgt}} \wedge (\forall i \in \text{Ident}, i \notin d \implies s\texttt{@}i = t\texttt{@}i) \wedge\\ &\qquad (\forall i \in \text{Ident}, i \in d \implies i \in (\texttt{statics\_of}\ \ s) \wedge t\texttt{@}i = \texttt{None})),\\ &\ \ldots)\end{aligned}$$

Here, `d` means dropped identifiers, `s` source Scode, and `t` target Scode. `statics_of` means the list of identifiers marked as static variable and $s\texttt{@}i$ is a definition in s whose identifier is i. We omit the exact definition of `serel`; it morally means that the injection does not map dropped identifier in the source, and all other identifiers are mapped one-to-one.

Similarly to the above, the virtue of parameterizing SR is that in (INIT) case of Figure III.10, one can rely on $(sem_{src}.\texttt{senv}, sem_{tgt}.\texttt{senv}) \in \texttt{serel}(d, w)$. For this, a pair of the module first guarantees their symbol codes are related: $(M_{\texttt{src}}.\texttt{scode}, M_{\texttt{tgt}}.\texttt{scode}) \in \texttt{screl}(d)$. Recall that in the loading process (Section 11.1), individual symbol codes are linked to constitute a global symbol code, loaded into the global symbol environment, and finally projected into individual symbol environments. There should be a mechanism that ensures the information written in `screl` (*i.e.*, the list of dropped identifiers) to be transferred into `serel` (*i.e.*, dropped identifiers are not injected). For instance, suppose that module A claims that a static identifier i of another module B

is dropped while it actually is not. Then, the resulting injection will not map i, so the `serel` of B is broken. In order to prevent this, `screl` requires that the identifiers one claims to be dropped should be its own *static* variables (`statics_of` s), which prevents invalidating other modules' invariants. Then, such information is maintained throughout the linking process by $\sqsubseteq$ relation.

The idea behind MP is already discussed in Section 12.3. For passes that do not employ the value analysis, we offer the following trivial instance accompanied by an open preservation proof with a trivial *sound_state*, so there is no additional proof obligation:

$$(\texttt{Unit},\ \texttt{fun}\ \_\ \_ \to \top,\ \texttt{fun}\ \_\ \_ \to \top,\ \texttt{fun}\ \_\ \_\ \_ \to \top,\ \texttt{fun}\ \_\ \_\ \_ \to \top,\ \texttt{fun}\ \_\ \_\ \_ \to \top)$$

$$\mathit{sound\_state} \stackrel{\text{def}}{=} \texttt{fun}\ \_\ \_\ \_ \to \top$$

For passes that employ value analysis, we use a unary version of enriched memory injection. Its definition is largely straightforward, so we omit here; instead, we note some interesting points. The idea behind the two examples Section 9 and Figure III.1 are similar: protection of private memory. However, while the former in its essence requires relational reasoning so it should be verified with enriched memory injection, the latter suffices to use unary reasoning, so it is verified with memory identity plus open preservation, which greatly simplifies the proof.

13 CompCertR

In this section, we list the major differences between CompCert and CompCertR.

As discussed, we enriched CompCert's existing memory injection so that it can be used in an open simulation. Roughly speaking, CompCert only *relies* on private memory protection (by system call axioms) while it does not *guarantee* it. This is indeed okay because system calls are expected not to make a mutually recursive call and well-behaved. However, in our open setting, an external call can make a mutually recursive call, and each translation should also guarantee the condition. For instance, suppose that `f` calls `g` and `g` calls `f` again. Here, for the first f to rely on private memory protection, second f needs to guarantee it.

For technical reasons, we also refactored some language's semantics but CompCert's correctness results remain the same. Specifically, we changed the followings:

- We changed the `Callstate` of each language to carry function pointer instead of identifiers. Note that in the open setting, a module can call another module's

function via a function pointer. To be specific, suppose that module X's symbol code is `[b, c]` and the global symbol code is `[a, b, c, d]`. Then, the global symbol environment is $[1 \mapsto \texttt{a}, 2 \mapsto \texttt{b}, 3 \mapsto \texttt{c}, 4 \mapsto \texttt{d}]$, and module X's is $[2 \mapsto \texttt{b}, 3 \mapsto \texttt{c}]$. If the `Callstate` carries identifiers, X cannot call a function `a` via its function pointer 1 because it first needs to lookup its environment with 1 and find the corresponding identifier, but it is lacking. For this reason, we changed `Callstate` to carry more raw information, a function pointer itself, than an identifier.

It is worth noting that CompComp takes a different approach; in their setting X's symbol environment is $[1 \mapsto \texttt{extern a}, 2 \mapsto \texttt{b}, 3 \mapsto \texttt{c}, 4 \mapsto \texttt{extern d}]$, where `extern` means that it just has an identifier and its body is missing. However, this symbol environment is quite different from the one originally used in CompCert, $[1 \mapsto \texttt{b}, 2 \mapsto \texttt{c}]$. We deviated from CompComp's approach because it is desirable to maintain each symbol environment as a mere relocation from the original ones. This property is beneficial when proving optimizations that are sensitive to symbol code/symbol environment, such as `Unusedglob`, which was missing in the days of CompComp.

- CompCert's main function – the entry point of the whole program – does not accept any argument, and this fact is exploited in various proofs. We relaxed such proofs not to rely on empty arguments so that such proofs can be reused in the open setting where each module can be called with arbitrary arguments.
  In detail, in CompCertM we employ a special gadget called *dummy stack* in three consecutive intermediate languages (*LTL, Linear, Mach*) to take initial arguments into account. Those languages are unusual because the callee reads arguments from caller's stack frame; higher-level languages directly pass the argument, and a lower-level language (assembly) passes arguments via memory. Therefore, unlike CompCert, where the initial stack frame of those languages is empty, in CompCertM, we put a *dummy stack* that only contains initial arguments but nothing else. [8] For this, we relaxed CompCert's simulation proof so that it can be used in both CompCert and CompCertM. The two passes that introduce (`Allocation.v`; translation to *LTL*) and eliminates (`Asmgen.v`; translation from *Mach*) dummy stack required slight change. The other

[8] Correspondingly, the `corestep` is defined as follows: $\texttt{corestep}(st_0, e, st_1) \stackrel{\text{def}}{=} st_0 \overset{e}{\hookrightarrow} st_1 \wedge \texttt{get\_stack}(st_0) \neq []$. This is in contrast with CompComp, where they did not reuse the existing $\overset{e}{\hookrightarrow}$ and defined a totally new `corestep`, which led to more engineering effort.

passes whose source and target both have dummy stack (*e.g.*, `Linearize.v`, `Tunneling.v`) are almost unchanged, except for the `Stacking.v` whose proof is deeply involved with the structure of stack frame.

- In CompCert's assembly semantics, the `RSP` register is initialized to an integer 0. This is okay for CompCert generated assemblies where the initial value of `RSP` is never used, but in our setting, we may include a hand-written assembly module, so a more faithful formalization would be desirable. Therefore, we initialize `RSP` with a junk pointer.[9] As we did in Section 8.1, we adjust the initial memory to contain a junk block (block number 1), and initialize `RSP` with that address.

The correctness result of CompCert remains the same, and the reason is as follows.

- The semantics of the source language is unchanged. In `ClightBigstep.v` the equivalence between small-step style and big-step style is proven. We did not modify the big-step style at all, and the equivalence is still proven.

- The only thing that is changed in the target language's semantics is the initial value of RSP register, where our version admits more behavior.

Additionally, we have strengthened the C type checker in order to prove Theorem 9; our modification is faithful w.r.t modern C compliers.Specifically, our modified type checker additionally rejects the following cases: (1) void function returning a value, (2) non-void function not returning a value, and (3) ill-formed composite types. With these enhancements, we proved that when executing a well-typed C module, arguments of a function call and a return value of a function are always well-typed. [10] This additional property guarantees that nothing weird happens in the inter-module step, which is necessary for proving Theorem 9.

14 Related Work

We discuss related work on compositional compiler correctness for CompCert and other higher-order languages.

[9] We can also use `undef` value but junk pointer is more convenient in simulation proof.

[10] CompCert's type soundness theorem already maintains all the value in its state are well-typed.

14.1 Compositional Correctness for CompCert

CompComp Besides what we have discussed, CompComp introduces self-relatedness as a part of the notion of well-defined context and shows refinement under well-defined contexts as a result of the compiler correctness proof, whereas we uses such refinement as a method to prove compiler correctness. Also, the PhD thesis of [26] observed, with a counterexample, one of the three reasons for inadequacy of interaction semantics at assembly level: not enforcing the assumption on the outgoing arguments area of the stack. It informally concludes that assembly contexts should respect the compiler's assumption without giving a formal solution. Our repaired interaction semantics gives a formal way to enforce the assumption by giving UB to those behaviors violating it.

CompCertX Besides what we have discussed, the latest version of CompCertX [28] supports two features that CompCertM currently does not support. First, it proves that CompCertX preserves the stack consumption by instrumenting the languages' semantics to record the size of the concrete stack frames. Second, it carries the compiler correctness down to assembly with the flat memory model instead of CompCert's block-based memory model. On the other hand, CompCertX instruments the languages' semantics to record permissions in the stack frames in order to support address-taken stack variables, whereas CompCertM supports them, without instrumenting the semantics, by adding the private memory components to memory injections as shown in Section 9. Interesting future work would be to apply the techniques of CompCertX to CompCertM to support the two missing features, and conversely apply the technique of CompCertM to CompCertX to support address-taken stack variables without recording permissions on the stack.

SepCompCert SepCompCert [15] proves a weaker form of compositional correctness for CompCert, namely correctness of separate compilation. Specifically, the proof assumes that all modules are separately compiled by the *same* compiler and then linked together without linking with any handwritten assembly. For this, SepCompCert employs a surprisingly lightweight *closed* simulation technique, which, therefore, has been officially adopted by CompCert since version 2.7.

CASCompCert CASCompCert [11] extends CompComp to support concurrency in the absence of data races, which demonstrates that the proof technique of CompComp (*i.e.*, structured simulations) scales to a concurrent setting. However, in the Coq formalization, CompComp tames the complexity of structured simulations by (*i*) not allowing address-taken stack variables (although how to support them using structured simulations is described with paper proofs in the associated technical report[11]); and (*ii*) only covering 12 out of the 20 passes in its base version, CompCert 3.0.1 (although the 12 passes are exactly those that are covered by the original CompComp): these restrictions unnecessitate the use of memory injection. Also, CASCompCert can only allow special nondeterminism caused by scheduling threads by slightly relaxing the conditions for forward simulation, while CompCertM can allow arbitrary nondeterminism by mixing forward and backward simulations.

We do not currently see any problem with applying the approach of CASCompCert to CompCertM to support concurrency in the absence of data races. Moreover, we expect that the compiler verification technique for *promising semantics* [13], which is also based on simple closed simulations, applies to CompCertM to fully support relaxed-memory concurrency.

14.2 Compositional Compiler Correctness for Higher-Order Languages

Pilsner Pilsner [19, 10] is a multi-pass optimizing compiler from a higher-order imperative language down to an idealized assembly language. To verify horizontally and vertically compositional correctness in the presence of higher-order functions, Pilsner uses very general and flexible open simulations, called *parametric simulations*, whose vertical compositionality proof is also technically very involved. Since it would be hard to define interaction semantics due to the different representations of values and memory in the source and target languages, the RUSC technique is unlikely to be applicable to Pilsner.

Also, our approach to reasoning about dynamically and statically allocated local memory, presented in Section 12, follows that of Pilsner, which is based on the work of [5]. A minor difference is that we simplify the formulation by restricting the occurrence of private transitions only to just before and after external calls, while Pilsner allows private transitions at every local step only requiring public transitions between the end-to-end worlds of the execution of a

[11] https://plax-lab.github.io/publications/ccc/ccc-tr.pdf

function.

Multi-language semantics Ahmed and her collaborators propose multi-language semantics [23, 22, 24, 20, 21] as an approach to prove compositional correctness and full abstraction of a compiler for both assembly-like and higher-order languages. Specifically, they define a language that combines all of the source, intermediate and target languages, and prove contextual equivalence and/or full abstraction for each translation pass in the combined language using logical relations (with back-translations). In this approach, they rule out ill-formed contexts by syntactic type systems and use the typed contextual equivalence for compositionality. Since RUSC rules out ill-formed contexts by semantic program relations, it would be interesting to see if RUSC could be applicable and beneficial to the approach of Ahmed *et al.*, in particular, for full abstraction.

Chapter IV

Program Verification

In this chapter, we will show the potential of RUSC-as-a-program-logic with interesting examples and give a comprehensive comparison with modern variants of Hoare logic (such as [1]), arguably the most successful and widely used program verification technique so far. We first give a short introduction to Hoare logic (Section 15), discuss its problems (Section 16), and present how we verify interesting examples without the aforementioned problems (Section 17). However, RUSC-as-a-program-logic is still in its early stage, so we clarify its current limitations, compare it with Hoare logic, and discuss future research directions (Section 18). Finally, we list up related works (Section 19).

15 Background

Hoare logic, since its origination in the 1960s, has shown great success as a program verification technique, and modularity is at the heart of its success. In Hoare logic, for a given program *prog*, verifier tries to establish the following formula: $\{P\}$ *prog* $\{Q\}$. This is called *Hoare triple* and means that if the *prog* starts in a state satisfying the *precondition* P, its returning state should satisfy the *postcondition* Q. Then, the following *sequence* (or sequential composition) rule holds.

$$\frac{\{P\}\ c_0\ \{Q\} \qquad \{Q\}\ c_1\ \{R\}}{\{P\}\ c_0; c_1\ \{R\}}\ (\textsc{Sequence})$$

The rule says that two Hoare triples can be composed as long as the former's postcondition coincides with the latter's precondition. Thanks to this sequence rule, the verifier can modularly verify each instruction of the program and then compose them to establish whole program correctness.

Modern higher-order variants of Hoare logic[1, 3] supports another useful modular reasoning principle, which is written below (simplified for presentation purpose).

$$\frac{\begin{array}{c}\{P_g\}\ \texttt{call g}\ \{Q_g\} \implies \{P_f\}\ \texttt{f}\ \{Q_f\} \\ \{P_f\}\ \texttt{call f}\ \{Q_f\} \implies \{P_g\}\ \texttt{g}\ \{Q_g\}\end{array}}{\begin{array}{c}\{P_f\}\ \texttt{f}\ \{Q_f\} \\ \{P_g\}\ \texttt{g}\ \{Q_g\}\end{array}}\ (\textsc{Call})$$

When verifying two mutually recursive functions, `f` and `g`, this reasoning principle allows one to verify `f` assuming the specification of `g` and vice versa. Then, together with the sequence rule, one can pass by `call f` (and `call g`, respectively) instead of going through its body. The principle is seemingly unsound because it is circular reasoning, but it is indeed sound. In order to justify the principle, a technique called *step-index* is employed. With this, it is sufficient to verify that `f`'s specification holds until $k + 1$ steps assuming the specification of `g` holds until k steps and vice versa. Then, by using induction on the step-index k, one can show that f and g's specifications hold for any finite number of steps, which is sufficient because Hoare triple requires only the *partial correctness*. Partial correctness means that it does not say anything about termination; recall that Hoare triple requires the postcondition to hold *if* it happens to terminate.

16 Problems

However, those modern variants of Hoare logic also have few drawbacks. First, the step-index complicates the underlying model, which makes soundness result (or even the meaning of a Hoare triple) esoteric. Second, it supports only partial correctness (*i.e.*, cannot prove termination). In fact, if we change the meaning of Hoare triple to guarantee *total correctness* (*i.e.*, both the postcondition

and termination), the call rule becomes unsound. [1] Therefore, to prove total correctness, one needs to employ a separate tool. Third, its adequacy results hold only when the whole program is verified with the same Hoare logic. This is restrictive because we often want to verify each module with different tools (such as model checker). Finally, it cannot verify a program that communicates with the outside world. Think of a simple REPL, a web server, or an operating system; it is even unclear how to write a Hoare triple for such a program.

17 Our Approach

In this section, we first present our approach with the `mutual-sum` example (Section 17.1). To the best of our knowledge, our framework is the first, in the context of CompCert, that is capable of verifying the *mutually recursive* modules. Then, we point out that our approach does not suffer from the aforementioned problems in Section 16 while supporting the modular reasoning principle (Section 17.2). Additionally, we verify `utod`, CompCert's internal handwritten assembly function whose behavior is axiomatized, and show that such axioms can be removed (Section 17.3). Finally, we report their verification efforts (Section 17.4).

17.1 Verification of `mutual-sum`

Section 17.1 shows a C module, `a.c`; a handwritten assembly module, `b.asm` (presented in C syntax for readability); their open specification modules, `a.spec` and `b.spec`; and the combined closed specification module `ab.spec`. Both functions `f` in `a.c` and `g` in `b.asm` mutually recursively compute the summation from 0 up to the given argument integer i (denoted $\mathrm{sum}(i)$), performing different memoization optimizations. The function `f` memoizes the result of `f(i)` in the static variable `memoized1[i]`, which is initialized with zero representing invalid value. The function call `f(i)` first reads the memoized value, and returns it if it is valid; otherwise, it calculates, memoizes, and returns `g(i-1)`, expected to be $\mathrm{sum}(\texttt{i} - 1)$, plus `i`. On the other hand, the function `g` memoizes only the result of the latest call `g(i)` with the index `i`, where `memoized2[0]` $=$ `i` and `memoized2[1]` $=$ `g(i)`. The code of `g` is self-explanatory under the assumption that the call `f(i-1)` returns $\mathrm{sum}(\texttt{i} - 1)$.

[1] Here is a simple counter example: `f` := `call f`. Here, one can prove arbitrary pre/-postcondition on `f` by using the call rule, which in turn implies that `f` terminates. This is contradiction.

a.c

```
static int memoized1[1000] = {0};
int f(int i)   {
  int sum;
  if (i == 0) return 0;
  sum = memoized1[i];
  if (sum == 0) {
    sum = g(i-1) + i;
    memoized1[i] = sum;
  }
  return sum;
}
```

$\bigoplus$ b.asm

```
// hand-optimized in assembly
static int memoized2[2] = {0,0};
int g(int i) {
  int sum;
  if (i == 0) return 0;
  if (i == memoized2[0]) {
    sum = memoized2[1];
  } else {
    sum = f(i-1) + i;
    memoized2[0] = i;
    memoized2[1] = sum;
  }
  return sum;
}
```

a.spec with $\mathbf{X} = \mathsf{f}, \mathbf{Y} = \mathsf{g}$

b.spec with $\mathbf{X} = \mathsf{g}, \mathbf{Y} = \mathsf{f}$

$\mathtt{States} := \{\, \mathtt{Init}\ i \mid 0 \le i \,\} \uplus \{\, \mathtt{Ecall}\ i \mid 0 \le i \,\} \uplus \{\, \mathtt{Ret}\ r \mid 0 \le r \,\}$
$\mathtt{init\_core} := \{\, (\mathbf{X}, [i], \mathtt{Init}\ i) \mid 0 \le i < 1000 \,\}$
$\mathtt{at\_external} := \{\, (\mathtt{Ecall}\ i, \mathbf{Y}, [i-1]) \mid 0 < i < 1000 \,\}$
$\mathtt{after\_external} := \{\, (\mathtt{Ecall}\ i, \mathrm{sum}(i-1), \mathtt{Ret}\ \mathrm{sum}(i)) \mid 0 < i < 1000 \,\}$
$\mathtt{halted} := \{\, (\mathtt{Ret}\ r, r) \mid 0 \le r \,\}$
$\mathtt{step} := \{\, ((\mathtt{Init}\ i, m), \tau, (\mathtt{Ret}\ \mathrm{sum}(i), m)) \mid 0 \le i < 1000 \,\} \cup$
$\quad \{\, ((\mathtt{Init}\ i, m), \tau, (\mathtt{Ecall}\ i, m)) \mid 0 < i < 1000 \,\}$

ab.spec

$\mathtt{States} := \{\, \mathtt{Init}\ i \mid 0 \le i \,\} \uplus \{\, \mathtt{Ret}\ r \mid 0 \le r \,\}$
$\mathtt{init\_core} := \{\, (\mathsf{f}, [i], \mathtt{Init}\ i) \,\} \cup \{\, (\mathsf{g}, [i], \mathtt{Init}\ i) \,\}$
$\mathtt{at\_external} := \{\ \}$
$\mathtt{after\_external} := \{\ \}$
$\mathtt{halted} := \{\, (\mathtt{Ret}\ r, r) \mid 0 \le r \,\}$
$\mathtt{step} := \{\, ((\mathtt{Init}\ i, m), \tau, (\mathtt{Ret}\ \mathrm{sum}(i), m)) \mid 0 \le i < 1000 \,\}$

The open specification modules `a.spec` and `b.spec` are the same except that the names of the internal and external functions are swapped. This is natural because the two functions `f` and `g` compute the same summation. The open specification `a.spec` is an abstract, nondeterministic, version of the function `f` in `a.c` including all the observable behaviors of `f`. It has three kinds of states, `Init` i, `Ecall` i and `Ret` r, representing the initial state with argument i, the call state executing $\mathsf{g}(i-1)$, and the halt state returning r, respectively. Then `init_core` starts with `Init` i when `f` is invoked with argument i if $0 \le i < 1000$, otherwise UB; `at_external` recognizes `Ecall` i as the state invoking `g` with $i-1$; `after_external` transitions from `Ecall` i to `Ret` $\mathrm{sum}(i)$ only when the return value from the external call $\mathsf{g}(i-1)$ is $\mathrm{sum}(i-1)$, otherwise UB, which means that this module gives a conditional specification under the assumption

that $\texttt{g}(i)$ returns $\text{sum}(i)$; `halted` recognizes `Ret` r as the halted state returning r; and finally `step` transitions from `Init` i to either `Ret` $\text{sum}(i)$ or `Ecall` i nondeterministically (without updating the memory), where the former abstracts reading from memoization and the latter recursively computing the sum. The same applies to `b.spec`. Finally, the combined specification `ab.spec` does not make any external function call and simply returns the summation.

Then, we perform our verification as follows. First, we prove $\texttt{a.spec} \succcurlyeq_{\mathcal{R}} \texttt{a.c}$ using memory injections with the following invariant:

$$\forall 0 \leq i < 1000,\ \texttt{memoized1}[i] = 0 \vee \texttt{memoized1}[i] = \text{sum}(i)\ .$$

Second, we prove $\texttt{b.spec} \succcurlyeq_{\mathcal{R}} \texttt{b.asm}$ using memory injections with the following invariant:

$$\exists 0 \leq i < 1000,\ \texttt{memoized2}[0] = i \wedge \texttt{memoized2}[1] = \text{sum}(i)\ .$$

Finally, we prove $\texttt{ab.spec} \succcurlyeq_{\mathcal{R}} \texttt{a.spec} \oplus \texttt{b.spec}$ using the memory identity. Note that $\mathcal{R}$ is the set containing open simulations with the three memory relations used in the above verification (*i.e.*, memory injections with the two invariants above and the memory identity).

17.2 Advantages

Our approach does not suffer from the aforementioned problems in Section 16 while supporting the modular reasoning principle (Section 17.2).

First and foremost, by passively modeling the expected behavior of other modules with UB, we do not have any circularity in our reasoning. Therefore, we do not need a step-index, so our specification is clean and easy to understand. Furthermore, as our specification is written in operational semantics, it can even be executed and tested. It is crucial to have a reliable and understandable specification, perhaps as important as the verification itself, and testing is widely adopted as a tool to establish such trust. Second, we trivially support total correctness because the notion of behavior is termination-sensitive, and we prove behavioral refinement. Third, we do not impose any restriction on the client module because RUSC quantifies over an arbitrary context. Finally, our specification module can trigger events just as C and assembly modules do, so we can definitely give a REPL-like specification. Note that while we give terminating specifications for terminating implementations and reactive (non-terminating and consistently communicating with the outside world, like REPL) specification for reactive modules, their correctnesses are all expressed in the same RUSC relation. This is in contrast with where they define Hoare

```
double my_func() {
  unsigned long x = 42;
  double y = (double) x;
  return y;
}
```

(a) Source program purely written in C

```
double my_func() {
  unsigned long x = 42;
  double y = __compcert_i64_utod(x);
  return y;
}
```

(b) Translated program calling a runtime function

```
Axiom i64_helpers_correct : ... ∧
  (∀ x y, Val.floatoflongu x = Some y →
    external_impls "__compcert_i64_utod" [x] y)
```

(c) Dedicated axiom for the runtime function

```
__compcert_i64_utod:
testq %rdi, %rdi
jmp .L100
xorpd %xmm0, %xmm0
cvtsi2sdq %rdi, %xmm0
ret
.L100:
movq  %rdi, %rax
shrq  $1, %rax
andq  $1, %rdi
orq %rdi, %rax
xorpd %xmm0, %xmm0
cvtsi2sdq %rax, %xmm0
addsd %xmm0, %xmm0
ret
```

(d) Code of the runtime function

Figure IV.1: Verification of `utod`

triple for partial correctness and total correctness separately, and these two do not coexist.

17.3 Verification of `utod`

`__compcert_i64_utod` (Figure IV.1 - (d)) is one of the CompCert's internal handwritten assembly functions, which converts `unsigned long` to `double` by utilizing architecture-specific instructions like `cvtsi2sdq`. Note that the call to this assembly function is introduced during compilation passes; for example a source program purely written in C (Figure IV.1 - (a)) is translated to `__compcert_i64_utod` (Figure IV.1 - (b)). In order to justify this translation, CompCert currently axiomatizes the behaviors of such runtime libraries as described in Figure IV.1 - (c).

We demonstrate that such axioms can be essentially removed in CompCertM by proving the axiom for `__compcert_i64_utod`. We first turn the axiom for `__compcert_i64_utod` into a specification module and then establish an open simulation with memory injections between the assembly module containing `__compcert_i64_utod` and the specification module.

Table IV.1: SLOC of additional developments

Portion	`mutual-sum`	`utod`
Pass Proofs	3,088	361
The Rest	2,707	424
Total	5,795	785

17.4 Verification Effort

Table IV.1 shows SLOC for the program verification examples presented above.

Pass proofs (establishing open simulation between the specification and the implementation module) are mostly mechanical and tedious. They mostly consist of executing C (or assembly) semantics step-by-step, making a simple reduction on the local environment (or register state), and applying memory related lemmas appropriately. Note that all of these are shared patterns among general C (or assembly) program verification. We did not do any proof automation at all, so there is a big room for improvement.

For "The Rest," `mutual-sum` and `utod` have quite different reasons for their SLOC. In `utod`, the whole SLOC is about proving the self-relatedness of the specification module. Recall that we verified self-relatedness for arbitrary C and assembly modules but not for arbitrary specification modules (it is not true). In `mutual-sum`, however, about 1,089 SLOC are from $\texttt{ab.spec} \succcurlyeq_{\mathcal{R}} \texttt{a.spec} \oplus \texttt{b.spec}$. As discussed above, the proof of this translation is mainly about termination, which should (morally) be easy. The main technical hassle that made this long SLOC is that we had to take unknown context into account. Such reasoning can be made as a general-purpose meta-theory.

18 Limitations and Future Works

At the moment, our framework has a number of limitations, but we plan to extend it and use it to verify a realistic operating system. In the following list, we clarify our current limitations (by comparing them with modern variants of Hoare logic) and discuss future research directions.

- Specification language: it is often very inconvenient and inefficient to articulate each and every state and its transitions. Instead, what we want is a language that can describe state-transition systems concisely while maintaining expressibility.

- Module-local abstract state: for writing a richer specification module, we want

each module to have its own abstract state that is shared among different invocations. We already have such a mechanism – namely, a static variable – so that we may able to encode any mathematical data to an integer (*e.g.*, by Gödelization), but it is very inconvenient.

- Notion of separation: modern variants of Hoare logic also employ a notion of *separation* – hence called higher-order separation logic – which gives a very natural way to reason about programs handling shared resources (*e.g.*, memory). In a smaller sense, we can think of *intra-module* separation, by which we mean the separation between memory locations owned by a single module. It is already supported, and a simpler form of it is even implemented in CompCert.[2] However, what is more interesting is *inter-module* separation.

```
{{ p |-> 0 }}                                    {{ }} void g() {{ }}
int f(int* p) {
  *p = 42;
{{ p |-> 42 }}
  g();
{{ p |-> 42 }}
  print(*p); --> print(42);
}
```

In the above example, ignore the code colored blue at the moment. A function `f` takes a parameter `p`, initializes it with `42`, calls an external function `g`, and finally prints `*p`. Note that the pointer `p` is from an outside world (*i.e.*, leaked); thus, `p` may point to a value other than 42 after the call to `g`. However, if the function `f` is given *full ownership* of the pointer `*p` – which prevents the outside world from accessing it – the `print(*p)` can be abstracted into `print(42)`.

With separation logic, such reasoning is naturally supported. The precondition of `f` (first line with blue color) specifies that it will require the full ownership of `p`, and it points to value 0 (`p |-> 0`). Then, when calling `g`, its precondition does not require the ownership of `p`, which means that it does not access the location of `p`. Therefore, it is guaranteed that `p` points to the same value after the call to `g`, thus justifying the abstraction.

The key idea here is that by employing the notion of separation and writing specifications with respect to it, one can accommodate a *richer form of*

[2] `In Separation.v`

cooperation between modules. As discussed, we do support cooperation between modules to some extent, but ours are not expressive enough. For instance, in `f`'s specification, we can execute UB if the memory after the call to `g` has an unexpected value in `p`'s address. However, in order to erase such UB when *merging* the modules `f` and `g`, an additional proof is required. This is in contrast with separation logic, where each module is verified modularly according to precise specifications (expressed in terms of separation), and there is no additional obligation on the top level.

- Concurrency: The modern variants of Hoare logic also support concurrent programs – hence called higher-order concurrent separation logic – and we plan to support it too. The key ingredients of higher-order current separation logic are the notion of separation and invariants. We already discussed the former, and the latter is already supported. Specifically, when the execution begins (*e.g.*, the function begins), we assume MR.`mrel`, and when the execution ends (*e.g.*, the function returns), we guarantee it. Such mechanisms are precisely the same as opening/closing the invariants.
  As we are mainly concerned with verifying operating systems, we are not interested in the fork/join model or more abstract notion of concurrency (*e.g.*, message passing). For our purpose, it is sufficient to have a simple model where the number of concurrent agents (*i.e.*, cores) and the programs they are executing are predetermined. Instead, we want to concentrate on supporting a more realistic concurrency model (*e.g.*, [13]).

- Automation: modern variants of higher-order concurrent separation logic[12, 1] are equipped with highly automated tactics that greatly reduce proof effort. In contrast, we do not have any automation yet. We plan to adopt the automation engine from separation logic. Also, note that our approach naturally supports gradual abstraction (by vertical compositionality). This is also favorable for automation because each abstraction is kept small, and we can choose different tools for each abstraction. For instance, when abstracting just arithmetic expressions, we may use an SMT-solver.

Also, for a fair comparison with higher-order concurrent separation logic, we clarify that our framework does not support higher-order functions. Overcoming this limitation is also an interesting future work, but it is not our priority at the moment.

19 Related Works

CompComp The thesis suggests *closed* specification modules (*i.e.*, without making external calls) written in Coq's Gallina language, which foreshadows our *open* specification modules and verification against them.

CompCertX and CertiKOS Using CompCertX as a backend compiler, CertiKOS– an OS kernel specially developed for verification purposes – has been verified. As discussed in Chapter III, CompCertX has several restrictions: it does not support (mutual) recursion and passing a pointer as an argument. Also, they write specifications in CAL, which only works when each function does not make an externally visible event and is guaranteed to terminate. On the other hand, they have a notion of a module-local abstract state while we do not. Also, their recently extended version [9] supports some form of concurrency while we do not.

Chapter V

Conclusion

We conclude by recalling the main contributions of this dissertation. We developed a novel modular verification technique called RUSC (Chapter II) and demonstrated its usefulness by applying it to both compiler verification (Chapter III) and program verification (Chapter IV). On the one hand, RUSC is a significant step forward for compiler verification; it achieves both the flexibility of CompCertX and the generality of CompComp. We have applied RUSC to CompCert and developed CompCertM, a full extension of CompCert supporting multi-language linking. On the other hand, RUSC as a program verification technique is in its early stage but shows considerable potential. We have applied RUSC to verify interesting examples and discussed its advantages, current limitations, and future research directions.

Bibliography

[1] A. W. Appel. Verified software toolchain. In *Proceedings of the 20th European Symposium on Programming*, ESOP 2011, 2011.

[2] A. W. Appel. *Program Logics for Certified Compilers.* Cambridge University Press, 2014.

[3] A. W. Appel. *Program Logics for Certified Compilers.* Cambridge University Press, 2014.

[4] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory c. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems*, ESOP 2014, 2014.

[5] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2010, 2010.

[6] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. Certikos: A certified kernel for secure cloud computing. In *Proceedings of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys 2011, 2011.

[7] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2015, 2015.

[8] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. Certikos: An extensible architecture for building certified concurrent os

kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 2016, 2016.

[9] R. Gu, Z. Shao, J. Kim, X. N. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 646–661, New York, NY, USA, 2018. Association for Computing Machinery.

[10] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and kripke logical relations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2012, 2012.

[11] H. Jiang, H. Liang, S. Xiao, J. Zha, and X. Feng. Towards certified separate compilation for concurrent programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, 2019.

[12] R. JUNG, R. KREBBERS, J.-H. JOURDAN, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 01 2018.

[13] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2017, 2017.

[14] J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis. A formal c memory model supporting integer-pointer casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, 2015.

[15] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, 2016.

[16] X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2006, 2006.

[17] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[18] J. R. Lorch, Y. Chen, M. Kapritsos, B. Parno, S. Qadeer, U. Sharma, J. R. Wilcox, and X. Zhao. Armada: Low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 197–210, New York, NY, USA, 2020. Association for Computing Machinery.

[19] G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, 2015.

[20] M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, 2016.

[21] D. Patterson and A. Ahmed. The next 700 compiler correctness theorems (functional pearl). In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2019, 2019.

[22] D. Patterson, J. Perconti, C. Dimoulas, and A. Ahmed. Funtal: Reasonably mixing a functional language with assembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, 2017.

[23] J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *Proceedings of the 23rd European Symposium on Programming*, ESOP 2014, 2014.

[24] G. Scherer, M. S. New, N. Rioux, and A. Ahmed. FabULous interoperability for ML and a linear language. In *Proceedings of the European Joint Conferences on Theory and Practice of Software*, ETAPS 2018, 2018.

[25] Y. Song, M. Cho, D. Kim, Y. Kim, J. Kang, and C.-K. Hur. Compcertm: Compcert with c-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019.

[26] G. Stewart. *Verified Separate Compilation for C*. PhD thesis, Princeton University, 2015.

[27] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2015, 2015.

[28] Y. Wang, P. Wilke, and Z. Shao. An abstract stack based approach to verified compositional compilation to machine code. In *Proceedings of the 46th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2019, 2019.

초록

현대의 소프트웨어 시스템은 매우 복잡하고, 이 복잡성을 길들이기 위해 모듈별로 나눠 개발하는 것은 매우 중요하다. 이런 시스템을 검증하려면, 모듈별로 나눠 검증하는 검증 기법이 필수적이다. 하지만, 기존의 방법들은 만족스럽지 못하다. 본 박사학위 논문에서 우리는 나눠서 검증하는 새로운 기법인 RUSC (Refinement Under Self-related Contexts)를 개발하고, 그 유용함을 번역기 검증과 프로그램 검증을 통해 입증한다.

우리는 RUSC를 이용하여 번역기 검증의 최첨단을 개척하였다. 구체적으로, 우리는 전체 CompCert를 확장하여 적은 비용으로 아무런 제약 없이 다중 언어 링킹을 지원하는 CompCertM을 개발하였고, 이것은 기존의 최첨단 기술인 CompCertX와 Compositional CompCert의 결과를 능가한다. 다른 한편, RUSC를 프로그램 검증에 사용하는 것은 아직 초기 단계이지만 주목할만한 잠재력을 보여준다. 고차 분리 논리(higher-order separation logic)와 비교하여, 우리의 방법은 더 간단한 명세와 더 강력한 결과를 제공하지만, 검증 비용이 훨씬 높고 복잡한 기능들을 지원하지 않는다.

주요어:
학번: 2015-21244

감사의 글

연구실에 입학하기 위해 처음 교수님을 만나 뵈었던 때가 바로 어제같이 생생합니다. 교수님께서는 세상에 도움이 되는 재미있는 문제를 해결해나가자고 하셨었고, 지난 6년간 정말로 그런 시간을 보낼 수 있어서 감사했습니다.

저의 지도교수님이신 허충길 교수님께 감사드립니다. 교수님께서는 논문을 읽고, 문제를 찾고, 해결하고, 구현하고, 쓰고, 발표하는 모든 과정에서 저를 이끌어주셨고, 이를 통해 연구란 무엇인지 조금씩 배워나갈 수 있었습니다. 무엇보다도 교수님의 연구자로서의 자세를 배워나갈 수 있어 감사했습니다. **깊은 이해**란 무엇인지, 또 이를 위해서 얼마나 노력해야 하는지 배울 수 있었습니다. 복잡하고 거창한 것을 만드는 게 아니라, 실제 문제를 가장 단순하게 해결하려는 태도를 배울 수 있었습니다. 남들이 해놓은 일에 겁먹지 않고, 자신감을 갖고 문제에 도전하는 게 얼마나 중요한지 배울 수 있었습니다.

이광근 교수님께 감사드립니다. 교수님께서 한국 프로그래밍 언어 동네의 토양을 도탑게 가꾸어주셔서, 저도 조그마한 싹을 틔울 수 있었습니다. 이를테면, 교수님께서는 항상 우리말로 쉽고 정확하게 표현하라고 말씀해주셨고, 저는 그 과정에서 각 단어에 대한 견고한 이해를 쌓아나갈 수 있었습니다. 특히 compiler, interpreter를 번역기, 실행기로 표현하면 된다는 것을 배웠을 때의 충격은 잊을 수 없습니다.

강지훈 교수님에게 감사드립니다. 교수님은 제가 아는 가장 훌륭한 엔지니어이고, 짧지 않은 시간 함께 일하며 저도 훌쩍 성장할 수 있었습니다. 특히 생산성과 완벽함 사이의 오묘한 균형을 잡는 방법, 먼저 뼈대를 잡고 살을 채워나가는 효율적인 개발 방법을 배울 수 있었습니다. 연구실 선배로서 저에게 정말 많은 조언을 해주셨고, 어려울 때 언제든 의지할 수 있는 든든한 버팀목이 되어주셨습니다.

조민기에게 감사합니다. 민기는 비상한 이해력과 문제해결 능력, 논리적 모순을 찾아내는 날카로움으로 제 연구에 큰 도움이 되어주었습니다. 끊임없는 지적인 호기심과 밝은 에너지로 항상 주변을 밝혀주었고, 저에게 귀감이 되어주었습니다.

연구 외적으로도 많은 철학과 과학 상식을 배울 수 있었습니다.

소프트웨어 원리 연구실과 프로그래밍 연구실의 동료분들께 감사드립니다. 이렇게 뛰어난 동료들과 이렇게 허물없이, 이렇게 즐겁게 시간을 보낼 수 있는 날은 아마도 다시 오기 힘들 것 같습니다. 연구실 모든 구성원들께서 연구실에 크고 작은 일이 있을 때마다 발 벗고 나서 주셔서 감사합니다. 특히 강지훈 교수님, 김윤승형, 이준영, 김용현에게 감사합니다. 함께 연구할 때 저를 믿고 잘 따라와 준 조민기, 김동주, 김용현에게 감사합니다. 같이 맛집도 다니고 잡담도 하면서 즐겁게 지내게 해준 이준영, 이동권, 김세훈, 이성환, 조민기에게 감사합니다. 항상 따뜻하게 반겨주시고, 선배로써 많은 조언 해주시는 이우석 교수님, 허기홍 교수님에게 감사드립니다.

기나긴 대학원 과정을 잘 마무리할 수 있게 도와주신 부모님과 가족에게 이 논문을 바칩니다. 물심양면으로 지원해주신 덕분에 부족함 없이 편안한 마음으로 공부에 임할 수 있었습니다. 이것이 얼마나 감사할 일인지 잘 알고 있습니다. 갚아나가며 살겠습니다.