CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification

YOUNGJU SONG, Seoul National University, Korea
MINKI CHO, Seoul National University, Korea
DONGJOO KIM, Seoul National University, Korea
YONGHYUN KIM, Seoul National University, Korea
JEEHOON KANG, KAIST, Korea
CHUNG-KIL HUR*, Seoul National University, Korea

Supporting multi-language linking such as linking C and handwritten assembly modules in the verified compiler CompCert requires a more compositional verification technique than that used in CompCert just supporting separate compilation. The two extensions, CompCertX and Compositional CompCert, supporting multi-language linking take different approaches. 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.

In this paper, we develop a novel lightweight verification technique, called RUSC (Refinement Under Self-related Contexts), and demonstrate how RUSC can solve the problem without any restrictions but still with low verification overhead. For this, we develop CompCertM, a full extension of the latest version of CompCert supporting multi-language linking. Moreover, we demonstrate the power of RUSC as a program verification technique by modularly verifying interesting programs consisting of C and handwritten assembly against their mathematical specifications.

CCS Concepts: • **Software and its engineering** → **Software verification**; **Compilers**; • **Theory of computation** → *Program verification*.

Additional Key Words and Phrases: Compositional Compiler Verification, CompCert, Multi-Language Linking

ACM Reference Format:
Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (January 2020), 31 pages. https://doi.org/10.1145/3371091

1 INTRODUCTION

CompCert [Leroy 2006, 2009], the first *verified optimizing* compiler for *the C programming language*, has served as a backend in end-to-end verified software [Appel 2014]. Specifically, CompCert compiles programs written in (a large subset of) C down to assembly code via various translation

*Corresponding author.

Authors' addresses: Youngju Song, Seoul National University, Korea, youngju.song@sf.snu.ac.kr; Minki Cho, Seoul National University, Korea, minki.cho@sf.snu.ac.kr; Dongjoo Kim, Seoul National University, Korea, dongjoo.kim@sf.snu.ac.kr; Yonghyun Kim, Seoul National University, Korea, yonghyun.kim@sf.snu.ac.kr; Jeehoon Kang, KAIST, Korea, jeehoon.kang@kaist.ac.kr; Chung-Kil Hur, Seoul National University, Korea, gil.hur@sf.snu.ac.kr.

2475-1421/2020/1-ART23
https://doi.org/10.1145/3371091

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 [Gu et al. 2011, 2016] and VST [Appel 2011].

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 [Gu et al. 2015; Wang et al. 2019] and Compositional CompCert (shortly, CompComp) [Beringer et al. 2014; Stewart et al. 2015], 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 [Gu et al. 2011, 2016] 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.

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 [Stewart 2015] (see §8 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 paper, 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 but also 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:

- 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;

[1] we counted SLOC using `coqwc`.

- CompCertM: the latest version of 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*;
- `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;
- 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.

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 [Neis et al. 2015] with a slight generalization needed for CompCert (see §2.5).

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 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 §3. 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).

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

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.

The three applications, `Unreadglob`, `mutual-sum` and verification of `utod`, show the flexibility of our framework: allowing arbitrary memory relations and mathematical specification modules. In particular, to the best of our knowledge, our work is the first verification, in the context of CompCert, that reasons about module-local static variables with private invariants that can be modified across external function calls (due to mutual dependence between multiple modules) .

The Coq development is available at:

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

The remainder of the paper is structured as follows. We give a high-level overview of the main ideas in §2-§4; the main results of CompCertM and an analysis of its development in §5; its formal details in §6-§7; and a comparison to related work in §8.

2 VERIFICATION TECHNIQUES

We review the notions of closed and open simulations (§2.1), discuss the problems with open simulations (§2.2) and present our solution (§2.3). We also discuss the memory relations used by CompCertM in §2.4 and present mixed simulations in §2.5.

2.1 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 $\mathsf{Beh}(P)$ (seen as a specification), includes that of its compiled target program Q, *i.e.*, $\mathsf{Beh}(Q)$ (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[3]. Since CompCert mostly uses forward simulations, we will also focus on forward ones throughout the paper and discuss how to mix forward and backward simulations to support forward reasoning even when the target language is not deterministic in §2.5.

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):

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

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, `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 [Hur et al. 2012; Kang et al. 2015; Neis et al. 2015]. 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 §2.4 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 Fig. 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

[3]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** $\mathtt{at\_external}\ s_{\mathtt{src}}$ **with**
3: | $\mathsf{Some}\ (f_{\mathtt{src}}, \vec{v}_{\mathtt{src}}) \Rightarrow$
4: $\exists f_{\mathtt{tgt}}, \vec{v}_{\mathtt{tgt}},\ \mathtt{at\_external}\ s_{\mathtt{tgt}} = \mathsf{Some}\ (f_{\mathtt{tgt}}, \vec{v}_{\mathtt{tgt}}) \wedge$
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: $\forall w' \sqsupseteq w,\ \forall (m'_{\mathtt{src}}, m'_{\mathtt{tgt}}) \in \mathtt{mrel}(w'),\ \forall (r_{\mathtt{src}}, r_{\mathtt{tgt}}) \in \mathtt{vrel}(w'),$
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: | $\mathsf{None} \Rightarrow$
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**

Fig. 1. Structured (or, open) simulations (simplified for presentation purposes)

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 $\mathsf{Beh}(M) \supseteq \mathsf{Beh}(M')$.

2.2 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 [Neis et al. 2015; Patterson and Ahmed 2019]. 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* [Stewart et al. 2015].

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).

2.3 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 [Stewart et al. 2015] (see §8 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[4], denoted $\mathsf{Beh}(p)$ for $p \in \text{Module}$, a linking operation $\oplus$ between modules that is associative[5], and the identity (*i.e.*, empty) module $\mathtt{id} \in \text{Module}$:

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

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 §3.1 and §6 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 $p, p', q, q' \in \text{Module}$,

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

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

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

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 $\mathcal{R}$.

[4]Behaviors just need to be defined for closed programs. Technically, we can give undefined behavior (UB) to open modules.
[5]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.

Theorem 2.1 (Properties of RUSC). *The RUSC relation $\succcurlyeq_{\mathcal{R}}$ satisfies the following key properties.*

$$\begin{array}{lrl}
\textit{(Inclusion)} & \forall R \in \mathcal{R}, & R \subseteq \succcurlyeq_{\mathcal{R}} \\
\textit{(Adequacy)} & \forall p, p' \in \textit{Module}, & p \succcurlyeq_{\mathcal{R}} p' \implies \mathsf{Beh}(p) \supseteq \mathsf{Beh}(p') \\
\textit{(VerComp)} & \forall p, p', p'' \in \textit{Module}, & p \succcurlyeq_{\mathcal{R}} p' \wedge p' \succcurlyeq_{\mathcal{R}} p'' \implies p \succcurlyeq_{\mathcal{R}} p'' \\
\textit{(HorComp)} & \forall p, p', q, q' \in \mathrm{Self}(\mathcal{R}), & p \succcurlyeq_{\mathcal{R}} p' \wedge q \succcurlyeq_{\mathcal{R}} q' \implies p \oplus q \succcurlyeq_{\mathcal{R}} p' \oplus q' \\
\textit{(SelfComp)} & \forall p, q \in \mathrm{Self}(\mathcal{R}), & p \oplus q \in \mathrm{Self}(\mathcal{R})
\end{array}$$

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

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 §7.1 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 §4 for details).

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 §7 for details).

Applications. In our paper, we use RUSC in two situations: compiler and program verification.

First, we give an abstract example 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` → `a.il1` → `a.asm`

and $\mathtt{b.c} \rightarrow \mathtt{b.il2} \rightarrow \mathtt{b.il3} \rightarrow \mathtt{b.asm}$, each of which is verified using a different relation as follows:

$$(\mathtt{a.c}, \mathtt{a.il1}) \in R_1 \quad (\mathtt{a.il1}, \mathtt{a.asm}) \in R_2$$
$$(\mathtt{b.c}, \mathtt{b.il2}) \in R_3 \quad (\mathtt{b.il2}, \mathtt{b.il3}) \in R_4 \quad (\mathtt{b.il3}, \mathtt{b.asm}) \in R_5$$

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:

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

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

- $\mathtt{a.c} \succcurlyeq_{\mathcal{R}} \mathtt{a.asm}$ and $\mathtt{b.c} \succcurlyeq_{\mathcal{R}} \mathtt{b.asm}$ by Inclusion and VerComp of Theorem 2.1;
- $\mathtt{c.asm} \succcurlyeq_{\mathcal{R}} \mathtt{c.asm}$ since $(\mathtt{c.asm}, \mathtt{c.asm}) \in R_1 \subseteq \succcurlyeq_{\mathcal{R}}$ by Inclusion of Theorem 2.1;
- $\mathtt{a.c} \oplus \mathtt{b.c} \oplus \mathtt{c.asm} \succcurlyeq_{\mathcal{R}} \mathtt{a.asm} \oplus \mathtt{b.asm} \oplus \mathtt{c.asm}$ by HorComp of Theorem 2.1;
- $\mathtt{Beh}(\mathtt{a.c} \oplus \mathtt{b.c} \oplus \mathtt{c.asm}) \supseteq \mathtt{Beh}(\mathtt{a.asm} \oplus \mathtt{b.asm} \oplus \mathtt{c.asm})$ by Adequacy of Theorem 2.1.

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[6] 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 §4), 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

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

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

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

as long as the mathematical specification modules `a.spec`, `b.spec`, `c.spec`, `abc.spec` are in $\mathrm{Self}(\mathcal{R})$, 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 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.

[6]Clight is taken as the source language in most verification projects using CompCert such as VST [Appel 2011], CertiKOS and even CompComp. However, we also prove behavioral refinement w.r.t. the C source language (see §5).

2.4 Memory Relations of CompCertM

CompCertM uses the original memory identity and extension of CompCert (§2.4.1) 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 (§2.4.2). 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 (§2.4.3).

2.4.1 Memory Relations of CompCert. CompCert's memory model consists of a finite set of blocks of finite size and a pointer value (or, an address) is a pair (b, o) of a block id b and an offset o inside it. 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[7] 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.

2.4.2 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 [Dreyer et al. 2010; Hur et al. 2012].

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

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

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

Though private transitions are allowed before and after an external function call (*i.e.*, $w' \sqsupseteq_{\mathsf{prv}} w$ and $w''' \sqsupseteq_{\mathsf{prv}} w''$), the overall transition should be *public* (*i.e.*, $w''' \sqsupseteq w$) assuming the external call also makes a public transition (*i.e.*, $w'' \sqsupseteq w'$).[8]

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 $(\iota, m_{\mathsf{src}}^{\mathsf{prv}}, m_{\mathsf{tgt}}^{\mathsf{prv}})$ 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 $m_{\mathsf{src}}^{\mathsf{prv}}$ and that of the target $m_{\mathsf{tgt}}^{\mathsf{prv}}$ where $m_{\mathsf{src}}^{\mathsf{prv}}$ and $m_{\mathsf{tgt}}^{\mathsf{prv}}$ should be disjoint from the public memories specified by ι. Then, private transitions from $(\iota, m_{\mathsf{src}}^{\mathsf{prv}}, m_{\mathsf{tgt}}^{\mathsf{prv}})$ to $(\iota', m'^{\mathsf{prv}}_{\mathsf{src}}, m'^{\mathsf{prv}}_{\mathsf{tgt}})$ only require that ι' should extend ι, while public transitions additionally require that private memories should be unchanged (*i.e.*, $m_{\mathsf{src}}^{\mathsf{prv}} = m'^{\mathsf{prv}}_{\mathsf{src}}$ and $m_{\mathsf{tgt}}^{\mathsf{prv}} = m'^{\mathsf{prv}}_{\mathsf{tgt}}$). Note that all the areas of the source and target memories that are not on $m_{\mathsf{src}}^{\mathsf{prv}}$, $m_{\mathsf{tgt}}^{\mathsf{prv}}$ or the injection map ι are considered as *private* (*i.e.*, dynamic local) memory of the current module.

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 block for both `a[0]` and `a[1]` is allocated and the address of the block escapes to `g`.

```
    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];
    }                        }
```

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 $(m_{\mathsf{src}}, m_{\mathsf{tgt}})$ 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 $w_1.m_{\mathsf{tgt}}^{\mathsf{prv}}$ 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 $w_2.m_{\mathsf{tgt}}^{\mathsf{prv}} = w_3.m_{\mathsf{tgt}}^{\mathsf{prv}}$, 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 $w_3.m_{\mathsf{tgt}}^{\mathsf{prv}}$. Since $w_4.m_{\mathsf{tgt}}^{\mathsf{prv}} = w_1.m_{\mathsf{tgt}}^{\mathsf{prv}}$, 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 $w.m_{\mathsf{src}}^{\mathsf{prv}}$ and $w.m_{\mathsf{tgt}}^{\mathsf{prv}}$ 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

[8] We only allow private transitions just before and after external calls for simplicity. See §8 for comparison with [Dreyer et al. 2010; Hur et al. 2012].

`a[1] = 0` preserves $w_1.m^{\mathsf{prv}}_{\mathsf{tgt}}$ (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.*, $m^{\mathsf{prv}}_{\mathsf{src}}$ and $m^{\mathsf{prv}}_{\mathsf{tgt}}$) cannot be converted to 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 $m^{\mathsf{prv}}_{\mathsf{src}}$ and $m^{\mathsf{prv}}_{\mathsf{tgt}}$).

2.4.3 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 $w.m^{\mathsf{prv}}_{\mathsf{src}}$, $w.m^{\mathsf{prv}}_{\mathsf{tgt}}$ and $w.\iota$. We give examples using module-local invariants in §4.

2.5 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 (see §3) 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 [Neis et al. 2015]. 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 $(ms_{\mathsf{src}}, ms_{\mathsf{tgt}}) \in R$,

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

(2) $\forall e, ms'_{\mathsf{src}},\ ms_{\mathsf{src}} \overset{e}{\hookrightarrow} ms'_{\mathsf{src}} \implies \exists ms'_{\mathsf{tgt}},\ ms_{\mathsf{tgt}} \overset{\tau}{\circlearrowright}^{*} \overset{e}{\circlearrowright} \overset{\tau}{\circlearrowright}^{*} ms'_{\mathsf{tgt}} \wedge (ms'_{\mathsf{src}}, ms'_{\mathsf{tgt}}) \in R$

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

Fig. 2 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_{src} is receptive and
$\forall e, ms'_{\mathsf{src}},\ ms_{\mathsf{src}} \overset{e}{\hookrightarrow} ms'_{\mathsf{src}} \implies \exists ms'_{\mathsf{tgt}},\ ms_{\mathsf{tgt}} \overset{\tau}{\bullet}^{*} \overset{e}{\bullet} \overset{\tau}{\bullet}^{*} ms'_{\mathsf{tgt}} \wedge (ms'_{\mathsf{src}}, ms'_{\mathsf{tgt}}) \in R$
where $ms \overset{e}{\bullet} ms'$ denotes that ms is locally determinate and $ms \overset{e}{\hookrightarrow} ms'$.

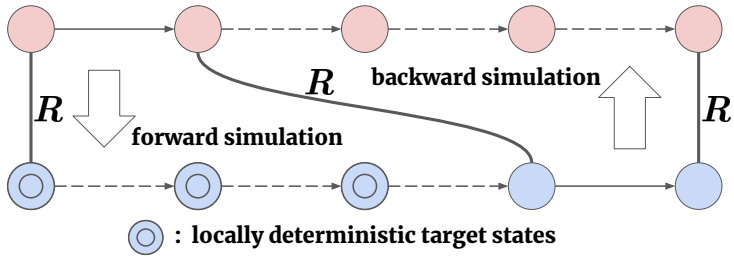

Fig. 2. A visualized example of mixed simulations

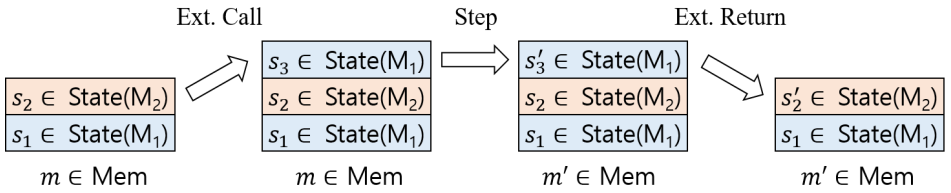

Fig. 3. An execution of interaction semantics

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

3 REPAIRED INTERACTION SEMANTICS

We briefly review interaction semantics (§3.1), discuss the problems (§3.2) and present our solutions (§3.3).

3.1 Background

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*) State(M) with the following operations:

- `init_core`: given a function f with arguments $\vec{v}$, gives the initial module state $s \in$ State(M) executing the invoked function f with $\vec{v}$.
- `at_external`: given $s \in$ State(M), checks if an external function f is called with arguments $\vec{v}$.
- `after_external`: given $s \in$ 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 $s \in$ State(M), checks if the module execution is halted with a return value r.
- `corestep`: given $s \in$ 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 Fig. 3, 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 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.

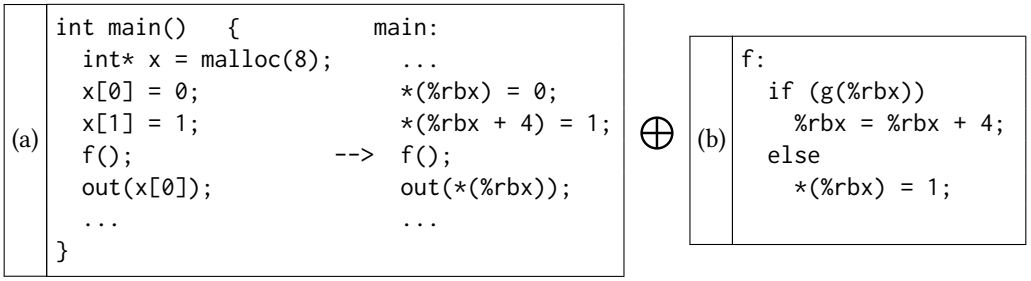

Fig. 4. A counterexample showing the problem with the assumptions on the registers

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.

3.2 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 $\mathsf{Beh}(M_1 \oplus \ldots \oplus M_n)$ should *be included in* those according to the physical semantics $\mathsf{Beh}(M_1 \circ \ldots \circ M_n)$. 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 $\mathsf{Beh}(M_1 \oplus \ldots \oplus M_n)$ should *include* those according to the physical semantics $\mathsf{Beh}(M_1 \circ \ldots \circ M_n)$. 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).

3.3 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.

3.3.1 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 Fig. 4 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) 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.

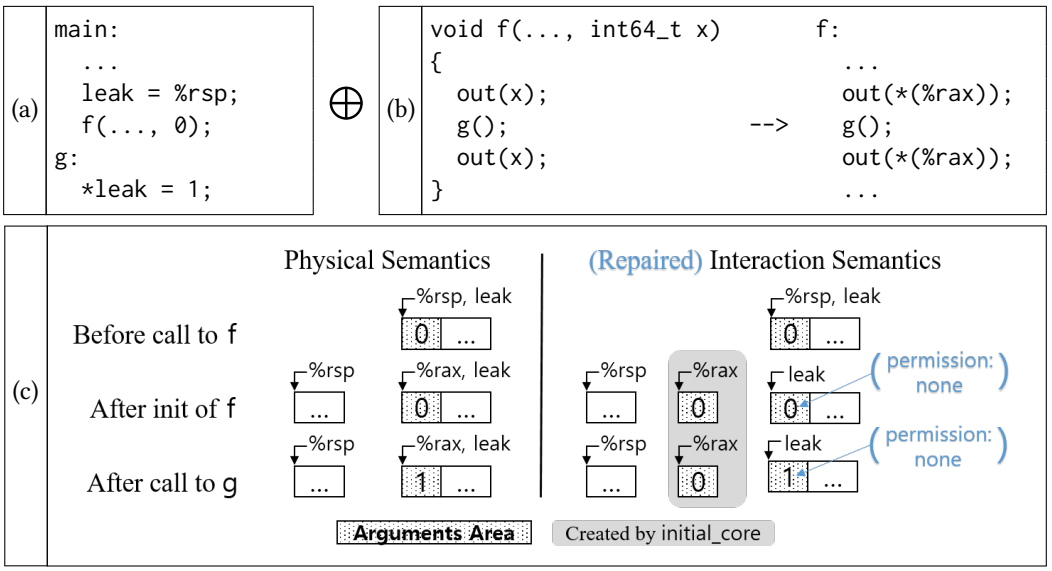

Fig. 5. A counterexample showing the problem with the assumption on the stack

Finally, note that introducing nondeterminism as above is not a showstopper thanks to the mixed simulation, as discussed in §2.5: we can do forward simulation everywhere except for the `init_core` step of assembly modules, where we should do backward simulation.

3.3.2 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 Fig. 5 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

```
static int x = 0;          static int x = 0;                    |  static int y = 0;
int f() {                  int f() {                int f() {   |  void g() {
  g();            [CP]       g();          [UG]      g();      |    if (y == 0) {
  x = 1;         ----->      x = 1;        ----->               |      y = 1; f();
  return x;                  return 1;                 return 1; |    }
}                          }                        }           |  }
```

Fig. 6. An example of Unreadglob optimization

the language-independent design. For example, consider the interaction semantics of the above example, depicted in the right-hand-side (RHS) of Fig. 5 (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.

4 MODULE-LOCAL INVARIANTS AND SPECIFICATION MODULES

In §2 we presented how to achieve compositional compiler correctness in our framework. In this section we present what our framework additionally offers about compiler and program verification: verifying more advanced compiler optimizations with module-local invariants (§4.1) and verifying program modules against their mathematical specification modules (§4.2 and §4.3). To the best of our knowledge, our framework is the first, in the context of CompCert, that is capable of verifying the *mutually recursive* example presented in §4.2.

4.1 Advanced Optimizations with Module-Local Invariants

We developed a new optimization Unreadglob eliminating all unread static variables and instructions writing to them. Fig. 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 Fig. 6.

In verification of the optimization UG above, we have to use memory injections w with module-local invariants introduced in §2.4.3. 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 $w.m_{\mathsf{src}}^{\mathsf{prv}}$ 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.

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 $\mathbf{X} = \mathsf{g}, \mathbf{Y} = \mathsf{f}$

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

ab.spec

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

Fig. 7. The `mutual-sum` example

4.2 Verification against Specification Modules

Fig. 7 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 $\mathrm{sum}(\mathtt{i}-1)$, 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 $\mathrm{sum}(\mathtt{i}-1)$.

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` sum(i) only when the return value from the external call `g`($i - 1$) is sum($i - 1$), otherwise UB, which means that this module gives a conditional specification under the assumption that `g`(i) returns sum(i); `halted` recognizes `Ret` r as the halted state returning r; and finally `step` transitions from `Init` i to either `Ret` 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 `a.spec` $\succcurlyeq_{\mathcal{R}}$ `a.c` using memory injections with the following invariant:

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

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

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

Finally, we prove `ab.spec` $\succcurlyeq_{\mathcal{R}}$ `a.spec` $\oplus$ `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).

4.3 Verification of utod

`__compcert_i64_utod` is one of the CompCert's internal handwritten assembly functions, which converts `unsigned long` to `double` by utilizing architecture-specific instructions like `cvtsi2sdq`. CompCert currently axiomatizes the behaviors of such runtime libraries as the following axiom.

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

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.

5 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 (§5.1) and evaluate its verification efforts (§5.2). CompCertM currently supports the x86 backend only. We do not currently see any technical problem with supporting other architectures.

5.1 Compositional Correctness

CompCertM uses open simulations with three parameters: memory relations, symbol relations and memory predicates (see §7.2 for details). It supports (*i*) the memory relations discussed in §2.4: 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 C transforms the source module with a series of passes that are independently verified using open simulations in $\mathcal{R}$.

Lemma 5.1 (Pass Correctness). *For any Clight module S and Asm module T, if $C(S) = T$, 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 5.2 (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 §3, every Asm module can be self-related.

From Lemmas 5.1 and 5.2, the RUSC relation for the compiler follows.

Theorem 5.3 (Modular Correctness). *For any Clight module S and Asm module T, if $C(S) = T$:*

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

This theorem provides a truly compositional correctness thanks to the compositionality of RUSC (Theorem 2.1): 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 5.4 (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., $C(S_i) = T_i$ 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*

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

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 5.2).

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 [Appel 2011], 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.[9]

Supporting CompCert C. However, we can still prove a compositional correctness (not modular correctness as in Theorem 5.3) for CompCert C following SepCompCert's *Level A* technique [Kang et al. 2016], 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 5.5 (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., 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*

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

By composing Corollary 5.4, Lemma 5.5 and Lemma 5.2, we have the following theorem.

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

Table 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	CompCert 3.0	CompCertX
Pass Proofs	34,376	35,893 (+4.41%)	4,923(+14.32%)	21,215	52,140 (+145.77%)	26,466	30,572 (+15.51%)
The Rest	85,617	87,965 (+2.74%)	25,558(+29.85%)	59,365	107,910 (+81.77%)	82,312	121,532 (+47.65%)
Total	119,993	123,858 (+3.22%)	30,481(+25.40%)	80,580	160,050 (+98.62%)	108,778	152,104 (+39.83%)

Table 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 3. SLOC of additional developments

Portion	`Unreadglob` 3.5	`Unreadglob` pack	`mutual-sum`	`utod`	Adequacy w.r.t. C
Pass Proofs	1,842	338	3,088	361	-
The Rest	260	1,933	2,707	424	4,044
Total	2,102	2,271	5,795	785	4,044

Theorem 5.6 (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., $C(S_i) = T_i$ 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*

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

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.

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

Theorem 5.7 (Adequacy w.r.t. Assembly). *Let $M_1, \cdots, M_n$ be Asm modules. We have:*

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

This theorem allows us to carry verification results on the interaction semantics such as Theorem 5.6 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.

Theorem 5.8 (Adequacy w.r.t C). *Let $M_1, \cdots, M_n$ be CompCert C modules. We have:*

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

By composing Theorems 5.6 to 5.8, we obtain the same separate compilation correctness result of SepCompCert [Kang et al. 2016]:

Corollary 5.9 (Separate Compilation Correctness). *Let $S_1, \ldots, S_n$ be CompCert C modules and $T_1, \ldots, T_n$ be Asm modules. If $C(S_i) = T_i$ for each i, we have:*

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

5.2 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 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.[10] Also note that CompCertR uses the enriched memory injections of §2.4.2 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 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 support multi-language linking. We counted SLOC reported by `coqwc`.[11] 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.[12]

Table 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 §3. 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 5.2); 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 5.7).

Table 3 shows SLOC for the new optimization pass and the verification examples given in the paper. 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. As the verification of `mutual-sum` and `utod` show, directly proving open simulation between programs and specifications is costly. We believe that program logics like VST [Appel 2011] can be used to prove such simulation, which could significantly reduce the verification cost.

6 FORMAL SEMANTICS

In this section we give a few interesting details of formal semantics: the loading of interaction semantics (§6.1) and a few tweaks we made for module semantics (§6.2).

6.1 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 Fig. 8.

[10] 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.

[11] 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 [Gu et al. 2015; Stewart et al. 2015; Wang et al. 2019].

[12] 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

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 [Kang et al. 2016].

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 loading process follows the original CompCert's.

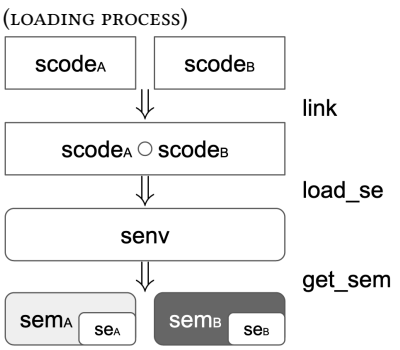

Fig. 8. Loading in Interaction Semantics

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.

6.2 Module Semantics

We briefly discuss the notions of module and module semantics presented in Fig. 9. To support loading described in §6.1, 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 §3.1.

- 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 §3.3.
- 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 §3.3.
- 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.

7 FORMALIZATION OF VERIFICATION TECHNIQUES

Now we present the formalization of our verification techniques. We parameterize the notion of open simulation presented in §2 with three parameters: memory relations, symbol relations, and memory predicates. We present the three parameters (§7.1), the parameterized open simulations (§7.2), and their horizontal compositionality and adequacy theorems (§7.3).

7.1 Parameters for Open Simulations

Fig. 10 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.

(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)

$sem \in \text{ModSem} =$

$\{\ \texttt{state} \in \text{Set}, \texttt{senv} \in \text{Senv}, \hookrightarrow \in \mathcal{P}((\text{Mem} \times \texttt{state}) \times \text{Event} \times (\text{Mem} \times \texttt{state})),$
$\texttt{init\_core} \in \text{CallData} \to \mathcal{P}(\text{Mem} \times \texttt{state}), \texttt{at\_external} \in (\text{Mem} \times \texttt{state}) \to \texttt{Option CallData},$
$\texttt{after\_external} \in (\texttt{state} \times \text{RetData}) \to \texttt{Option}\ (\text{Mem} \times \texttt{state}), \texttt{halted} \in (\text{Mem} \times \texttt{state}) \to \texttt{Option RetData} \mid$
$\{ms \mid \exists e, ms',\ ms \xhookrightarrow{e} ms'\}, \{ms \mid \exists c, \texttt{at\_external}(ms) = \texttt{Some}\ c\}, \{ms \mid \exists r, \texttt{halted}(ms) = \texttt{Some}\ r\}$ disjoint $\}$

$\text{CallData} \stackrel{\text{def}}{=} \{(\texttt{m}, \texttt{f}, \texttt{vs}) \in (\text{Mem} \times \text{Val} \times \overrightarrow{\text{Val}})\} \uplus \{(\texttt{m}, \texttt{f}, \texttt{rs}) \in (\text{Mem} \times \text{Val} \times (\text{Reg} \to \text{Val}))\}$

$\text{RetData} \stackrel{\text{def}}{=} \{(\texttt{m}, \texttt{v}) \in (\text{Mem} \times \text{Val})\} \uplus \{(\texttt{m}, \texttt{rs}) \in (\text{Mem} \times (\text{Reg} \to \text{Val}))\}$

Fig. 9. Module and Module Semantics

(MEMORY RELATION)

$\text{MR} \in \text{MemRel} =$

$\{\ (\texttt{t}, \sqsubseteq, \sqsubseteq_{\texttt{prv}}, \texttt{mrel}, \texttt{vrel}) \in (\text{Set} \times \mathcal{P}(\texttt{t} \times \texttt{t}) \times \mathcal{P}(\texttt{t} \times \texttt{t}) \times (\texttt{t} \to \mathcal{P}(\text{Mem} \times \text{Mem})) \times (\texttt{t} \to \mathcal{P}(\text{Val} \times \text{Val}))) \mid$
$(\sqsubseteq \text{ is preorder}) \wedge (\sqsubseteq \subseteq \sqsubseteq_{\texttt{prv}}) \wedge (\forall w, w',\ w \sqsubseteq w' \implies \texttt{vrel}(w) \subseteq \texttt{vrel}(w')) \wedge$
$(\forall w, i,\ (\text{Vint}\ i, v_{\texttt{tgt}}) \in \texttt{vrel}(w) \implies v_{\texttt{tgt}} = \text{Vint}\ i)\ \}$

$c_{\texttt{src}} \gtrsim_w c_{\texttt{tgt}} \stackrel{\text{def}}{=} (c_{\texttt{src}}.\texttt{m}, c_{\texttt{tgt}}.\texttt{m}) \in \texttt{mrel}(w) \wedge (c_{\texttt{src}}.\texttt{f}, c_{\texttt{tgt}}.\texttt{f}) \in \texttt{vrel}(w) \wedge (c_{\texttt{src}}.\texttt{vs}, c_{\texttt{tgt}}.\texttt{vs}) \in \overrightarrow{\texttt{vrel}(w)} \wedge$
$(c_{\texttt{src}}.\texttt{rs}, c_{\texttt{tgt}}.\texttt{rs}) \in \overrightarrow{\texttt{vrel}(w)}$

$r_{\texttt{src}} \gtrsim_w r_{\texttt{tgt}} \stackrel{\text{def}}{=} (r_{\texttt{src}}.\texttt{m}, r_{\texttt{tgt}}.\texttt{m}) \in \texttt{mrel}(w) \wedge (r_{\texttt{src}}.\texttt{v}, r_{\texttt{tgt}}.\texttt{v}) \in \texttt{vrel}(w) \wedge (r_{\texttt{src}}.\texttt{rs}, r_{\texttt{tgt}}.\texttt{rs}) \in \overrightarrow{\texttt{vrel}(w)}$

(SYMBOL RELATION)

$\text{SR} \in \text{SymbRel} =$

$\{(\texttt{t}, \sqsubseteq, \texttt{screl}, \texttt{serel}) \in (\text{Set} \times \mathcal{P}(\texttt{t} \times \texttt{t}) \times (\texttt{t} \to \mathcal{P}(\text{Scode} \times \text{Scode})) \times (\texttt{t} \to \text{MR}.\texttt{t} \to \mathcal{P}(\text{Senv} \times \text{Senv}))) \mid$

(1) $\sqsubseteq$ is preorder

(2) $\forall sc_{\texttt{src}}, sc'_{\texttt{src}}, sc''_{\texttt{src}}, sc_{\texttt{tgt}}, sc'_{\texttt{tgt}}, sc''_{\texttt{tgt}},\ sc''_{\texttt{src}} = sc_{\texttt{src}} \circ sc'_{\texttt{src}} \wedge sc''_{\texttt{tgt}} = sc_{\texttt{tgt}} \circ sc'_{\texttt{tgt}} \implies$
$\forall d, d',\ (sc_{\texttt{src}}, sc_{\texttt{tgt}}) \in \texttt{screl}(d) \wedge (sc'_{\texttt{src}}\ sc'_{\texttt{tgt}}) \in \texttt{screl}(d') \implies$
$\exists d'',\ (sc''_{\texttt{src}}, sc''_{\texttt{tgt}}) \in \texttt{screl}(d'') \wedge d \sqsubseteq d'' \wedge d' \sqsubseteq d''$

(3) $\forall sc_{\texttt{src}}, sc_{\texttt{tgt}}, d,\ (sc_{\texttt{src}}, sc_{\texttt{tgt}}) \in \texttt{screl}(d) \implies$
$\exists w,\ (\texttt{load\_mem}(sc_{\texttt{src}}), \texttt{load\_mem}(sc_{\texttt{tgt}})) \in \texttt{mrel}(w) \wedge (\texttt{load\_se}(sc_{\texttt{src}}), \texttt{load\_se}(sc_{\texttt{tgt}})) \in \texttt{serel}(d, w)$

(4) $\forall d, w, w',\ w \sqsubseteq_{\texttt{prv}} w' \implies \texttt{serel}(d, w) \subseteq \texttt{serel}(d, w')$

(5) $\forall d, w, se_{\texttt{src}}, se_{\texttt{tgt}},\ (se_{\texttt{src}}, se_{\texttt{tgt}}) \in \texttt{serel}(d, w) \implies se_{\texttt{src}}.\texttt{pubs} = se_{\texttt{tgt}}.\texttt{pubs} \wedge$
$\forall (v_{\texttt{src}}, v_{\texttt{tgt}}) \in \text{MR}.\texttt{vrel}(w),\ v_{\texttt{src}} \in \texttt{ftns}(se_{\texttt{src}}) \implies v_{\texttt{tgt}} \in \texttt{ftns}(se_{\texttt{tgt}})$

(6) $\forall d, d', w, sc_{\texttt{src}}, sc_{\texttt{tgt}}, se_{\texttt{src}}, se_{\texttt{tgt}},\ d \sqsubseteq d' \wedge (sc_{\texttt{src}}, sc_{\texttt{tgt}}) \in \texttt{screl}(d) \wedge (se_{\texttt{src}}, se_{\texttt{tgt}}) \in \texttt{serel}(d', w) \implies$
$(se_{\texttt{src}}|_{sc_{\texttt{src}}}, se_{\texttt{tgt}}|_{sc_{\texttt{tgt}}}) \in \texttt{serel}(d, w)$

(7) $\forall d, w, se_{\texttt{src}}, se_{\texttt{tgt}}, c_{\texttt{src}}, c_{\texttt{tgt}},\ (se_{\texttt{src}}, se_{\texttt{tgt}}) \in \texttt{serel}(d, w) \wedge c_{\texttt{src}} \gtrsim_w c_{\texttt{tgt}} \implies$
$\forall e, r_{\texttt{src}},\ \text{external\_call}\ se_{\texttt{src}}\ c_{\texttt{src}}\ e\ r_{\texttt{src}} \implies \exists r_{\texttt{tgt}},\ \text{external\_call}\ se_{\texttt{tgt}}\ c_{\texttt{tgt}}\ e\ r_{\texttt{tgt}} \wedge \exists w' \sqsupseteq w,\ r_{\texttt{src}} \gtrsim_{w'} r_{\texttt{tgt}}\ \}$

(MEMORY PREDICATE)

$\text{MP} \in \text{MemPred} =$

$\{\ (\texttt{t}, \sqsubseteq, \sqsubseteq_{\texttt{prv}}, \texttt{mpred}, \texttt{vpred}, \texttt{sepred}) \in (\text{Set} \times \mathcal{P}(\texttt{t} \times \texttt{t}) \times \mathcal{P}(\texttt{t} \times \texttt{t}) \times (\texttt{t} \to \mathcal{P}(\text{Mem})) \times (\texttt{t} \to \mathcal{P}(\text{Val})) \times (\texttt{t} \to \mathcal{P}(\text{Senv}))) \mid$
$(\sqsubseteq \text{ is preorder}) \wedge (\sqsubseteq \subseteq \sqsubseteq_{\texttt{prv}}) \wedge (\forall u, u',\ u \sqsubseteq u' \implies \texttt{vpred}(u) \subseteq \texttt{vpred}(u')) \wedge$
(sepred should satisfy the unary version of serel's conditions where SR.$\sqsubseteq$ and screl are the total relations) $\}$

$\texttt{cpred}(u) \stackrel{\text{def}}{=} \{c \in \text{CallData} \mid c.\texttt{m} \in \texttt{mpred}(u) \wedge c.\texttt{f} \in \texttt{vpred}(u) \wedge c.\texttt{vs} \in \overrightarrow{\texttt{vpred}(u)} \wedge c.\texttt{rs} \in \overrightarrow{\texttt{vpred}(u)}\}$

$\texttt{rpred}(u) \stackrel{\text{def}}{=} \{r \in \text{RetData} \mid r.\texttt{m} \in \texttt{mpred}(u) \wedge r.\texttt{v} \in \texttt{vpred}(u) \wedge r.\texttt{rs} \in \overrightarrow{\texttt{vpred}(u)}\}$

Fig. 10. Three parameters for open simulations

Memory Relation. The first parameter ranges over Kripke-style memory/value relations in MR. Following [Dreyer et al. 2010; Hur et al. 2012], 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 $\mathtt{t}$ of possible worlds; (ii) *public* and *private* transition relations $\sqsubseteq$ and $\sqsubseteq_{\mathtt{prv}}$ over the worlds; and (iii) for each world $w \in \mathtt{t}$, memory relation $\mathtt{mrel}(w)$ and value relation $\mathtt{vrel}(w)$. 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 §2.4. 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 $\mathtt{t}$ of symbol relation states; (ii) an extension relation $\sqsubseteq$ on the states; (iii) for each state d, a (compile-time) symbol code relation $\mathtt{screl}(d)$; and (iv) for each state d and world $w \in \mathrm{MR}.\mathtt{t}$, (run-time) symbol environment relation $\mathtt{serel}(d, w)$. There are seven well-formedness conditions: (1) the extension relation $\sqsubseteq$ is transitive and reflexive; (2) $\mathtt{screl}$ is closed under the syntactic linking; (3) if symbol codes are related by $\mathtt{screl}$, then the initial memories and symbol environments loaded by load_mem and load_se are related by $\mathtt{mrel}$ and $\mathtt{serel}$, respectively; (4) $\mathtt{serel}$ is monotone w.r.t. private transitions; (5) for symbol environments related by $\mathtt{serel}$, their public symbols are identical and their functions have the same signatures; (6) $\mathtt{serel}$ 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 §7.2). MP is essentially a unary version of MR combined with SR where SR.$\sqsubseteq$ and $\mathtt{screl}$ are taken as the total relations (*i.e.*, relating everything): it consists of (i) the set $\mathtt{t}$ of possible worlds; (ii) public and private transition relations $\sqsubseteq$ and $\sqsubseteq_{\mathtt{prv}}$ over the worlds, respectively; and (iii) for each world $w \in \mathtt{t}$, a memory predicate $\mathrm{mpred}(w)$, a value predicate $\mathrm{vpred}(w)$, and a symbol environment predicate $\mathrm{sepred}(w)$. The well-formedness conditions are self-explanatory.

7.2 Open Simulations with Parameters

Fig. 11 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 §2.5. In this section, we omit MR, SR, and MP when clear from context (*e.g.*, $\mathtt{vrel}(w)$ for $\mathrm{MR}.\mathtt{vrel}(w)$). Also, $\boxed{\mathrm{R}}$ and $\boxed{\mathrm{G}}$ 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 $sem_{\mathtt{src}}$, $sem_{\mathtt{tgt}}$ and a (source) soundness predicate *sound_state* (discussed later), the relation *match_states* over worlds is an open simulation if the relatedness of $ms_{\mathtt{src}}$ and $ms_{\mathtt{tgt}}$ at a world w with the soundness of $ms_{\mathtt{src}}$ 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
  line 4: the next states are related by $match\_states(w')$ for a public future world $w' \sqsupseteq w$.

(SIM:STATES)

$match\_states \in \mathrm{open\_sim}(sem_{\mathrm{src}}, sem_{\mathrm{tgt}}, sound\_state) \stackrel{\mathrm{def}}{=}$
$\forall w, \forall((m_{\mathrm{src}}, s_{\mathrm{src}}), (m_{\mathrm{tgt}}, s_{\mathrm{tgt}})) \in match\_states(w), \boxed{(\exists u,\ (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sound\_state(u))} \implies$
(STEP) $sem_{\mathrm{src}}.\mathtt{at\_external}(m_{\mathrm{src}}, s_{\mathrm{src}}) = \mathrm{None} \wedge sem_{\mathrm{src}}.\mathtt{halted}(m_{\mathrm{src}}, s_{\mathrm{src}}) = \mathrm{None} \wedge$
$\forall e, m'_{\mathrm{src}}, s'_{\mathrm{src}},\ (m_{\mathrm{src}}, s_{\mathrm{src}}) \xhookrightarrow{e} (m'_{\mathrm{src}}, s'_{\mathrm{src}}) \implies$
$\exists m'_{\mathrm{tgt}}, s'_{\mathrm{tgt}},\ (m_{\mathrm{tgt}}, s_{\mathrm{tgt}}) \xhookrightarrow{\tau}{}^{*} \xhookrightarrow{e} \xhookrightarrow{\tau}{}^{*} (m'_{\mathrm{tgt}}, s'_{\mathrm{tgt}}) \wedge$
$\boxed{\exists w' \sqsupseteq w}, ((m'_{\mathrm{src}}, s'_{\mathrm{src}}), (m'_{\mathrm{tgt}}, s'_{\mathrm{tgt}})) \in match\_states(w')$
$\vee$ (CALL) $\boxed{\exists w' \sqsupseteq_{\mathrm{prv}} w}, \exists c_{\mathrm{src}}, c_{\mathrm{tgt}}, \boxed{c_{\mathrm{src}} \gtrsim_{w'} c_{\mathrm{tgt}}} \wedge$
$sem_{\mathrm{src}}.\mathtt{at\_external}(m_{\mathrm{src}}, s_{\mathrm{src}}) = \mathrm{Some}\ c_{\mathrm{src}} \wedge sem_{\mathrm{tgt}}.\mathtt{at\_external}(m_{\mathrm{tgt}}, s_{\mathrm{tgt}}) = \mathrm{Some}\ c_{\mathrm{tgt}} \wedge$
$\boxed{\forall w'' \sqsupseteq w'}, \forall r_{\mathrm{src}}, r_{\mathrm{tgt}}, \boxed{r_{\mathrm{src}} \gtrsim_{w''} r_{\mathrm{tgt}}} \implies$
$\forall m'_{\mathrm{src}}, s'_{\mathrm{src}},\ sem_{\mathrm{src}}.\mathtt{after\_external}(s_{\mathrm{src}}, r_{\mathrm{src}}) = \mathrm{Some}\ (m'_{\mathrm{src}}, s'_{\mathrm{src}}) \implies$
$\exists m'_{\mathrm{tgt}}, s'_{\mathrm{tgt}},\ sem_{\mathrm{tgt}}.\mathtt{after\_external}(s_{\mathrm{tgt}}, r_{\mathrm{tgt}}) = \mathrm{Some}\ (m'_{\mathrm{tgt}}, s'_{\mathrm{tgt}}) \wedge$
$\boxed{\exists w''' \sqsupseteq_{\mathrm{prv}} w'',\ w''' \sqsupseteq w} \wedge ((m'_{\mathrm{src}}, s'_{\mathrm{src}}), (m'_{\mathrm{tgt}}, s'_{\mathrm{tgt}})) \in match\_states(w''')$
$\vee$ (RET) $\boxed{\exists w' \sqsupseteq w}, \exists r_{\mathrm{src}}, r_{\mathrm{tgt}}, \boxed{r_{\mathrm{src}} \gtrsim_{w'} r_{\mathrm{tgt}}} \wedge$
$sem_{\mathrm{src}}.\mathtt{halted}(m_{\mathrm{src}}, s_{\mathrm{src}}) = \mathrm{Some}\ r_{\mathrm{src}} \wedge sem_{\mathrm{tgt}}.\mathtt{halted}(m_{\mathrm{tgt}}, s_{\mathrm{tgt}}) = \mathrm{Some}\ r_{\mathrm{tgt}}$

(SIM:MODSEM)

$sem_{\mathrm{src}} \gtrsim_{d, sound\_state} sem_{\mathrm{tgt}} \stackrel{\mathrm{def}}{=} \exists match\_states \in \mathrm{open\_sim}(sem_{src}, sem_{tgt}, sound\_state),$
(INIT) $\forall w \in \mathrm{MR}.\mathtt{t},\ \forall c_{\mathrm{src}}, c_{\mathrm{tgt}}, \boxed{c_{\mathrm{src}} \gtrsim_{w} c_{\mathrm{tgt}}} \implies$
$c_{\mathrm{src}}.\mathtt{f} \in \mathrm{ftns}(sem_{\mathrm{src}}.\mathtt{senv}) \wedge c_{\mathrm{tgt}}.\mathtt{f} \in \mathrm{ftns}(sem_{\mathrm{tgt}}.\mathtt{senv}) \implies$
$\boxed{(sem_{src}.\mathtt{senv}, sem_{tgt}.\mathtt{senv}) \in \mathtt{serel}(d, w)} \implies \forall (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sem_{\mathrm{src}}.\mathtt{init\_core}(c_{\mathrm{src}}),$
$\exists (m_{\mathrm{tgt}}, s_{\mathrm{tgt}}) \in sem_{\mathrm{tgt}}.\mathtt{init\_core}(c_{\mathrm{tgt}}),$
$\boxed{\exists w' \sqsupseteq w},\ ((m_{\mathrm{src}}, s_{\mathrm{src}}), (m_{\mathrm{tgt}}, s_{\mathrm{tgt}})) \in match\_states(w')$

(SIM:MOD)

$M_{\mathrm{src}} \gtrsim M_{\mathrm{tgt}} \stackrel{\mathrm{def}}{=} \exists d \in \mathrm{SR}.\mathtt{t},\ \exists sound\_state : \mathrm{MP}.\mathtt{t} \to \mathcal{P}(\mathrm{Mem} \times M_{\mathrm{src}}.\mathtt{state}),$
(1) $\boxed{(M_{\mathrm{src}}.\mathtt{scode}, M_{\mathrm{tgt}}.\mathtt{scode}) \in \mathtt{screl}(d)}$
$\wedge$ (2) $\boxed{\forall se_{\mathrm{src}},\ sound\_state \in \mathrm{open\_prsv}(M_{\mathrm{src}}.\mathtt{sem}\ se_{\mathrm{src}})}$
$\wedge$ (3) $\forall d' \sqsupseteq d,\ \forall w,\ \boxed{\forall (se_{\mathrm{src}}, se_{\mathrm{tgt}}) \in \mathtt{serel}(d', w)},$
$M_{\mathrm{src}}.\mathtt{sem}(se_{\mathrm{src}}) \gtrsim_{d, sound\_state} M_{\mathrm{tgt}}.\mathtt{sem}(se_{\mathrm{tgt}})$

(SIM:PROG)

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

(PRESERVATION)

$sound\_state \in \mathrm{open\_prsv}(sem_{\mathrm{src}}) \stackrel{\mathrm{def}}{=}$
(INIT) $\forall u \in \mathrm{MP}.\mathtt{t}, \boxed{\forall c_{\mathrm{src}} \in \mathtt{cpred}(u)}, \boxed{sem_{\mathrm{src}}.\mathtt{senv} \in \mathtt{sepred}(u)} \implies \forall (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sem_{\mathrm{src}}.\mathtt{init\_core}(c_{\mathrm{src}}),$
$\boxed{\exists u' \sqsupseteq u},\ (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sound\_state(u')$
$\wedge$ (STEP) $\forall u,\ \forall (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sound\_state(u), \forall e, m'_{\mathrm{src}}, s'_{\mathrm{src}},\ (m_{\mathrm{src}}, s_{\mathrm{src}}) \xhookrightarrow{e} (m'_{\mathrm{src}}, s'_{\mathrm{src}}) \implies$
$\boxed{\exists u' \sqsupseteq u},\ (m'_{\mathrm{src}}, s'_{\mathrm{src}}) \in sound\_state(u')$
$\wedge$ (CALL) $\forall u,\ \forall (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sound\_state(u), \forall c_{\mathrm{src}},\ sem_{\mathrm{src}}.\mathtt{at\_external}(m_{\mathrm{src}}, s_{\mathrm{src}}) = \mathrm{Some}\ c_{\mathrm{src}} \implies$
$\boxed{\exists u' \sqsupseteq_{\mathrm{prv}} u}, \boxed{c_{\mathrm{src}} \in \mathtt{cpred}(u')} \wedge$
$\boxed{\forall u'' \sqsupseteq u',\ \forall r_{\mathrm{src}} \in \mathtt{rpred}(u'')},\ \forall m'_{\mathrm{src}}, s'_{\mathrm{src}},\ sem_{\mathrm{src}}.\mathtt{after\_external}(s_{\mathrm{src}}, r_{\mathrm{src}}) = \mathrm{Some}\ (m'_{\mathrm{src}}, s'_{\mathrm{src}}) \implies$
$\boxed{\exists u''' \sqsupseteq_{\mathrm{prv}} u'',\ u''' \sqsupseteq u \wedge \exists m'_{\mathrm{src}} \in \mathtt{mpred}(u''')} \wedge (m'_{\mathrm{src}}, s'_{\mathrm{src}}) \in sound\_state(u''')$
$\wedge$ (RET) $\forall u,\ \forall (m_{\mathrm{src}}, s_{\mathrm{src}}) \in sound\_state(u), \forall r_{\mathrm{src}},\ sem_{\mathrm{src}}.\mathtt{halted}(m_{\mathrm{src}}, s_{\mathrm{src}}) = \mathrm{Some}\ r_{\mathrm{src}} \implies$
$\boxed{\exists u' \sqsupseteq u},\ \boxed{r_{\mathrm{src}} \in \mathtt{rpred}(u')}$

Fig. 11. Parameterized Open Simulations

- (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 $w' \sqsupseteq_{\mathtt{prv}} w$, 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 $d \in$ SR and a (source) soundness predicate $sound\_state$, a target module semantics $sem_{\mathtt{tgt}}$ simulates a source one $sem_{\mathtt{src}}$ if for an open simulation $match\_states$:

- (INIT) the initial machine states of $sem_{\mathtt{src}}$ and $sem_{\mathtt{tgt}}$ are related by $match\_states$. Specifically:
  line 1: for any source and target call data related at any world $w \in$ 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 $M_{\mathtt{tgt}}$ simulates a source one $M_{\mathtt{src}}$ 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 uses a relation $match\_states$ to prove correctness of a translation pass and a predicate $sound\_state$ to prove correctness of the analyzer performing 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.

7.3 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 MR $\in$ MemRel, SR $\in$ SymbRel, MP $\in$ MemPred be parameters, and $\gtrsim$ be the program simulation relation for the parameters, given in (SIM:PROG) of Fig. 11. Then we have:

THEOREM 7.1 (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* $Q_{\mathtt{src}} \gtrsim Q_{\mathtt{tgt}}$:

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

THEOREM 7.2 (ADEQUACY). *For any programs* $P_{\mathtt{src}}$ *and* $P_{\mathtt{tgt}}$, *if* $P_{\mathtt{src}} \gtrsim P_{\mathtt{tgt}}$:

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

8 RELATED WORK

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

8.1 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 [Stewart 2015] 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. The thesis also 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. Besides what we have discussed, the latest version of CompCertX [Wang et al. 2019] 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 §2.4.2. 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 [Kang et al. 2016] 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 [Jiang et al. 2019] 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[13]); 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* [Kang et al. 2017], which is also based on simple closed simulations, applies to CompCertM to fully support relaxed-memory concurrency.

8.2 Compositional Compiler Correctness for Higher-Order Languages

Pilsner. Pilsner [Hur et al. 2012; Neis et al. 2015] 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 §2, follows that of Pilsner, which is based on the work of [Dreyer et al. 2010]. 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 function.

Multi-language semantics. Ahmed and her collaborators propose multi-language semantics [New et al. 2016; Patterson and Ahmed 2019; Patterson et al. 2017; Perconti and Ahmed 2014; Scherer et al. 2018] 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.

ACKNOWLEDGMENTS

We thank anonymous reviewers for very helpful feedback and Sung-hwan Lee and Yeonwoo Kim for their contribution to early development. This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (2017R1A2B2007512) and by a KAIST new faculty fund (Project No. G04190021).

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

REFERENCES

Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the 20th European Symposium on Programming (ESOP 2011)*.

Andrew W Appel. 2014. *Program Logics for Certified Compilers*. Cambridge University Press.

Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems (ESOP 2014)*.

Derek Dreyer, Georg Neis, and Lars Birkedal. 2010. 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)*.

Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. 2011. CertiKOS: A Certified Kernel for Secure Cloud Computing. In *Proceedings of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2011)*.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. 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)*.

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The Marriage of Bisimulations and Kripke Logical Relations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*.

Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*.

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*.

Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. 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)*.

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*.

Xavier Leroy. 2006. 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)*.

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115.

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. 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)*.

Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation via Universal Embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*.

Daniel Patterson and Amal Ahmed. 2019. The Next 700 Compiler Correctness Theorems (Functional Pearl). In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming (ICFP 2019)*.

Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: Reasonably Mixing a Functional Language with Assembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*.

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *Proceedings of the 23rd European Symposium on Programming (ESOP 2014)*.

Gabriel Scherer, Max S. New, Nick Rioux, and Amal Ahmed. 2018. FabULous Interoperability for ML and a Linear Language. In *Proceedings of the European Joint Conferences on Theory and Practice of Software (ETAPS 2018)*.

Gordon Stewart. 2015. *Verified Separate Compilation for C*. Ph.D. Dissertation. Princeton University.

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*.

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. 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)*.