AFRL-AFOSR-JP-TR-2018-0072

Formal Model of a Multi-Core Kernel-based System

June Andronick
NATIONAL ICT AUSTRALIA LIMITED
L 5 13 GARDEN ST
EVELEIGH, 2015
AU

10/10/2018
Final Report

Air Force Research Laboratory
Air Force Office of Scientific Research
Asian Office of Aerospace Research and Development
Unit 45002, APO AP 96338-5002

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
10-10-2018	Final	07 Jul 2015 to 06 Jul 2018

4. TITLE AND SUBTITLE
Formal Model of a Multi-Core Kernel-based System

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA2386-15-1-4055

5c. PROGRAM ELEMENT NUMBER
61102F

6. AUTHOR(S)
June Andronick, Carroll Morgan, Gerwin Klein

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NATIONAL ICT AUSTRALIA LIMITED
L 5 13 GARDEN ST
EVELEIGH, 2015 AU

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AOARD
UNIT 45002
APO AP 96338-5002

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/AFOSR IOA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-AFOSR-JP-TR-2018-0072

12. DISTRIBUTION/AVAILABILITY STATEMENT
A DISTRIBUTION UNLIMITED: PB Public Release

13. SUPPLEMENTARY NOTES

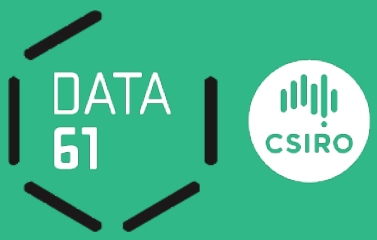
14. ABSTRACT
Professor Andronick and her research team sought to address the grand challenge of providing strong, mathematical guarantees for software that runs on multi-core platforms, thus meeting the increasing demand for more computing power even for critical real-world systems. The research team targeted the operating system kernel, which is the core and foundation of any software system.

Their approach was to to solve a large part of the scalability problem in foundational concurrency reasoning by exploiting automation in modern machine-checked theorem proving. In addition, they sought to soundly reduce reasoning about interleaving with a faithful representation of hardware and software mechanisms such as scheduling, priorities, interrupts, and locks; and to continue to exploit kernel design principles for reducing verification effort.

The project has achieved all planned goals, including its stretch goals. They have defined a formal model of execution for a multi-core kernel, as well as a low-level formal language to push the guarantees as close as possible to the real implementation. This framework has been applied to a multi-core version of seL4 [Klein et al., 2009], the landmark verified microkernel, whose verification so far is restricted to uniprocessor systems. They have defined a formal high-level model of multi-core seL4 and proved the correctness of its most critical operation. They have also developed an initial refinement framework that will allow us to carry the proofs done at the high specification level down to the low implementation level. The project paves the way for proving full functional correctness of multi-core, high-performance microkernels.

15. SUBJECT TERMS
multicore kernel, AOARD

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			SINGLETON, BRIANA
Unclassified	Unclassified	Unclassified	SAR		19b. TELEPHONE NUMBER *(Include area code)* 315-227-7007



Formal Model of a Multi-Core Kernel-based System

AOARD Final Report

June Andronick, Gerwin Klein and Corey Lewis

firstname.lastname@data61.csiro.au

October 2018

Summary

This document is the final report of the project *Formal Model of a Multi-Core Kernel-based System*, under AOARD grant 154055, that ran from July 2015 to July 2018.

The project addressed the grand challenge of providing strong, *mathematical guarantees* for software that runs on *multi-core* platforms, thus meeting the increasing demand for more computing power even for critical real-world systems. We targeted the operating system kernel, which is the core and foundation of any software system.

The project has achieved all planned goal, including its stretch goals.

We have defined a formal model of execution for a multi-core kernel, as well as a low-level formal language to push the guarantees as close as possible to the real implementation. We have applied this framework to a multi-core version of seL4 [Klein et al., 2009], the landmark verified microkernel, whose verification so far is restricted to uniprocessor systems. We have defined a formal high-level model of multi-core seL4 and proved the correctness of its most critical operation. We have also developed an initial refinement framework that will allow us to carry the proofs done at the high specification level down to the low implementation level.

The project paves the way for proving full functional correctness of multi-core, high-performance microkernels.

1 Introduction

The software world keeps demanding more and more computing power. End-users continue to want increasingly better performing software devices. This frantic race does not spare critical software systems, like autonomous vehicles or medical devices, which must look into increasing their performance by running concurrently on multiple processors/cores[1].

This strongly intensifies the technological, financial and sociological challenge of performance versus security, safety and correctness.

Critical software systems need strong guarantees about their correct behaviour, and existing sequential techniques to provide such formal, mathematical proofs do not apply well to concurrent software. Reasoning about concurrent programs is much harder than reasoning about sequential ones because of all the possible execution interleavings that need to be considered.

In this project we have created a framework for producing formal models and proofs of concurrent software, and we have demonstrated it by extending our landmark verification of the seL4 operating system kernel [Klein et al., 2009] to a multi-core version.

An operating system kernel is the most critical part of a software system. It serves as an interface between the hardware and the applications, and has privileged status: it controls all applications' accesses and can enforce security, isolate faults, and avoid propagation of attacks. This makes it the most important piece of software to verify formally. Our seL4 kernel is the world's most verified operating system kernel [Klein et al., 2009], with mathematical proofs of correct execution [Klein et al., 2014] and security enforcement [Murray et al., 2013] of its binary implementation [Sewell et al., 2013]. This formal verification effort, partially funded from a number of previous AOARD grants over the years, has led to great interest from industry, with seL4 having been demonstrated in autonomous vehicles in the DARPA-funded HACMS program[HACMS] and now being deployed in a wide range of areas. This verification landmark is also pushing the boundaries of certification requirements, with seL4 currently going through a security evaluation.

The seL4 proofs so far only apply to systems running on single-processor hardware, and we are seeing increasing demand for multi-core support. In separate work, we have produced an implementation of seL4 on multi-core hardware, based on a "big-lock" approach where most-but-not-all of the code is protected by a lock [Peters et al., 2015]. The project presented here has solved a number of the main research challenges to porting the existing proofs to this new multi-core seL4; and we are now looking for funding opportunities to perform this full port using our new framework.

In particular, within this project we have achieved the following significant milestones, producing:

[1] In this report we use the terms core and processor interchangeably.

- a low-level, formal, concurrent, imperative language; we have extended the formal, sequential, imperative SIMPL language [Schirmer, 2006, 2008] to support parallel execution, leading to the COMPLX language, and we have provided both Owicki-Gries and Rely-Guarantee reasoning for COMPLX (Section 2.1);
- a formal model of interleaved execution of programs on multi-core hardware (Section 2.2.1);
- a formal high-level model of multi-core seL4 and proof of correctness of its most critical operation (Section 2.2.2 and Section 2.2.3);
- an initial refinement framework that will allow us to carry the proofs done at the high level down to the low level (Section 2.3).

This achieves all of the milestones defined in the project proposal, including its stretch goals.

With most of the high-level research challenges solved and demonstrated, we would now be able (funding permitting) to use this framework to lift the entire existing seL4 specifications and correctness proofs fully into a multi-core setting.

This would result in a complete proof of correctness for the multi-core implementation of seL4 with respect to its high-level specification, which would achieve significant impact by providing the critical-software industry with a solution that achieves both performance and security/safety/correctness.

2 Results and contributions

We first describe the language we have defined for reasoning about low-level concurrent imperative programs, then our formalisation of multi-core interleaving and its use in an abstract model and proof of multi-core seL4, and finally our initial framework to prove that this abstract model can be refined down to the implementation.

2.1 COMPLX: formalising concurrent imperative programs

We have defined COMPLX, a formal, imperative, concurrent language, with a syntax, a semantics, a set of logic rules, together with their soundness proof, and an automated verification condition generator. The sections below explain the background and approach, and then give an overview of each part of COMPLX. More details can be found in our published paper about COMPLX [Amani et al., 2017], appended to this report.

COMPLX is available from the open source Archive of Formal Proofs [Amani et al., 2016]. It is a general purpose concurrency reasoning framework, and hence can be used in other settings that require reasoning about concurrent imperative programs.

2.1.1 Background and approach

Operating system kernels, like seL4, are written mostly as low-level, highly optimised C programs. Providing strong, mathematical guarantees that such code behaves correctly and has the expected security properties requires a representation of the program in a formal language and logic. A common approach is to formally define the *syntax* and *semantics* of the programming language. The syntax describes all the possible constructs of the language, and the semantics rigorously defines the effects of each construct on a general state of the machine that the program runs on. A program logic is often added to ease the verification of specific programs. For example, Hoare logic describes how to derive that some property, the *postcondition*, holds after the execution of the program from an initial state, described by a *precondition*. Using the logic rules rather than the semantics to conclude properties about the program is only safe if a *soundness* proof is provided for the logic itself. This guarantees that deriving a postcondition using the rules does ensure that it holds on the state resulting from the execution in the semantics. Finally, some automation is also often provided in the form of tools mechanically applying the logic rules, such as a *verification condition generator (VCG)*.

For instance, in our existing formal model and verification of (unicore) seL4, we first translate the C implementation of seL4 into SIMPL [Schirmer, 2006, 2008], within the Isabelle/HOL theorem prover [Nipkow et al., 2002]. Isabelle/HOL provides a general formal framework to build models and prove properties about these models. The SIMPL language is a generic, sequential

and imperative language formalised in Isabelle/HOL and designed for program verification. It can be used to model (almost) all C constructs (function calls, guards to check runtime failures, etc). The SIMPL framework also provides a standard Hoare logic for this language, along with the corresponding VCG and soundness proof.

Another example is our work formalising the eChronos real-time operating system [Andronick et al., 2015, 2016], where the code contains interrupt-driven concurrency. We use an existing framework within Isabelle/HOL called Hoare-Parallel [Prensa Nieto, 2002] that supports verification of a concurrent high-level while-language called IMP. Hoare-Parallel provides a logic based on the foundational Owicki-Gries (OG) method [Owicki and Gries, 1976], as well as the more compositional Rely-Guarantee (RG) method [Jones, 1983], again along with a VCG and proof of soundness.

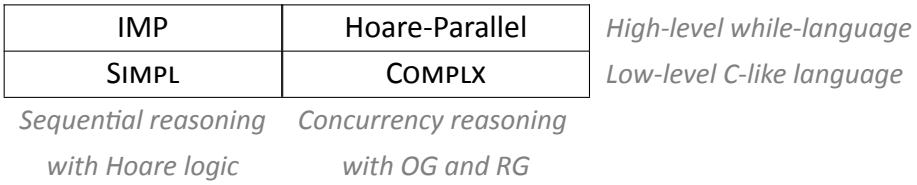


Figure 2.1: COMPLX, SIMPL, IMP and Hoare-Parallel

For this project, we combine the low-level details of SIMPL with the concurrency support of Hoare-Parallel, as depicted in Figure 2.1. COMPLX is defined as an extension to SIMPL with added support for shared-variable concurrency. We have extended the SIMPL abstract syntax with two new constructors: parallel composition, and an *await* statement for synchronisation. We have also defined practical logics based on the OG and RG methods. We have provided VCGs and proven soundness for both logics.

2.1.2 Syntax and semantics

As mentioned previously, COMPLX is an extension of the SIMPL abstract syntax, with the aim of being able to easily reuse many of the tools previously developed for use with specifications written in SIMPL. Therefore, we will begin by presenting the syntax and semantics that COMPLX shares with SIMPL.

As expected, SIMPL (and hence COMPLX) provides all of the standard imperative language constructs, such as variable assignment, sequential composition, conditional statements, while loops, exceptions and function calls. Below is a list of the COMPLX commands taken from SIMPL, with e representing shallowly embedded expressions[1]:

$$\begin{aligned} c = \mathit{Skip} \mid v \ := \ e \mid c_1 \ ; ; \ c_2 \mid \mathit{IF}\ e\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ \mathit{FI} \\ \mid \mathit{WHILE}\ e\ \mathit{DO}\ c\ \mathit{OD} \mid \mathit{TRY}\ c_1\ \mathit{CATCH}\ c_2\ \mathit{END} \\ \mid \mathit{Throw} \mid \mathit{Call}\ n \mid \mathit{DynCom}\ c_s \mid \mathit{Guard}\ f\ g\ c \end{aligned}$$

[1] Shallowly embedded means that the concrete syntax is not represented; instead the semantics of the expression is directly encoded in Isabelle/HOL

DynCom c_s is a dynamic command with a c_s argument that is a function from states to commands. We use it to formalise argument passing and scoping in function calls. The *Guard* $f\ g\ c$ statement throws the fault f if the condition g is false and executes c otherwise. We use it during verification to ensure that C programs do not exhibit undefined behaviour.

While SIMPL has several equivalent semantics, we are mainly interested in its small-step semantics, as COMPLX *must* have a similar small-step semantics to represent the fine-grained interleaving of concurrent programs. This small-step semantics is represented by statements of the form $\Gamma \vdash \langle c, s\rangle \rightarrow \langle c', s'\rangle$, and is read as: program c in state s takes a step to program c' and the updated state s' under the procedure environment Γ which maps function names to function bodies.

In addition to the commands taken from SIMPL, COMPLX has two new commands: *Await* $b\,c$ and *Parallel* $[c_1..c_n]$. Their semantics is that *Await* $b\,c$ will block unless its guard b is true, in which case it can atomically execute all of c. *Parallel* $[c_1..c_n]$ performs a step of any c_i that is not blocked by an *Await*. Several complications arose due to combining these commands with the intricacies of SIMPL; for example, we had to decide how parallel programs behave when one component raises an exception. The details of these problems and exactly how we defined the semantics can be found in our published result [Amani et al., 2017].

2.1.3 Logic, VCG and soundness

In verification of sequential programs the most commonly used technique is Hoare logic [Hoare, 1969], where programs are specified through pre- and post-conditions. Unfortunately, this cannot be used when verifying a concurrent program, as both conditions could be interfered with by another command being executed concurrently.

While there is a wide variety of techniques used to verify concurrent programs, many of them follow a similar approach, which is to separate the problem into two cases, *local correctness* and *global correctness*. A program is locally correct if it is correct with respect to a sequential reading of its semantics, while it is globally correct if it is not interfered with by other commands that are executed concurrently.

The Owicki-Gries (OG) method [Owicki and Gries, 1976] mentioned above is one of the foundational methods used to verify concurrent programs, and it was the first logic we implemented for COMPLX. In this method we first use traditional Hoare logic to prove local correctness, before proving global correctness by showing that every atomic command in each concurrent component does not invalidate the local correctness of other components in the system.

This method requires the concurrent programs to be fully annotated with manually created assertions. Furthermore, the number of proof obligations the verification condition generator (VCG) then creates from these annotations is quadratic in the number of lines of code to be verified. Although we have shown in a previous project [Andronick et al., 2015, 2016] that it is possible to verify complex programs with this approach, we believe that it is unlikely that

OG will scale sufficiently to verify all of multi-core seL4. In short, we believe that OG is a good method for small to medium-sized programs, because it is relatively easy to apply, but for larger programs a scalable method is needed.

One such compositional approach is that of Rely-Guarantee (RG) [Jones, 1983]. Here, instead of annotating every atomic command with an intermediate assertion, we annotate each entire component being run in parallel with both a *rely* and a *guarantee*. The rely captures the behaviour that a component expects from its environment, while the guarantee expresses boundaries on how the component is able to affect its environment. To verify a concurrent program there are now three cases that need to be verified; each component must be locally correct with respect to its rely, each component must satisfy its guarantee, and all of the relies and guarantees must be compatible, in the precise sense that each of the guarantees must imply each of the relies.

We have defined both RG and OG for COMPLX, implemented VCGs and proven both sound with respect to the semantics. Furthermore, we have developed several examples using both of these logics, including the large case-study presented in Section 2.2 which uses our RG framework.

2.2 Model and proof of the big-lock kernel

A major component of this project was to build a model of the big-lock version of seL4 [Peters et al., 2015], with a stretch goal of proving the correctness of its locking implementation. At this stage we have successfully completed both of these goals, building a high-level model in COMPLX and then using Rely-Guarantee to prove that the lock provides the required mutual exclusion.

For this model we have focused on two aspects of the multi-core seL4 implementation; the locking mechanism and the deletion algorithm, which must deal with some intricate details after the introduction of multiple cores.

2.2.1 Model of interleaving

Before modelling and verifying the big-lock kernel we first needed to create a model of interleaving that faithfully represented the fine-grained concurrency between multiple kernel instances, user programs and device interrupts. The model we have developed is partly based on our previous work which used OG to reason about a model of the small eChronos real-time operating system (RTOS) [Andronick et al., 2015, 2016].

Roughly, the system is modelled as two levels of parallel composition, with full interleaving at the top level and *controlled* interleaving in the second. The top level is $D_1||...||D_M||C_1||...||C_N$, which represents the true concurrency possible between the devices D_i and different cores C_j.

Each core C_j is then the parallel composition $K_j||MT_j||U_j^1||...||U_j^L||I_j$, which models the interleaving between the kernel instance K_j and user threads U_j^k, along with the idle thread I_j. However, this interleaving is controlled by the mode transition MT_j, which switches control from a user to the kernel when required.

The key feature of this framework is the controlled interleaving present within each core, which uses a technique from our previous work called *await-painting* [Andronick et al., 2015, 2016]. To control the interleaving we introduce an active-task variable for each core, AT_j, and associate the kernel and user tasks of that core with unique identifiers. Each atomic statement c in task t is then converted into *Await* $(AT_j = t)\, c$, which as seen in Section 2.1.2 means that the execution of c is blocked until the await-condition holds. In particular, this means that if $AT_j = t$ then only t can execute, and all other tasks are not able to interfere with it. Furthermore, there are explicitly only two lines that modify AT_j, one in MT_j that sets $AT_j =$ *Kernel*, and one at the end of the kernel task that sets it back to the current user.

In this way we model the true concurrency possible in real-world multi-core systems. There is full interleaving between different cores and devices while within each core there is only ever one routine that can execute. However, through the use of a parallel composition with controlled interleaving we are able to represent relatively easily the way in which control can jump from one routine to another and back again.

2.2.2 Abstract model of big-lock seL4

We have instantiated the interleaving model to our big-lock seL4 implementation [Peters et al., 2015], meaning that we have provided a model for the kernel code K_j used on each core. (Note that the code is the same on each core.) The user programs U_j^k stay unspecified: we want to prove that some properties are guaranteed by the kernel for any system composed of any user applications running on top of the kernel.

We have intentionally specified the kernel itself at a very abstract level. This has allowed us to focus on two critical issues: whether the locking mechanism correctly provides mutual exclusion, and whether the deletion algorithm correctly protects the system from unsafe situations.

The main feature of the big-lock version of seL4 is that there is a large coarse-grained lock around the kernel. This is intended to ensure that no two cores are executing inside the kernel simultaneously, and makes most of the complex kernel code internally sequential. However, this is not completely possible when deleting objects that other cores might depend on. In this case, seL4 must ensure that all other cores do not currently depend on the object being deleted, which can force those other cores to execute part of the kernel outside of the lock.

To understand this deletion issue in more detail, note that each core has several hardware registers that in normal use contain a pointer to an object in memory. Safe execution of the system requires that the objects these registers point to are valid; that is, they are the correct type of object and have been correctly set up. If they are not valid then the specific core could

```
kernel i ≡
acquire-lock-unless-ipi i;;
IF has-lock i
THEN IF exception i = Syscall Delete ∧
        (∃x. obj-to-delete x)
     THEN delete i
     ELSE SKIP
     FI;;
     schedule i;;
     activate-thread i;;
     release-lock i
ELSE handle-ipi i
FI;;
return-from-exception i
```

```
delete i ≡ select-obj i;;
           init-barriers i;;
           send-all-ipi i;;
           wait-for-all-barriers i;;
           delete-cap-cur-thr i;;
           activate-thread i;;
           invalidate-del-obj i;;
           set-bsection i False
```

Figure 2.2: Model of the multi-core seL4 kernel

crash or violate the overall security policy.

To avoid this, before deleting such an object, an instance of the kernel first ensures that all other cores do not depend on the specific object. However, as it is impossible for one core to inspect the registers of another, the kernel instead does this by signalling the other cores to transition to a state guaranteed not to depend on the object. For example, before the kernel running on core i deletes a thread control block (TCB) x, it will first send a message to all other cores that might be currently running x.[2] Each of these cores will then switch control to their instances of the kernel, which will execute a small section of code outside of the lock. This code switches the current thread running on that core to the *idle thread*, which is guaranteed to always be valid. Once all other cores signal completion of this process back to core i, the original kernel instance can proceed to delete the TCB x.

With this in mind, an overview of the model we have developed for the kernel can be seen in Figure 2.2. The full model is approximately 100 lines of Isabelle source, as compared to the existing abstract specification which is approximately 1000 lines and the concrete implementation which is 10,000 lines of C code. As mentioned, in this model we have focused primarily on the big lock around the kernel and the synchronisation during deletion.

2.2.3 Proof of correctness

The statement we have proven about our model can be seen in Figure 2.3. It says that the complete multi-core system, starting in an initial state satisfying the *global invariants* and *relying* on nothing else running in parallel[3], can *guarantee* that it will always maintain the global invariants. Furthermore, if it terminates then the global invariants holds and the multi-core system

[2] This message is sent via an inter-processor interrupt (IPI).

[3] This is expressed by the *RELY* $\{\!|False|\!\}$, which says that any step made by the environment is not accepted.

$$0 < \textit{num-cores} \wedge 0 < \textit{num-threads} \Longrightarrow \Gamma, \Theta \vdash_{\emptyset} \textit{multi-core-system}$$
$$\textit{PRE}\ \{\!|\textit{global-invs}|\!\}$$
$$\textit{RELY}\ \{\!|\textit{False}|\!\}$$
$$\textit{GUAR}\ \{\!|{}^{o}\textit{global-invs} \longrightarrow {}^{a}\textit{global-invs}|\!\}$$
$$\textit{POST}\ \{\!|\textit{global-invs}|\!\}$$
$$\textit{ABR}\ \{\!|\textit{False}|\!\}$$

Figure 2.3: The correctness statement for our high-level model

must not abruptly terminate with an exception.

The global invariants used encompass the two properties described in Section 2.2.2. That is, they require both mutual exclusion for the kernel, and that each core's registers points to valid objects. The invariants also cover a range of other properties that were required to complete the proof. For example, we needed to know that the idle thread and its related objects are always valid so that we could prove that switching to the idle thread preserves the invariant about valid registers.

As part of the proof we had to develop local relies and guarantees for the different parts of the system running in parallel. For example, each kernel instance running on a separate core still maintains the global invariants, and additionally guarantees that various global variables are only modified if the kernel lock is held. As we also prove mutual exclusion, this means that the kernel instances can rely on these variables being unmodified when they hold the lock.

Proving this condition led to three main outcomes. First, the process was a significant test of COMPLX and our RG framework that led to several improvements in proof automation. Second, while not a complete proof down to the implementation, it helped us convince ourselves that the current implementation of multi-core seL4 is indeed correct with respect to these properties. Finally, performing this proof forced us to develop relies and guarantees for the system that we expect to reuse when we prove the full functional correctness down to the multi-core seL4 implementation.

The proof itself was an iterative process involving several steps. These were:

- writing and modifying the model;
- developing relies and guarantees for the different components of the system;
- attempting to prove the goals produced by the verification condition generator (VCG);
- improving the VCG and other parts of the COMPLX RG framework.

The proof ended up taking approximately 1.5 person months and was approximately 850 lines long. In comparison, the proofs for the complete model of unicore seL4 also required global invariants, and the correctness proof for these invariants was approximately 32,000 lines.

2.3 Concurrency aware refinement

A key lesson from our existing work on verifying the functional correctness of the unicore version of seL4 is that we want to prove properties at a high level of abstraction while still ensuring that the properties hold at the code level. We do this by proving *refinement* between the code and our abstract specification. This refinement guarantees that all possible behaviours of the actual code are also possible behaviours of the model.

Unsurprisingly, the framework developed for the existing unicore seL4 refinement proofs is not suitable for specifying and proving refinement between two concurrent programs. As the final stretch goal and milestone of this project we have developed a new framework for proving refinement of concurrent programs and have tested it on a worked example.

There are three key requirements for this new framework. First, it must be able to perform atomicity refinement, where a simple, single-step abstract program is refined to one with multiple instructions. Secondly, the new refinement framework must be compositional so it can scale to the size of the seL4 code. Third, like the original seL4 refinement framework, the new framework must be contextual, which means that the refinement proof can make use of the conditions that previous code has established.

There are two main aspects of our new framework; a specification language amenable to concurrency reasoning and a definition of refinement that satisfies the above requirements.

2.3.1 Specification language

The existing abstract specification of unicore seL4 was written in Isabelle/HOL in a functional style that expresses state change as a mathematical structure known as a *monad*. In the case of seL4, the specification uses the nondeterministic state monad with failure [Cock et al., 2008]. This style, in addition to enabling the usual Hoare logic and refinement reasoning, also allows additional proof techniques such as rewriting to be applied to the specification directly. Since our goal is to re-use as much as possible of the existing sequential seL4 proof, the concurrent specification has to be in a similar monadic style, enabling the same kind of reasoning, but for concurrent instead of sequential specifications. For this, we introduced a new *interference trace monad*, which is designed to be compatible with the existing proofs where possible, while also enabling both concurrency reasoning at the abstract level and refinement to lower levels.

The key idea of the interference trace monad is to record both the final result, as in the sequential model, and the trace of interactions between the program and its environment that led to the results. The trace idea goes back to Aczel [1983], with each trace being a list of transitions from one state to the next. Our model lifts this idea into a functional, monadic setting.

Along with the interference trace monad we have also developed a style of concurrency that we call *limited interference*. Most other languages allow all possible interleavings, except where explicitly forbidden by atomic section. In comparison, our limited interference approach allows

no interleavings, except where explicitly allowed by a new special operator.

The main reason that we are interested in this approach is that it allows us to reuse much of the existing specification and proof for unicore seL4. Due to the lock protecting the kernel, most of the existing specification can be treated as an atomic block, with interference allowed only at a few key points.

Our interference trace monad and limited interference approach has allowed us to develop an RG logic where we can separate the local and global correctness proofs. In this way we can mostly reuse the existing sequential proofs for local correctness, and separately prove global correctness for our specification.

2.3.2 Definition of refinement

The existing seL4 proofs contain several proofs of refinement that we will need to reuse in a complete proof of correctness for multi-core seL4. For this purpose we have developed a new notion of refinement for use on the interference trace monad. To the best of our knowledge this is the first formalisation of such a refinement framework for an Aczel style trace semantics. However, we have noticed that there are similarities between our work and that of Liang et al. [2012] who have also developed a simulation framework for data refinement in a small-step rely-guarantee environment. While there are strong parallels between the issues both frameworks solved, there is also a large difference in that they use an imperative language with deep embedding, whereas we have a functional specification language with shallow, monadic embedding.

The definition we have ended up with satisfies all of our requirements. To allow parallel composition of data refinement we define what we call *prefix fragment refinement*. This definition is additionally contextual both sequentially and in parallel, through optional assumptions about the preconditions and rely conditions.

The other requirement of our refinement framework was that it enables refinement of atomicity, where we decompose an atomic block to several smaller steps. For this purpose we have added additional state relations to the definition of prefix fragment refinement that allows related programs to temporarily diverge.

With this framework defined, we are now in a position (funding permitting) to use it to fully lift the entire existing seL4 specifications and correctness proofs to a multi-core setting.

3 Research Outcomes

Through this project we have solved several fundamental research challenges related to concurrency reasoning. We have developed COMPLX, a new language framework for reasoning about low-level, concurrent code and have provided two proof logics for it. We have also constructed a model of interleaving that faithfully represents the real hardware and software mechanisms while enabling and optimising interference-freedom proofs. We then used this to create a high-level model of multi-core seL4, along with a proof of correctness of its critical operation. Finally, we have an initial framework for performing refinement of concurrent programs, which has been designed to allow us to reuse the existing proofs of unicore seL4.

With these challenges solved we now believe that we would be able to use this work to develop a complete proof of correctness for multi-core seL4 with respect to a high level specification. This would have significant impact as multi-core computing power is increasingly in demand, even for critical software. It would enable formally verified software to be deployed on modern, concurrent systems, achieving both performance and security.

Bibliography

Peter Aczel. On an inference rule for parallel composition, 1983. URL http://homepages.cs.ncl.ac.uk/cliff.jones/publications/MSs/PHGA-traces.pdf. Private communication to Cliff Jones. 12

Sidney Amani, June Andronick, Maksym Bortin, Corey Lewis, Christine Rizkallah, and Joseph Tuong. Complx: A verification framework for concurrent imperative programs. *Archive of Formal Proofs*, Nov 2016. ISSN 2150-914x. http://isa-afp.org/entries/Complx.html, Formal proof development. 5

Sidney Amani, June Andronick, Maksym Bortin, Corey Lewis, Christine Rizkallah, and Joey Tuong. COMPLX: a verification framework for concurrent imperative programs. In *International Conference on Certified Programs and Proofs*, pages 138–150, Paris, France, Jan 2017. ACM-SIGPLAN. 5, 7

June Andronick, Corey Lewis, and Carroll Morgan. Controlled Owicki-Gries concurrency: Reasoning about the preemptible eChronos embedded operating system. In *Workshop on Models for Formal Analysis of Real Systems (MARS)*, 2015. 6, 7, 8, 9

June Andronick, Corey Lewis, Daniel Matichuk, Carroll Morgan, and Christine Rizkallah. Proof of OS scheduling behavior in the presence of interrupt-induced concurrency. In *International Conference on Interactive Theorem Proving*, Nancy, France, Aug 2016. 6, 7, 8, 9

David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Muñoz, Sofiène Tahar, editor, *21st TPHOLs*, pages 167–182, Montreal, Canada, Aug 2008. Springer. doi: 10.1007/978-3-540-71067-7_16. 12

HACMS. https://www.darpa.mil/program/high-assurance-cyber-military-systems. 3

C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12:576–580, 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. 7

C. B. Jones. Tentative steps towards a development method for interfering programs. *Trans. Progr. Lang. & Syst.*, 5(4):596–619, 1983. 6, 8

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM. 2, 3

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *Trans. Comp. Syst.*, 32(1):2:1–2:70, 2014. doi: 10.1145/2560537. 3

Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *ACM SIGPLAN Notices*, volume 47, pages 455–468. ACM, 2012. 13

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *S&P*, pages 415–429, San Francisco, CA, May 2013. doi: 10.1109/SP.2013.35. 3

Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. ISBN 978-3-540-43376-7. doi: 10.1007/3-540-45949-9. 5

Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976. 6, 7

Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. For a microkernel, a big lock is fine. In *APSys*, Tokyo, JP, Jul 2015. 3, 8, 9

Leonor Prensa Nieto. *Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL*. PhD thesis, T.U. München, 2002. 6

Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006. 4, 5

Norbert Schirmer. A sequential imperative programming language syntax, semantics, Hoare logics and verification environment. *Archive of Formal Proofs*, Feb 2008. ISSN 2150-914x. http://isa-afp.org/entries/Simpl.shtml, Formal proof development. 4, 5

Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *PLDI*, pages 471–481, Seattle, Washington, USA, Jun 2013. ACM. 3

CONTACT US

t 1300 363 400
+61 3 9345 2176
e enquiries@data61.csiro.au
w www.data61.csiro.au

AT CSIRO WE SHAPE THE FUTURE

We do this by using science and technology to solve real issues. Our research makes a difference to industry, people and the planet.

FOR FURTHER INFORMATION

June Andronick, Gerwin Klein and Corey Lewis

e firstname.lastname@data61.csiro.au
w trustworthy.systems

COMPLX: A Verification Framework for Concurrent Imperative Programs

Sidney Amani[1] June Andronick[1,2] Maksym Bortin[1]
Corey Lewis[1] Christine Rizkallah[3,*] Joseph Tuong[4,*]

[1]Data61, CSIRO, Australia [2]UNSW, Australia [3]University of Pennsylvania, U.S. [4]Freelancer, Australia
[1] firstname.lastname@data61.csiro.au [3] criz@seas.upenn.edu [4] joey.tuong@gmail.com

[*] Work done while at Data61, CSIRO

Abstract

We propose a concurrency reasoning framework for imperative programs, based on the Owicki-Gries (OG) foundational shared-variable concurrency method. Our framework combines the approaches of Hoare-Parallel, a formalisation of OG in Isabelle/HOL for a simple while-language, and SIMPL, a generic imperative language embedded in Isabelle/HOL, allowing formal reasoning on C programs.

We define the COMPLX language, extending the syntax and semantics of SIMPL with support for parallel composition and synchronisation. We additionally define an OG logic, which we prove sound w.r.t. the semantics, and a verification condition generator, both supporting involved low-level imperative constructs such as function calls and abrupt termination. We illustrate our framework on an example that features exceptions, guards and function calls. We aim to then target concurrent operating systems, such as the interruptible eChronos embedded operating system for which we already have a model-level OG proof using Hoare-Parallel.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords formal verification, programming languages, imperative code, concurrency, Owicki-Gries, Isabelle/HOL

1. Introduction

C is still the language of choice for developing software with high performance and precise memory requirements, for it allows aggressive manual optimisation. At the same time, performance-demanding low-level systems, such as operating system (OS) kernels or real-time systems, also have strong safety and security objectives, which call for formal verification. Multiple frameworks for formal reasoning about C programs exist and have successfully been used, ranging from push-button automated tools to check the absence of classes of runtime errors, to more effort-intensive interactive methods to prove deeper correctness properties. We target the latter, minimising the trust needed in the tools, maximising the strength of properties that can be proven. To this end, we propose a framework for formal, interactive verification of shared-variable concurrent imperative low-level programs, which can be combined with a C parser front end for concurrent C code verification, paving the way to verified interruptible or multicore systems.

We follow a common approach to reasoning about programs: embedding the given language within a powerful theorem prover, Isabelle/HOL [Nipkow et al. 2002] in our case. That is, we define abstract and concrete syntax, and specify runtime behaviour. Although it is possible to directly reason over the semantics of programs, it is untenable and not scalable. It can instead be automated by the definition of a set of logic rules, reducing the program's correctness statement to a series of simpler verification conditions. This verification condition generator (VCG) is typically syntax-directed, unfolding the proof according to the rules of the logic. In order to justify such reasoning with our set of rules, we prove their soundness with respect to the language's formal semantics.

Such an infrastructure exists for reasoning about *sequential* C programs in Isabelle/HOL. C programs are translated into SIMPL [Schirmer 2006, 2008], a generic, sequential, imperative language formalised in Isabelle/HOL. The C-to-Isabelle translation [Tuch et al. 2007] is unavoidably trusted, parsing C code into formal logic, and is therefore as conservative and direct as possible. The SIMPL framework provides syntax and semantics for the language, as well as a Hoare logic (with its soundness proof) and a VCG. It has successfully been used in the landmark verification of the seL4 microkernel, guaranteeing multiple correctness prop-

erties of seL4 to the C implementation [Klein et al. 2010; Murray et al. 2012]. The framework, however, lacks the ability to reason about concurrency.

We extend SIMPL with support for shared-variable concurrency, following the foundational Owicki-Gries (OG) method [Owicki and Gries 1976]. OG has been formalised in Isabelle/HOL's library in the Hoare-Parallel theory [Prensa Nieto 2002] for a simple high-level while-language IMP. We chose OG over more recent variants (e.g. Rely-Guarantee [Jones 1983], Concurrent Separation Logic [OHearn 2007]) for the simplicity of OG logic and its suitability to reason about potentially-racy high-performance shared-variable system code: we previously successfully used it for a *model-level* verification of the interruptible eChronos embedded OS [Andronick et al. 2015, 2016]. The OS provides an API to applications for synchronisation and locking, but the OS code itself shares racy memory state with interrupt handlers[1]. In this previous work, we used Hoare-Parallel's formalisation of OG and we demonstrated how the well-known explosion of verification conditions of the OG method can be efficiently handled by the powerful automation of modern theorem provers and by the careful modelling of controlled interleaving. We now want to push our proofs down to guarantees about the C implementation, which is the motivation for the work presented here.

Our contributions are the following:

- We propose the language COMPLX as an extension of SIMPL with support for parallel composition and synchronisation, and we define its concurrent semantics. We largely reuse SIMPL's existing infrastructure to facilitate the port of existing verifications that use SIMPL (section 3).
- We define a practical OG-based logic, inspired by Hoare-Parallel, and a VCG to facilitate semi-automated proof using the logic (section 4).
- We prove our logic sound with respect to the semantics, ensuring that proofs using the logic are true guarantees about the execution of the program (section 5).
- Finally, we present a case-study demonstrating the use and practicality of this framework for the verification of concurrent imperative programs (section 6).
- As part of the examples, we demonstrate how we support concurrent function calls, including a technique to handle arguments passing and local variables.

With additional work, the existing infrastructure around SIMPL, including the C-to-Isabelle parser, can be updated to enable reasoning about a significant subset of concurrent C code in Isabelle/HOL. This would open up the applicability to several existing codebases, including the eChronos OS, and potentially a multicore variant of seL4. Our framework assumes that the granularity of interleaving is that of C instructions; porting the guarantees down to executable code and weak memory architectures is not in the scope of this paper.

All our Isabelle/HOL formalisations and the case studies are available online [COMPLX].

[1] Preventing races would require disabling interrupts, resulting in increases of latency unacceptable for such real-time systems.

2. Background

In this section, we present existing work that our paper combines and extends. The first section presents SIMPL, an existing formalisation of sequential imperative programs in Isabelle/HOL (and the existing infrastructure to verify C code). The second presents Hoare-Parallel, an existing formalisation of OG for a simple while-language in Isabelle/HOL. Our work consolidates those two components to create a language that provides a basis for reasoning about concurrent C code in Isabelle/HOL.

2.1 Verification of C in Isabelle/HOL

As mentioned in the introduction, SIMPL allows embedding of real programming languages into Isabelle/HOL, and is sufficiently expressive to model a substantial subset of C features. SIMPL can be used directly for reasoning about C code and it has indeed been used directly in the verification of LEDA's [Mehlhorn and Näher 1999] shortest path checker [Rizkallah 2014].

A far more common verification approach though is using the C-to-Isabelle parser [Tuch et al. 2007] which converts a large subset of C99 code into low-level SIMPL code. SIMPL and the C-to-Isabelle parser together provide an established infrastructure for the verification of sequential C programs in Isabelle/HOL. They have been used in the verification of the seL4 microkernel which is written in C [Klein et al. 2010] and in several other C verification projects [Amani et al. 2016; Murray et al. 2012; Noschinski et al. 2014].

Syntax SIMPL provides the usual imperative language constructs, including functions, variable assignment, sequential composition, conditional statements, while loops, and exceptions. SIMPL has no expression language of its own; expressions are shallowly embedded. The notion of state is also generic and left for instantiation; it is defined as an Isabelle record of local and global variables (variables are then simply functions on the state). The C-to-Isabelle parser only supports side-effect-free expressions, modelled as Isabelle/HOL expressions, and it instantiates the state space to C memory states. The following is a summary of SIMPL syntactic forms, where e represents an expression:

$$\begin{aligned} c = \; & \mathit{Skip} \mid v := e \mid c_1 \,;; \, c_2 \mid \mathit{IF}\ e\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ \mathit{FI} \\ & \mid \mathit{WHILE}\ e\ \mathit{DO}\ c\ \mathit{OD} \mid \mathit{TRY}\ c_1\ \mathit{CATCH}\ c_2\ \mathit{END} \\ & \mid \mathit{Throw} \mid \mathit{Call}\ n \mid \mathit{DynCom}\ c_s \mid \mathit{Guard}\ f\ g\ c \end{aligned}$$

The $\mathit{DynCom}\ c_s$ statement is a dynamic (state dependent) command that takes as argument c_s which is a func-

tion from states to commands. It is used in C verification to encode argument passing and scoping in function calls. The $Guard\ f\ g\ c$ statement throws the fault f if the condition g is false and executes c otherwise. It is used in C verification to encode certain correctness conditions ensuring that the C program does not exhibit undefined behaviour (e.g. division by zero). $Call$ just takes the name of the function being called.

Semantics Computations in SIMPL are described by several equivalent models, including big- and small-step semantics. Here we are interested in the small-step semantics, as we want to model the fine-grain interleaving of concurrent programs.

The small-step semantics is represented by statements of the form $\Gamma \vdash \langle c, s\rangle \rightarrow \langle c', s'\rangle$ that read as: program c in state s takes a step to program c' and the updated state s' under the procedure environment Γ which maps function names to function bodies. Both s and s' are *extended* states: they are either $Normal$ states, representing typical execution flow (including exception handling), or $Stuck$ states, generated by calls to non-existent procedures, or $Fault$ states, generated by failed Guard statements. For normal program states, $s = Normal\, x$, the semantics is as expected; whereas in cases $s = Stuck$ or $s = Fault\ f$ we may only transform c to $Skip$ with $s' = s$.

Exceptions are used to represent abrupt termination — function calls and loops are wrapped in a try-catch block and the C statements `return`, `break`, and `continue` are implemented by assigning appropriate value to an auxiliary variable and raising an exception with $Throw$. The exception is caught by $CATCH$, mimicking an abrupt termination of the TRY block.

Verification Specifications for SIMPL programs are given as Hoare triples, where pre-conditions and post-conditions are stated by Isabelle expressions. The SIMPL environment provides a VCG for partial correctness that converts those Hoare triples to a set of higher-order formulas that are easier to reason about. The Hoare triples are represented by statements of the form $\Gamma \vdash_{/F} P\, c\, Q, A$, where P is the pre-condition, Q is the post-condition for normal termination, A is the abrupt-condition for abrupt termination[2], and F is the set of faults allowed. A soundness proof guarantees the safe use of the Hoare logic instead of directly reasoning about the semantics: it states that if such a Hoare triple is established, then all final states reached through the execution of the command (according to the semantics) from an initial state that satisfies the pre-condition, will satisfy the post-condition and the abrupt-condition.

[2] Using Schirmer's [Schirmer 2006] terminology, we refer to post-conditions for abrupt-termination due to uncaught exceptions as abrupt-conditions.

2.2 Verification of Concurrent Code in Isabelle/HOL

Hoare logic may be used to prove that a thread in a concurrent program is *locally correct*, i.e. that it is correct under a sequential interpretation of its semantics without interleaving of external commands. In order to prove that it is correct in a concurrent setting, we have to additionally prove that it is *globally correct*, i.e. that is is still correct considering all possible interleavings with other threads in the system.

The Owicki-Gries [Owicki and Gries 1976] method for the verification of shared-variable concurrent programs extends the proof method for sequential correctness with the concept of *interference freedom*: each thread is first proved to be locally correct and then each atomic command in each thread is proved to not interfere with (i.e. invalidate) the local correctness proof of another thread in parallel. If the proof of local correctness for a command c requires a pre-condition P, and c may be interleaved with another command c' whose pre-condition is P', then in order to show interference freedom, we show that $\{P' \wedge P\}\, c'\, \{P\}$ holds, i.e. that P remains true after being interleaved with c'.

In order to satisfy the requirements for interference freedom over all threads in a system, it is necessary to store these intermediate *assertions*. Unlike for a sequential program, we may require post-conditions to be arbitrarily stronger than the weakest pre-condition implied by the Hoare logic. For this reason, we need to *fully annotate* the concurrent program with intermediate assertions in order to verify its correctness relative to other threads in the system, in contrast to a sequential program, for which these properties may largely be automatically derived and discarded once used.

Hoare-Parallel [Prensa Nieto 2002] is a formal reasoning framework in Isabelle/HOL for a simple concurrent language, including a formalisation of OG. The language consists of assignment, sequential composition, conditionals, loops, and two additional statements for concurrency: $Parallel\, [ac_1..ac_n]$ and $Await\, b\, c$. The execution is modelled through a small-step semantics; an await statement can only do a step if its boolean guard b is true, in which case its body c is executed atomically; a step of a parallel composition of programs is a step of any of its thread that is not blocked on an await. Hoare-Parallel's abstract syntax is defined using the following mutually recursive datatype:

$$
\begin{array}{ll}
ac = & AnnSeq\ ac\ ac \mid AnnBasic\ r\ f \mid AnnCond\ b\ ac\ ac \\
 & \mid AnnWhile\ r\ b\ r\ ac \mid AnnAwait\ r\ b\ c \\
\textbf{and} & \\
c = & Parallel\ [ac..ac] \mid Seq\ c\ c \mid Basic\ f \mid Cond\ b\ c\ c \\
 & \mid While\ b\ r\ c
\end{array}
$$

The outer layer c performs sequential actions or initiates a parallel composition, and the inner layer ac within the parallel composition expresses a thread, with each action annotated with an assertion r.

COBEGIN ... COEND is used as syntactic sugar for *Parallel*. Throughout the paper we reuse the Hoare-Parallel's *Parallel* concrete syntax for COMPLX.

3. COMPLX: Syntax and Semantics

Recall that the aim of COMPLX is to enrich SIMPL with parallel composition and synchronisation. The Hoare-Parallel development shows how the OG method can be formalised in Isabelle/HOL, introducing syntax and the small-step semantics for parallel components. We largely reuse this approach with two major deviations. Firstly, we do not incorporate annotations into COMPLX abstract syntax but rather represent annotations using a separate datatype. Annotations and programs will be related in the next section by means of the OG logic. This way the abstract syntax remains simple and clear, and we can reuse the existing C-to-Isabelle parser. Secondly, we do not separate parallel and sequential programs into different layers, but rather have one datatype representing both. These decisions make the soundness proof more complicated, but allow COMPLX programs to have nested parallelism, thus lifting unnecessary syntactic restrictions.

In this sense, COMPLX just extends the SIMPL abstract syntax by two new constructors: $Parallel\, cs$ and $Await\, b\, c$:

$$c = \quad Skip \mid \ldots \mid Parallel\, cs \mid Await\, b\, c$$

where $Parallel$ takes a list of programs cs that run in parallel, and $Await$ takes a set of states b specifying the await-condition, and a program c representing the await-body. It is worth noting that with nested parallelism we could use the canonical binary parallel composition operator $p \parallel q$ instead of $Parallel\, cs$ without any effect to semantic expressivity, since $Parallel\, cs$ can be represented by folding the binary operator. On the other hand, an OG-rule for $p \parallel q$ would lack the possibility to collect and handle interference freedom of more than two parallel components within a single proof obligation, but distribute it in accordance to the fold strategy. To avoid such complications, $Parallel$ takes a list of parallel components directly in COMPLX abstract syntax as shown above.

Next, we extend the small-step semantics of SIMPL to the new language constructs. As mentioned previously, we use small-step semantics to allow for reasoning about interleavings between each atomic step. In what follows, we reuse the SIMPL notation $\Gamma \vdash \langle c, s\rangle \to \langle c', s'\rangle$ meaning that the configuration $\langle c, s\rangle$, comprising a COMPLX program c and a state s, can be transformed in one step to the configuration $\langle c', s'\rangle$ under the procedure environment Γ. As usual, we write $\Gamma \vdash \langle c, s\rangle \to^* \langle c', s'\rangle$ for the reflexive-transitive closure of the small-step relation.

To adapt the semantics of $Parallel\, cs$ and $Await\, b\, c$ from Hoare-Parallel to SIMPL's involved computation model, we have to take into account several kinds of states: $Normal$, $Fault$, and $Stuck$, as well as exception handling. New situations arise that neither SIMPL nor the Hoare-Parallel formalisations had to deal with. For instance, we have to decide how a parallel program shall behave in the case when one of its threads raises an uncaught exception. In this case we allow the parallel program to stop all other threads and exit with the exception. The rule *Parallel-Throw*:

$$\frac{Throw \;\in\; set\, cs}{\Gamma \vdash \langle Parallel\, cs, s\rangle \to \langle Throw, s\rangle}$$

captures this behaviour, where set just converts a list to a set. However, the parallel program may also continue its computation, delaying the exception, provided by the fundamental *Parallel* rule:

$$\frac{\Gamma \vdash \langle cs_i, s\rangle \to \langle c, s'\rangle \quad i < |cs|}{\Gamma \vdash \langle Parallel\, cs, s\rangle \to \langle Parallel\, cs[i := c], s'\rangle}$$

where $|cs|$ denotes the length of the list cs, cs_i the i-th (counting from 0) element of cs, and $cs[i := c]$ the list cs with its i-th element replaced by c. Furthermore, a parallel program is allowed to terminate properly only if all its threads do so. This is described by the *Parallel-Skip* rule:

$$\frac{\forall c \;\in\; set\, cs.\; c = Skip}{\Gamma \vdash \langle Parallel\, cs, s\rangle \to \langle Skip, s\rangle}$$

Next, for $Await\, b\, c$ to be processed in a state $Normal\, x$, the await-condition must be satisfied, i.e. $x \in b$ must hold. Otherwise the execution is blocked. Moreover, the body of the await c must be a sequential program without any further $Await$ statements or $Parallel$ compositions. Following the Hoare-Parallel notation, we denote this condition by $atom\_com\ c$. Now, any computation $\Gamma \vdash \langle c, Normal\ x\rangle \to^* \langle Skip, Normal\ y\rangle$ allows us to derive $\Gamma \vdash \langle Await\, b\, c, Normal\ x\rangle \to \langle Skip, Normal\ y\rangle$. In other words, if the await-body terminates in a number of small-steps without any interleavings then $Await\, b\, c$ can make the same transition in a single step. Here again we have to consider potential exceptions raised by c, in which case we let $Await\, b\, c$ throw an exception as well. These behaviours are formalised by the following rules, where $s = Normal\, x$ and $s' = Normal\, y$.

$$\frac{x \in b \quad atom\_com\ c \quad \Gamma \vdash \langle c, s\rangle \to^* \langle Skip, s'\rangle}{\Gamma \vdash \langle Await\, b\, c, s\rangle \to \langle Skip, s'\rangle}$$

$$\frac{x \in b \quad atom\_com\ c \quad \Gamma \vdash \langle c, s\rangle \to^* \langle Throw, s'\rangle}{\Gamma \vdash \langle Await\, b\, c, s\rangle \to \langle Throw, s'\rangle}$$

The cases when an execution of the await-body c results not in $Normal\, y$, but in a state s' other than normal (e.g. $Stuck$), are handled in a similar manner: $\langle Await\, b\, c, Normal\ x\rangle$ can take a single small-step to $\langle Skip, s'\rangle$.

4. Owicki-Gries Logic for COMPLX

Verification of programs by directly reasoning about the semantics of the language is cumbersome and not easily

amenable to automation. For sequential SIMPL programs, SIMPL's Hoare logic allows for weakest pre-condition style reasoning, generating intermediate assertions, and a small set of verification conditions that guarantee partial correctness (i.e. correctness in case of termination). For our concurrent COMPLX programs, we create an OG logic, similar to the one defined in Hoare-Parallel, that breaks down the correctness of a parallel program into local correctness and global correctness verification conditions.

4.1 Annotations

As explained in section 3, we use a single datatype to represent sequential and concurrent programs. Moreover, our OG annotations are specified using a separate datatype called an *annotation tree*, which is isomorphic to the abstract syntax tree of the COMPLX program. The annotation tree contains *assertions* at each step in the program and is represented as follows:

$$\begin{aligned} a = \quad & AnnExpr\ r \mid AnnRec\ r\ a \mid AnnWhile\ r\ r\ a \\ & \mid AnnComp\ a\ a \mid AnnCond\ r\ a\ a \\ & \mid AnnPar\ l \mid AnnCall\ r\ i \end{aligned}$$

Non-recursive command constructors such as $Skip$, $Throw$, etc. are annotated via an $AnnExpr$ node, which carries a single assertion r that is merely a set of states and is also used for post-conditions of OG rules. $AnnRec$ is used to annotate recursive commands, such as $Await$, $DynCom$ or $Guard$s, that hold another annotated command a. While-commands require a special annotation type that provides an assertion for the while, a loop invariant, as well as an annotation tree for the loop body. Sequential composition and $Catch$ statements are annotated via $AnnComp$, where an annotation sub-tree is provided for each component of the sub-commands. Similarly, $AnnCond$ is used for conditional statements, but in addition to the two annotation sub-trees, it carries an assertion for the conditional statement itself.

$AnnPar$ is used to annotate $Parallel$ statements, hence, it stores a list l of triples containing an annotation tree, a post-condition and an abrupt-condition, with one element in the list per parallel component. The post-conditions and abrupt-conditions must be specified by the user, because they are part of the interference freedom requirements. More specifically, we must show that none of these conditions can be violated due to other components activity.

Finally, $Call$ statements are annotated with $AnnCall$, which holds an assertion r and a routine index i of type natural number, specifying which annotation tree to select from the annotation environment. We return to this at the end of subsection 4.2.

Despite having a separate datatype for the program and the annotation tree, COMPLX's syntactic sugar allows a user to annotate a program directly. This way we specify assertions at each step of the program, making it easy to keep track of the assertions when following the control flow of the program.

For instance, the following is a COMPLX program with the annotations and program text combined.

$x := 0;; y := 0;;$
COBEGIN
$\{\!|a|\!\}\ x := 1\ \{\!|Q_x|\!\}, \{\!|A_x|\!\} \parallel \{\!|b|\!\}\ y := 1\ \{\!|Q_y|\!\}, \{\!|A_y|\!\}$
COEND

This produces two different trees, one for the program itself (where $Basic\ f$ models state update by the function f, here variable assignment):

$$\begin{aligned} & Seq\ (Basic\ (x\_update\ (\lambda\text{-}.\ 0))) \\ & \quad (Seq\ (Basic\ (y\_update\ (\lambda\text{-}.\ 0))) \\ & \qquad (Parallel\ [Basic\ (x\_update\ (\lambda\text{-}.\ 1)), \\ & \qquad\qquad Basic\ (y\_update\ (\lambda\text{-}.\ 1))])) \end{aligned}$$

and a separate annotation tree of the form

$$\begin{aligned} & AnnComp\ (AnnExpr\ \{\!|True|\!\}) \\ & \quad (AnnComp\ (AnnExpr\ \{\!|True|\!\}) \\ & \qquad (AnnPar\ [(AnnExpr\ \{\!|a|\!\},\ \{\!|Q_x|\!\},\ \{\!|A_x|\!\}), \\ & \qquad\qquad (AnnExpr\ \{\!|b|\!\},\ \{\!|Q_y|\!\},\ \{\!|A_y|\!\})])) \end{aligned}$$

In the annotation tree, the trivial, unused assertions for the sequential parts are automatically added by the syntactic sugar, removing the burden from the user.

4.2 Owicki-Gries Rules

We define an OG statement of the form

$$\Gamma, \Theta \vdash_{/F} a\ c\ \{\!|Q|\!\}, \{\!|A|\!\}$$

stating that the COMPLX program c with the annotation tree a either ends in one of the fault states specified by F, or a $Normal$ state. If that $Normal$ state is an exception, it must satisfy the abrupt-condition A, otherwise it must satisfy the post-condition Q. Γ is the procedure environment, mapping function names to function bodies, and Θ is the annotation environment, mapping function names to annotation trees.

To enable weakest pre-condition reasoning when proving a sequential part of a program (i.e. within an $Await$ or top-level non-parallel commands), we have another OG statement which takes an extra pre-condition $\{\!|P|\!\}$:

$$\Gamma, \Theta \Vdash_{/F} \{\!|P|\!\}\ a\ c\ \{\!|Q|\!\}, \{\!|A|\!\}$$

This means that we duplicate every OG rules and the sequential version of a rule ignores the annotation tree. We borrowed this idea from Hoare-Parallel, which also has two versions for each rule. In our case, the annotation tree exists but is only used as soon as we switch to parallel mode.

Figure 1 illustrates some of the important OG logic rules for COMPLX. We omitted all the rules used for sequential reasoning except for $SeqParallel$ which allows switching from sequential mode (denoted by $\Vdash$) to parallel mode (denoted by $\vdash$). This rule would be used when the program is

$$\frac{P \subseteq \mathit{pre}\ (\mathit{AnnPar}\ as) \qquad \Gamma, \Theta \vdash_{/F} (\mathit{AnnPar}\ as)\ (\mathit{Parallel}\ cs)\ \{\!|Q|\!\}, \{\!|A|\!\}}{\Gamma, \Theta \Vdash_{/F} P\ (\mathit{AnnPar}\ as)\ (\mathit{Parallel}\ cs)\ \{\!|Q|\!\}, \{\!|A|\!\}}\ \textsc{SeqParallel}$$

$$\frac{\begin{array}{l}|as| = |cs| \qquad \forall i{<}|cs|.\ \Gamma, \Theta \vdash_{/F} (\mathit{pres}\ as_{[i]})\ cs_{[i]}\ (\mathit{postcond}\ as_{[i]}), (\mathit{abrcond}\ as_{[i]}) \\ \mathit{interfree}\ \Gamma\ \Theta\ F\ as\ cs \qquad \bigcap \mathit{map}\ \mathit{postcond}\ as \subseteq \{\!|Q|\!\} \qquad \bigcup \mathit{map}\ \mathit{abrcond}\ as \subseteq \{\!|A|\!\}\end{array}}{\Gamma, \Theta \vdash_{/F} (\mathit{AnnPar}\ as)\ (\mathit{Parallel}\ cs)\ \{\!|Q|\!\}, \{\!|A|\!\}}\ \textsc{Parallel}$$

$$\frac{\Gamma, \Theta \Vdash_{/F} (r \cap b)\ P\ c\ \{\!|Q|\!\}, \{\!|A|\!\} \qquad \mathit{atom\text{-}com}\ c}{\Gamma, \Theta \vdash_{/F} (\mathit{AnnRec}\ r\ P)\ (\mathit{Await}\ b\ c)\ \{\!|Q|\!\}, \{\!|A|\!\}}\ \textsc{Await} \qquad \frac{r \subseteq \mathit{pre}\ a \qquad \forall s{\in}r.\ \Gamma, \Theta \vdash_{/F} a\ (d\ s)\ \{\!|Q|\!\}, \{\!|A|\!\}}{\Gamma, \Theta \vdash_{/F} (\mathit{AnnRec}\ r\ a)\ (\mathit{DynCom}\ d)\ \{\!|Q|\!\}, \{\!|A|\!\}}\ \textsc{DynCom}$$

$$\frac{\Gamma, \Theta \vdash_{/F} P\ c\ \{\!|Q|\!\}, \{\!|A|\!\} \qquad r \cap g \subseteq \mathit{pre}\ P \qquad r \cap -g \neq \emptyset \longrightarrow f \in F}{\Gamma, \Theta \vdash_{/F} (\mathit{AnnRec}\ r\ P)\ (\mathit{Guard}\ f\ g\ c)\ \{\!|Q|\!\}, \{\!|A|\!\}}\ \textsc{Guard}$$

$$\frac{\Theta\ p = \mathit{Some}\ as \qquad r \subseteq \mathit{pre}\ as_{[n]} \qquad \Gamma\ p = \mathit{Some}\ b \qquad n < |as| \qquad \Gamma, \Theta \vdash_{/F} as_{[n]}\ b\ \{\!|Q|\!\}, \{\!|A|\!\}}{\Gamma, \Theta \vdash_{/F} (\mathit{AnnCall}\ r\ n)\ (\mathit{Call}\ p)\ \{\!|Q|\!\}, \{\!|A|\!\}}\ \textsc{Call}$$

Figure 1: Some of the important derivation rules of COMPLX.

finished dealing with an initial sequential part and reaches a parallel composition. Note that the pre-condition $\{\!|P|\!\}$ in the sequential OG statement disappears in the parallel one, as long as it implies the pre-condition of the assertion tree. Also note that the OG rules ensure that the annotation tree and the program match, e.g. a Parallel statement can only be proved correct if provided with an $AnnPar$ annotation.

As explained in section 3, COMPLX allows for nested $Parallel$ statements. Several conditions must be met when using the $Parallel$ rule to derive a $Parallel$ statement. Every component of $Parallel$ must itself be derivable. $pres$, $postcond$ and $abrcond$ respectively return the annotation tree, the post-condition and the abrupt-condition of an element of the list in $AnnPar$ described earlier. While the intersection of the post-conditions of all components must imply the post-condition of the overall $Parallel$, for abrupt-conditions only one of the components must satisfy the abrupt-condition of the $Parallel$. This is explained by the fact that exceptions can interrupt other components. The key requirement for derivability of $Parallel$ is $interfree$ which specifies interference freedom — we return to this definition in next section.

A derivation of $Await\ b\ c$ requires a sequential derivation of c with the assertion r combined with the condition b as pre-condition. In addition, the command c must be deprived of $Parallel$ and $Call$ statements since they cannot be atomic and thus are forbidden in $Await$. This restriction is achieved by the *atom-com* predicate and guarantees that a program does not end in a $Stuck$ state because of a non-atomic operation found in an $Await$.

Dynamic commands are functions that produce a command from a state. They provide a general mechanism to model programs that need to introspect their state. For instance, they could be used to model self-modifying code. However, for C verification their use is limited to restoring the value of local variables when a function is called. We elaborate on this in section 6. In order to be able to reason about dynamic commands in OG, we must be able to annotate them. Since program annotations are static, they must not depend on the state of the program. Thus our framework restricts their use to dynamic commands that can be annotated statically. $DynCom$ is derivable so long as an annotation tree a can be provided and that it allows derivation of the command produced by the dynamic command d for any state allowed by the assertion on $DynCom$. An additional requirement for the annotation tree to be valid is that its first assertion must be allowed by the assertion on $DynCom$ (i.e. r). $pre\ a$ returns the first assertion of the annotation tree a.

The $Guard$ rule is straightforward. For the command $Guard\ f\ g\ c$, if the guard condition g is not satisfied, the fault f must be allowed by the fault set F of the OG statement. The rule asserts this by requiring that, when the fault is not allowed by the OG statement, the assertion r allows more states than the ones that do not satisfy the guard.

The last interesting rule is $Call$. The annotation environment Θ stores a list of annotation trees per function. Since a function can be called from multiple places and each call may require a different set of assertions, multiple annotation trees of the same function may be kept in the environment. $AnnCall$ provides a *routine index* to select which tree to use. When deriving $Call$, the annotation environment needs to be correctly initialised such that the requested annotation tree matches the function body. This way a derivable program cannot end in a $Stuck$ state because of an undefined function call. Ideally, this index would be computed by the translation from C to COMPLX.

4.3 Interference Freedom

As explained in section 2, interference freedom states that, for every atomic command c extracted from parallel components, all the commands it may be interleaved with have their assertions preserved by the execution of c. COMPLX's

interfree definition follows the same principle as Hoare-Parallel's, but with several important differences.

First, in order to support function calls we extract *assertions* and *atomics* using relations instead of functions. Extracting assertions and atomic commands from a program requires going through every statement including statements inside function bodies. To avoid divergence, the extraction functions must keep track of which functions have been processed. The resulting function must maintain a state and becomes hard to use when induction is required. In addition, COMPLX's separation between program and annotations makes it harder to extract assertions and atomics using a function. Since the annotation tree and the program structure are not synchronised by construction, a function would have to be partial or undefined if the annotation tree does not match the structure of the program. To address these issues, in COMPLX we use relations instead of functions to extract assertions and atomics. Using a relation, any mismatch between annotation tree and program structure simply results in the relation not holding, and the infinite-recursion problem goes away since the relation does not have to terminate. More importantly, by using a relation we can describe an infinite set of assertions/atomics, which is specifically required for $DynCom$.

Second, COMPLX's semantics is significantly more complicated than Hoare-Parallel's. In particular, as the COMPLX semantics executes the program, it reduces the program to a final command (i.e. $Skip$ or $Throw$) which denotes termination of execution. This is visible on most of the small-step semantics rules presented in section 3, such as *Parallel-Throw*, *Parallel-Skip*... Consequently, $Skip$ and $Throw$ commands have two purposes: they denote final configurations, and they also are legitimate commands that can be found in any given program. In the latter case, they must be annotated manually. However, in the former case the COMPLX framework must automatically generate assertions for them. Typically, the assertion on a $Skip$ will be the assertion of the next command, or, if it is the last command of the program, the post-condition. Hence, the relation that extracts assertions takes the post-condition and the abrupt-condition of the program and generates the appropriate assertions for every semantics rule that leads to a final configuration.

Finally, since COMPLX allows nested $Parallel$ statements, assertions need to be collected recursively on each of the parallel components.

4.4 VCG

In order to automate the creation of verification conditions for programs in COMPLX, we ported and extended Hoare-Parallel's VCG. We added support for several constructors, including $Catch$, $Call$, $Guard$ and $DynCom$. This involved writing Isabelle/HOL tactic rules to decompose the derivation of these commands and convert interference freedom goals to OG statements showing that assertions are preserved. As in Hoare-Parallel, most of the generated proof obligations get easily discharged using Isabelle/HOL automation. This makes our framework ideal for concurrency verification as finding the right correctness assertions should be the bulk of the work for verifying a concurrent program.

5. Soundness Proof

Verification using logic rules and a VCG is much more efficient than reasoning directly with the semantics, but it needs to be proven *sound* if we want to preserve the same level of trust. In this section we outline our proof that COMPLX's OG rules presented in section 4 are sound with respect to the semantics presented in section 3. Namely we prove, in Isabelle/HOL, the following theorem (identical to SIMPL's soundness theorem):

$$\Gamma, \Theta \vdash_{/F} a\ c\ \{\!|Q|\!\}, \{\!|A|\!\} \Longrightarrow \Gamma \models_{/F} (pre\ a)\ c\ \{\!|Q|\!\}, \{\!|A|\!\}$$

This states that any Hoare triple[3] that is *derivable* from the OG-rules is *valid*, where validity is defined in terms of the small-step semantics (below e.g. $Normal\ `\ \{\!|P|\!\}$ denotes the image of $\{\!|P|\!\}$ under $Normal$, embedding this $\{\!|P|\!\}$ into extended states):

$$\begin{array}{l}
\Gamma \models_{/F} \{\!|P|\!\}\ c\ \{\!|Q|\!\}, \{\!|A|\!\} \equiv \\
\forall s\ t\ c'. \\
\quad \Gamma \vdash (c, s) \rightarrow^* (c', t) \longrightarrow \\
\quad final\ (c', t) \longrightarrow \\
\quad s \in Normal\ `\ \{\!|P|\!\} \longrightarrow \\
\quad t \notin Fault\ `\ F \longrightarrow \\
\quad c' = Skip \wedge t \in Normal\ `\ \{\!|Q|\!\} \vee c' = Throw \wedge t \in Normal\ `\ \{\!|A|\!\}
\end{array}$$

That is, a program c is called *valid* if final states of any of its full executions without any faults from a state s satisfying P, satisfy the relevant post-condition. More precisely, if c executes, after multiple steps, into either $Skip$ or $Throw$ (denoted by *final*) then the final state t satisfies Q if c' is $Skip$ and A if c' is $Throw$. Note that a sequence of small-steps cannot reach both, $Skip$ and $Throw$. It is also worth noting, that as a consequence of separating annotations from programs, the notion of validity is purely semantical, thus completely independent from annotations which are only needed for derivability.

We now outline the main challenges in the soundness proof that proceeds by induction on the structure of the OG-rules. For the sake of brevity, in the following we will focus on the cases when all considered states are $Normal$: apart from these we get several corner cases, such as that guards can fail only within specified $Fault$ states or absence of undefined function calls. These are, however, of technical nature and do not contribute much to the structure and complexity of the overall proof.

All OG-rules, beside those for parallel composition and synchronisation, retain their SIMPL form, such that in these cases we proceed similarly to the sequential setting. This changes, of course, as soon as we reach the parallel composition and await cases.

[3] Technically, this is more a Hoare quadruple but we still use the more traditional term of triple.

The major challenges arise in the proof of the parallel case, i.e. when $c = Parallel\ cs$. We can assume all the premises of the OG-rule PARALLEL (see Figure 1), in particular that interference freedom holds for the parallel components cs with respect to the annotation. Moreover, we can assume $\Gamma \vdash \langle Parallel\ cs, Normal\ x\rangle \to^* \langle c', Normal\ y\rangle$ with x satisfying the annotated pre-condition and c' being $Skip$ or $Throw$. What we need to prove is that y satisfies the relevant post-condition. For this we induct on the closure of the small-step relation, and the challenge is to show that all the assumed conditions (from the premise of the OG rule, e.g. interference freedom) are preserved by each execution step (to be able to apply the induction hypothesis). This is a challenge because each step 'consumes' a part of the program, which needs to be reflected in the annotation tree. We capture this by a separate lemma, where we collect all the necessary properties relating pre- and post-configurations of any small-step. That is, if $\Gamma \vdash \langle c, Normal\ x\rangle \to \langle c', Normal\ y\rangle$ holds for any c, x, c', y, and the program c is derivable with an annotation structure a by the OG-rules such that s satisfies the annotated pre-condition, then we can find an annotation structure a' such that c' is derivable with a', y satisfies the pre-condition in a' and, moreover, any assertion or atomic of a' is an assertion or atomic of a, respectively. Since the program c is an arbitrary COMPLX program, we induct on the structure of c. Here again, only the await and parallel cases are more involved. For await we can rely on the canonical restriction that the body of any $Await$-construct is purely sequential, i.e. a SIMPL program in fact. In the parallel case, however, we have to deploy our interference freedom assumption to show that any post-state of the whole parallel construct will satisfy annotated conditions regardless of which of the constituting components does its small step. To this end we need to establish a connection between the small-step semantics and atomics as follows. Any program transition $\Gamma \vdash \langle c, Normal x\rangle \to \langle c', Normal y\rangle$, where $Normal\ x$ satisfies the annotated pre-condition and $x \neq y$, can only happen due to an atomic subcomponent cc of c that performs the step $\Gamma \vdash \langle cc, Normal\ x\rangle \to \langle Skip, Normal\ y\rangle$. Now, the interference freedom property states that each of such atomic steps preserves assertions of any component other than the one that performs the step. This gives us the preservation we need to carry assumptions over single steps of execution.

For the proof of the top-level $Await$ case we similarly can assume $\Gamma \vdash \langle Await\ b\ cc, Normal\ x\rangle \to^* \langle c', Normal\ y\rangle$ with x satisfying the annotated pre-condition, c' being $Skip$ or $Throw$, and $Await\ b\ cc$ being derivable by the OG-rules. Moreover, by induction hypothesis we also know that the await-body cc is valid. On the other hand, from the semantics of $Await$ we can conclude that y can only be obtained by a certain number of small-step transformations of $\langle cc, Normal\ x\rangle$ until a $Skip$ or $Throw$ configuration is reached, establishing the desired result.

6. Case Study

We used our COMPLX framework to reproduce the proof of correctness of a few examples of concurrent algorithms that had been verified within Hoare-Parallel, including the proof of Peterson's solution to the mutual exclusion problem [Prensa Nieto 2002]. Our proofs can be found online [COMPLX] and were very easily achieved once our framework was complete. This shows that COMPLX is robust and backward compatible with Hoare-Parallel, as none of the proofs required extra work. The VCG generates approximately the same number of proof obligations and discharging them takes a similar processing time. These examples, however, did not exercise any C-specific features.

To demonstrate the practicality of our framework in verifying concurrent C code, we created an example (also available online [COMPLX]) of a concurrent C program that exercises the specific features that COMPLX supports. In particular, our example uses exceptions, guards and function calls, all of which are not supported by Hoare-Parallel.

In our example, we extracted manually the program model from the C source code. The C program and the COMPLX program are both less than 20 lines, and the whole model is ≈230 lines of Isabelle/HOL definitions, including the complete set of assertions used to annotate the program and verify its correctness. The VCG generates 688 conditions and most of them are easily discharged using Isabelle/HOL automation. Once again, the bulk of the work lies in finding the right correctness assertions.

The aim of the program is to compute the combined sum of all the elements of multiple arrays. It does this by running a number of threads in parallel, each computing the sum of elements of one of the arrays, and then adding the result to a global variable $gsum$ shared by all threads. We restrict the example to two arrays and threads, but this could be generalised: we would then just need to generate accordingly more copies of the function $sumarr$, pairwise disjoint in local variables, such that each thread can invoke its own copy of $sumarr$. The correctness statement for this program is:

$\Gamma, \Theta \Vdash_{/F}$ {|*precond*|}
COBEGIN
SCHEME [$0 \leq m < 2$]
call-sumarr m
{|*local-postcond m*|}, {|*False*|}
COEND
{|*postcond*|}, {|*False*|}

The $SCHEME$ syntax models a parametric number of parallel programs. Here we use it to model the creation of two threads running concurrently, each calling the function $sumarr$. The post-condition (definition not shown) states that the global variable $gsum$ is indeed equal to the combined sum of all elements of all arrays. Since the function $sumarr$ cannot terminate with an exception, the abrupt-condition is false, which forces us to prove that all exceptions are caught. As explained in section 4, prov-

ing interference freedom also requires that we specify the post-condition (*local-postcond*) and abrupt-condition (false again) of the parallel component.

To begin we explain the state of the program, then how we model function calls, and finally how the $sumarr$ function is defined.

State The state of the program is modelled with the following record:

record sumarr_state =
(function arguments *)*
tarr :: "routine ⇒ word32 array"
tid :: "routine ⇒ word32"
(local variables of threads *)*
ti :: "routine ⇒ word32"
tsum :: "routine ⇒ word32"
(global variables *)*
garr :: "(word32 array) array"
gsum :: word32
gdone :: word32
glock :: nat

We now explain the need for the routine argument. Major challenges arise when attempting to verify parallel programs that make use of function calls. In a sequential context, a call to a function named f in a state s means that we just lookup the body of f in the procedure environment Γ, continue with the execution of Γf in the state s and return to the calling routine afterwards. In a concurrent setting, however, this execution could be interleaved with another call of f invoked by a different thread. Thus, if Γf uses some local variables, the model of the overall parallel program might not behave as in reality, as both invocations of f can interfere on the same local variables. Therefore, in our state, function arguments and local variables are modelled as a mapping from routine index to value. This allows concurrent executions of a function to use different instances of the variables.

In contrast, global variables are shared by all threads, so they are not protected by a routine index. We use $garr$ to store two arrays of machine 32-bit words that will be passed to each thread via argument $tarr$. The global variable $gsum$ is used to store the total result, whereas $tsum$ stores the local result used by each thread. The bit-field $gdone$ is used to indicate whether a thread has finished its computation. Finally, the threads use $glock$ as a mutually exclusive lock in order to protect the shared variables $gsum$ and $gdone$.

For the remainder of this section, $\{\!|...|\!\}$ denotes the places where assertions are required. To improve the readability and to highlight the similarity between input source and COMPLX model, we display our models without most of the assertions.

Function calls We define *call-sumarr* as shown in Figure 2. The parameter m is the routine index (from the

call-sumarr m ≡
{|*local-precond m*|} *CALLX* (*sumarr-init m*)
{|*sumarr-precond m*|} (''*sumarr*'', *m*) *m*
(*sumarr-restore m*) (λ- -. *Skip*)
{|*sumarr-restore-post m*|} {|*sumarr-return-post m*|}
{|*False*|} {|*False*|}

Figure 2: Annotations for the function $sumarr$.

callx init body restore return =
DynCom (λ*s. TRY*
init;; body
CATCH
restore s;; THROW
END;;
DynCom (λ*t. restore s;; return s t*))

Figure 3: Argument passing and scoping for function calls.

$SCHEME$) used to specify which copy of a local variable is accessed. $CALLX$ is syntactic sugar for calling a function while passing arguments and implementing scoping, i.e. saving and restoring local variables. The computation is done by a function $callx$ shown in Figure 3 ($CALLX$ combines it with annotations, as we will explain shortly). The process of calling a function involves several steps:

1. Saving the value of local variables by keeping a copy of the state.
2. Initialising local variables and function arguments by updating the state.
3. Executing the function body.
4. Restoring the value of local variables using the copy of the state.
5. Extracting the return value of the function from the state.

Saving and restoring local variables is required to support recursive functions and is equivalent to setting up and tearing down the stack frame in C. Steps 1 and 5 of function calls are both implemented using $DynCom$ as seen in Figure 3.

The first $DynCom$ is used to keep a copy of the state that is later used for restoring local variables. As we can see, $callx$ is complicated by exceptions that may cross function boundaries. When an exception is uncaught, the local variables must first be restored before the exception is propagated.

Steps 2 and 4 use the provided functions $init$ and $restore$. In our example, there are two arguments being passed to $sumarr$, the array $tarr$ and the thread identifier tid. The initialisation and restore functions are:

sumarr-init m ≡
λ*s*. *s*(|*tarr* := (*tarr s*)(*m* := (*garr s*)$_{[m]}$),
tid := (*tid s*)(*m* := *m* + *1*), *ti* := (*ti s*)(*m* := *undefined*),
tsum := (*tsum s*)(*m* := *undefined*)|)

sumarr-restore m ≡
λ*s t*. *t*(|*tarr* := (*tarr t*)(*m* := *tarr s m*), *tid* := (*tid t*)(*m* := *tid s m*),
ti := (*ti t*)(*m* := *ti s m*), *tsum* := (*tsum t*)(*m* := *tsum s m*)|)

To reason about function calls in OG, the user also has to provide assertions for every step in the control flow of $callx$. To facilitate this, there is an equivalent helper function ann_callx. The arguments of the $CALLX$ statement are then the commands for dealing with argument passing and scoping, the corresponding annotations, and the function name and index used when looking up the procedure and annotation environment.

One restriction of our current implementation of $CALLX$ is that both initialising and restoring local variables is performed in one step. This does not match the reality of C, which uses multiple instructions and can potentially be interleaved. We believe that it would be relatively straightforward to change our framework to use multiple steps, one for each local variable. Additionally, we should be able to create syntactic sugar that hides many of these details from the end user.

In Figure 2, *local-precond* corresponds to the first assertion of the parallel component, whereas *summar-precond* is the assertion once the arguments and local variables have been initialised. The other assertions describe the state before restoring local variables (*summar-restore*) and before extracting the return value from the state (*summar-return*), for normal termination of the function. For abrupt termination these assertions are false, as the function $sumarr$ cannot terminate with an exception.

The $sumarr$ ***function*** Returning to our case study, Figure 4a shows the body of the sumarr function, as it would be implemented in C. The pointer tarr refers to the array of unsigned integers that is being summed and tid is a thread identifier of value 1 or 2 depending on the thread.

The loop calculates the sum of each element of the array and at each iteration if the sum or the current element exceeds MAXSUM, we break out of the loop and cap tsum at MAXSUM. This prevents potential word overflows, assuming that NSUM and MAXSUM are chosen appropriately. After the loop, we invoke the lock and unlock functions to protect the global variables gsum and gdone. In the C code, lock and unlock are implemented using a mutex.

Figure 4b shows the COMPLX model in Isabelle/HOL of this function. As seen in subsection 4.1, COMPLX's syntactic sugar allows the program to be directly annotated. Additionally, concurrent executions of functions require different instances of any local variables. Therefore, in Figure 4b we parametrise $sumarr$ with m, which is used as a routine index to select which local variables to use.

The first two lines initialise the local variables $tsum$ and ti to zero. Note that a for-loop is trivially converted to a while-loop by moving the loop counter update to the end of the loop body. The while-loop is wrapped inside a try-catch statement because COMPLX, just like SIMPL, uses exceptions to model early exit of a loop. Hence, the break statement present in the C code is replaced with a $Throw$ in COMPLX, and the $CATCH$ block only contains a $Skip$, i.e. do nothing. This means that when the exception is thrown, it has the effect of breaking out of the loop.

One of the most important aspects of verifying C code is proving the absence of behaviours undefined by the C standard. In both SIMPL and COMPLX undefined behaviours are usually specified using guards. For instance, in our case there could be undefined behaviour if an invalid pointer is dereferenced. Therefore, every time a pointer is accessed it must be protected by a guard forcing us to prove that the pointer is indeed valid.

In Figure 4b, since $tarr$ is a pointer, every access is guarded with an *array-in-bound* check which guarantees that $tarr$ is a valid pointer and that the index ti is less than the length of the array. If the guard is not satisfied the program returns the fault $InvalidMem$ indicating an invalid memory access. Having explicit guard commands in COMPLX also allows us to reason about concurrent programs that can actually end in a specific $Fault$ state. For instance, we are able to prove that in some circumstances a program is guaranteed to result in a $Fault$ state.

After the loop, the model calls the function $lock$, to acquire the global mutex that protects the shared variables $gsum$ and $gdone$. Since $lock$ and $unlock$ only access a global variable (the mutex) and do not take any arguments, their call does not require saving and restoring of local variables. The definition of $lock$ and $unlock$ follows:

lock m ≡ {|...|} *AWAIT glock = 0 THEN glock* := *1 END*

unlock m ≡ {|...|} *glock* := *0*

The mutex is modelled by using $glock$, a global variable set to 1 when the lock is held and 0 otherwise. The semantics of $Await$ guarantees that only one thread can be inside the lock at a time.

Summary COMPLX was designed to reason about an accurate representation of the C code, without requiring that programmers radically change their programming style and habits. A key feature of COMPLX is the syntactic sugar which allows annotating programs directly on the program model, despite having a separate datatype for annotation tree and program. This gives us the best of both worlds: a user-friendly framework for annotating program and a neat language abstract syntax which is not cluttered with irrelevant annotations. The lack of exceptions in Hoare-Parallel would force reimplementing our code to avoid having a `break`

```
void sumarr(unsigned int *tarr,
            unsigned int tid)
{
    unsigned int ti;
    unsigned int tsum;

    tsum = 0;
    for (ti = 0; ti < NSUM; ti++) {
            tsum += tarr[ti];
            if (tsum >= MAXSUM ||
                tarr[ti] >= MAXSUM) {
                tsum = MAXSUM;
                break;
            }
    }
    lock();
    gsum += tsum;
    gdone |= tid;
    unlock();
}
```

(a) C code

sumarr m ≡
{|...|} *tsum* := *tsum*(*m* := *0*);;
{|...|} *ti* := *ti*(*m* := *0*);;
TRY {|...|} *WHILE ti m* < *NSUM INV* {|...|}
  DO {|...|} (*InvalidMem*, {|*array-in-bound* (*tarr m*) (*ti m*)|}) ⟼
    {|...|} *tsum* := *tsum*(*m* := *tsum m* + *array-nth* (*tarr m*) (*ti m*));;
  {|...|} (*InvalidMem*, {|*array-in-bound* (*tarr m*) (*ti m*)|}) ⟼
    {|...|} *IF MAXSUM* ≤ *tsum m* ∨ *MAXSUM* ≤ *array-nth* (*tarr m*) (*ti m*)
    THEN {|...|} *tsum* := *tsum*(*m* := *MAXSUM*);;
      {|...|} *THROW*
    ELSE {|...|} *SKIP*
    FI;;
  {|...|} *ti* := *ti*(*m* := *ti m* + *1*)
  OD
CATCH {|...|} *SKIP END*;;
{|...|} *SCALL* (''*lock*'', *0*) *m*;;
{|...|} *gsum* := *gsum* + *tsum m*;;
{|...|} *gdone* := (*gdone OR tid m*);;
{|...|} *SCALL* (''*unlock*'', *0*) *m*

(b) COMPLX model

Figure 4: The C code and matching COMPLX model of our case study

statement in the middle of the loop. Furthermore, support for guard statements is critical to enable C verification, because they ensure that the semantics of the code is defined. Finally, supporting function calls is a key requirement for our framework, which we intend to use on larger scale verification projects.

7. Related Work

Logics for Concurrency Over the years, many logics were developed for reasoning about concurrency, the oldest and most straightforward of which is OG. OG provides, however, no modular way to reason about memory and quickly leads to an explosion in the number of verification conditions that need to be proven. Rely-Guarantee (RG) [Jones 1983], Concurrent Separation Logic (CSL) [OHearn 2007], and a number of more recent combinations and extensions of these (e.g. [Vafeiadis 2008; da Rocha Pinto et al. 2014]) have been developed since to overcome the modularity issues of OG. The separation-based logics typically rely on ownership over shared state: threads need to lock their accesses to shared state and ownership can be transferred along acquire/release atomic memory accesses.

In recent work [Andronick et al. 2015, 2016], we found that using the simple OG method is suitable for our reasoning of interrupt-induced concurrency in *racy* OS code. In that code, the OS API functions, the scheduler and the interrupt handlers all concurrently modify shared variables without any synchronisation (in order to meet stringent low-latency requirements). The correctness argument needs to rely on fine-grain assertions at these sharing points; it cannot rely on some atomicity or ownership argument. We used OG (at the simple high-level language IMP) and we introduced a technique that we called *await-painting*, essentially painting our program with $Await$ statements to limit the concurrency to places where it actually occurs. This technique allowed us to proof-engineer an Isabelle/HOL tactic that automatically discharges most of the verification conditions generated by OG. We successfully used the tactic to verify an abstract model of the eChronos OS scheduling behaviour. COMPLX will enable us to extend this verification to the C implementation.

In COMPLX, just like SIMPL, the notion of state is abstract: we propose a generic language for reasoning about concurrent imperative code. Modelling memory is orthogonal to the work done in this paper. The C-to-Isabelle parser defines a concrete notion of state on top of SIMPL which, for instance, can be reasoned about using separation logic [Tuch et al. 2007]. Building a framework in the spirit of FCSL (fine-grained concurrent separation logic) [Nanevski et al. 2014; Sergey et al. 2015] on top of the COMPLX language would be interesting future work.

Recent work showed that, as is, OG is unsound for weak memory models but can be extended in a sound logic by strengthening its interference-freedom condition [Lahav and Vafeiadis 2015]. We could look at a similar approach if we want to support weak memory models in the future.

Tools for Verification of Concurrent C We focus here on tools that allow the verification of specific properties or specifications, rather than e.g. static analysers that can only detect specific classes of errors. VCC [Cohen et al. 2009] is an industrial-strength verification environment for low-level concurrent system code. It is an assertional, auto-

matic, deductive code verifier for C, where specifications in the form of function contracts, data invariants, and loop invariants are added to the C code to guide VCC. From the annotated program, VCC generates verification conditions, which it then tries to discharge using the automatic theorem prover Z3 [de Moura and Bjørner 2008] or through the Boogie verifier [Barnett et al. 2006]. VCC has been used, among others, to verify the Microsoft Hyper-V hypervisor and has also been used in the Verisoft XT project [Verisoft XT]. Moreover, Isabelle/HOL was used as a backend to VCC for the verification of certifying graph algorithms from LEDA [Alkassar et al. 2014]. When the C-to-Isabelle parser was open-sourced, the LEDA verification project switched to only using SIMPL and the C-to-Isabelle parser and redid their verification completely within Isabelle/HOL, in order to provide higher trust guarantees [Noschinski et al. 2014; Rizkallah 2015]. Unlike an LCF based theorem prover (e.g. Isabelle/HOL or Coq [Bertot and Castéran 2004]), VCC relies on a large trusted computing base that includes the entire VCC engine and Z3 [Noschinski et al. 2014; Rizkallah 2015]. Similar to VCC, VeriFast [Jacobs et al. 2010] relies on Z3. We would like to enable reasoning about concurrency, within an LCF based theorem prover in order not to compromise on trust.

A number of recent efforts provide tools to reason about concurrency in Coq. Most of these efforts are based on CSL and primarily focus on modular reasoning about non-racy shared memory. The Verified Software Toolchain [Appel 2012] provides machine-checked guarantees that the CSL assertions about concurrent C code with primitive lock operations hold down to the machine-language program. Iris [Jung et al. 2015, 2016] is a general and expressive logic with a simple set of verified primitive mechanisms and proof rules for modular reasoning about shared memory. Once again, we are targeting potentially-racy high-performance code for which OG fine-grain assertions are well-suited.

8. Conclusion and Future Work

In this paper we have presented our COMPLX framework for sound verification of concurrent imperative code in Isabelle/HOL. We have emphasised how we use the Owicki-Gries method in order to extend the SIMPL tool to cope with concurrency. This way our framework inherits support for function calls and exception handling from SIMPL. The presented case-study illustrates how these features can be utilised in practical verification.

Future work includes more proof engineering to increase ease of use, integration with the C-to-Isabelle parser, and definition of more concrete notions of states.

With the work presented here, we bridge the gap between the verification of abstract algorithms and that of their imperative implementations. We plan to extend the C-to-Isabelle parser to translate C code into COMPLX code to provide guarantees for concurrent low-level C code, with the aim to verify concurrent operating systems, such as the interruptible eChronos embedded operating system or multicore seL4.

Acknowledgments

This research is supported by the Air Force Office of Scientific Research, Asian Office of Aerospace Research and Development (AOARD) and U.S. Army International Technology Center - Pacific under grant FA2386-15-1-4055.

References

E. Alkassar, S. Böhme, K. Mehlhorn, and C. Rizkallah. A framework for the verification of certifying computations. *Journal of Automated Reasoning*, 52(3):241–273, 2014. ISSN 0168-7433. doi: 10.1007/s10817-013-9289-2.

S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. Cogent: Verifying high-assurance file system implementations. In *ASPLOS*, Atlanta, GA, USA, Apr 2016. doi: 10.1145/2872362.2872404.

J. Andronick, C. Lewis, and C. Morgan. Controlled Owicki-Gries concurrency: Reasoning about the preemptible eChronos embedded operating system. In *Workshop on Models for Formal Analysis of Real Systems (MARS)*, 2015.

J. Andronick, C. Lewis, D. Matichuk, C. Morgan, and C. Rizkallah. Proof of OS scheduling behavior in the presence of interrupt-induced concurrency. In *International Conference on Interactive Theorem Proving*, Nancy, France, aug 2016.

A. W. Appel. Verified software toolchain. In A. Goodloe and S. Person, editors, *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, volume 7226 of *Lecture Notes in Computer Science*, page 2. Springer, 2012.

M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: a modular reusable verifier for object-oriented programs. In *4th FMCO*, volume 4111 of *LNCS*, pages 364–387, Amsterdam, The Netherlands, 2006. Springer.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN 3-540-20854-2. doi: 10.1007/978-3-662-07964-5.

E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *22nd TPHOLs*, volume 5674 of *LNCS*, pages 23–42, Munich, Germany, 2009. Springer. doi: 10.1007/978-3-642-03359-9_2.

COMPLX. cf. `http://ts.data61.csiro.au/projects/concurrency/complx`.

P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A logic for time and data abstraction. volume 8586 of *LNCS*, pages 207–231. Springer, 2014. doi: 10.1007/978-3-662-44202-9_9. URL `http://dx.doi.org/10.1007/978-3-662-44202-9_9`.

L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340, Budapest, Hungary, Mar 2008. Springer. ISBN 978-3-540-78799-0. doi: 10.1007/978-3-540-78800-3_24.

B. Jacobs, J. Smans, and F. Piessens. A quick tour of the verifast program verifier. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, pages 304–311, 2010.

C. B. Jones. Tentative steps towards a development method for interfering programs. *Trans. Progr. Lang. & Syst.*, 5(4):596–619, 1983.

R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015.

R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer. Higher-order ghost state. In J. Garrigue, G. Keller, and E. Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 256–269. ACM, 2016.

G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *CACM*, 53(6):107–115, Jun 2010. doi: 10.1145/1743546.1743574.

O. Lahav and V. Vafeiadis. *Owicki-Gries Reasoning for Weak Memory Models*, pages 311–323. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

T. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. Noninterference for operating system kernels. In Chris Hawblitzel and Dale Miller, editor, *CPP*, pages 126–142, Kyoto, Dec 2012. Springer. ISBN 978-3-642-35307-9.

A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *European Symposium on Programming Languages and Systems*, pages 290–310. Springer, 2014.

T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. ISBN 978-3-540-43376-7. doi: 10.1007/3-540-45949-9.

L. Noschinski, C. Rizkallah, and K. Mehlhorn. Verification of certifying computations through AutoCorres and Simpl. In J. M. Badger and K. Y. Rozier, editors, *NASA Formal Methods*, volume 8430 of *LNCS*, pages 46–61. Springer, 2014.

P. W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, Apr 2007. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.12.035.

S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

L. Prensa Nieto. *Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL*. PhD thesis, T.U. München, 2002.

C. Rizkallah. A Simpl shortest path checker verification. In *Proceedings of Isabelle Workshop 2014*, 2014.

C. Rizkallah. *Verification of program computations*. PhD thesis, Saarland University, 2015. URL `http://scidok.sulb.uni-saarland.de/volltexte/2015/6254/`.

N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

N. Schirmer. A sequential imperative programming language syntax, semantics, hoare logics and verification environment. *Archive of Formal Proofs*, Feb 2008. ISSN 2150-914x. `http://isa-afp.org/entries/Simpl.shtml`, Formal proof development.

I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *ACM SIGPLAN Notices*, volume 50, pages 77–87. ACM, 2015.

H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editor, *POPL*, pages 97–108, Nice, France, Jan 2007. ACM. ISBN 1-59593-575-4.

V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, Computer Laboratory, jul 2008.

Verisoft XT. Verisoft XT. `http://www.verisoftxt.de`, 2010.