CSimpl: A Rely-Guarantee-based Framework for Verifying Concurrent Programs

David Sanán[1], Yongwang Zhao[1,2], Zhe Hou[1], Fuyuan Zhang[1], Alwen Tiu[1], and Yang Liu[1]

[1] School of Computer Science and Engineering, Nanyang Technological University, Singapore
[2] School of Computer Science and Engineering, Beihang University, Beijing, China

Abstract. It is essential to deal with the interference of the environment between programs in concurrent program verification. This has led to the development of concurrent program reasoning techniques such as Rely-Guarantee. However, the source code of the programs to be verified often involves language features such as exceptions and procedures which are not supported by the existing mechanizations of those concurrent reasoning techniques. Schirmer et al. have solved a similar problem for sequential programs by developing a verification framework called Simpl, which provides a rich sequential language that can encode most of the features in real world programming languages. Since Simpl only aims to verify sequential programs, it does not support the specification nor the verification of concurrent programs. In this paper we introduce CSimpl, an extension of Simpl with concurrency-oriented language features and verification techniques. We prove the compositionality of the CSimpl semantics and we provide inference rules for the language constructors to reason about CSimpl programs using Rely-Guarantee. We show that the inference rules are sound w.r.t. the language semantics. Finally, we run a case study where we use CSimpl to specify and prove functional correctness of an abstract communication model of the XtratuM partitioning separation micro-kernel.

1 Introduction

In the past two decades, formal methods have been successfully applied in the verification of many critical systems. Verification of sequential programs has gained much attention both in academia and in industry, and now there is a reasonably strong tool support in this area. However, nowadays critical and high-assurance systems are often designed for multi-core architectures where multiple processes run in parallel, but verification techniques and tools for concurrent programs are relatively less developed than those for sequential programs. One of the main issues to be solved is the verification of implementation level programs that involve concurrency; this is explained below.

To ensure the reliability of computer systems, verification of functional correctness and security properties must be applied not only at the specification level, but also at the implementation or even at the machine code level. Verification at such low levels requires modelling languages that are able to capture the features in programming languages such as exceptions and procedure calls. On the other hand, most existing techniques for the verification of concurrent programs stem from Owicki and Gries's work [11] which introduced techniques for the verification of parallel programs. Later Jones [4] introduced Rely-Guarantee to improve Owicki and Gries's method by allowing compositional verification. The Owicki-Gries method has been mechanized in the Isabelle/HOL theorem prover in [10], and Jones's Rely-Guarantee method has been mechanized in Isabelle/HOL in [9] which follows the specification in [14]. Also, [1] models in Isabelle/HOL an algebraic specification of Rely-Guarantee. Although the languages used in previous mechanizations of the above mentioned methods are suitable for verifying system specifications, many implementations cannot be directly captured in those mechanizations. Therefore there is a need to develop a richer modelling language to accurately capture the behaviour of programs at the implementation level.

Simpl [12] is a while-language that supports most of the features of real world programming languages. The syntax and semantics of Simpl are modelled in Isabelle/HOL and Simpl has been used in the verification of seL4 source code [8]. However, its design aims only at reasoning about sequential programs, consequently, this language lacks constructors for parallel composition of programs. Moreover, its proof system is based on Hoare Logic for the verification of sequential languages, which cannot be used for reasoning about concurrent programs.

Building on the Simpl framework and the Rely-Guarantee method, we develop a formal verification framework in Isabelle/HOL, called CSimpl, for verifying partial correctness of high-assurance concurrent systems. CSimpl extends the work in [9, 14] with a richer (sequential) language based on Simpl and concurrent constructors. The main contributions of this paper are as below:

(1) We extend Simpl using the notion of computation [9, 14] to introduce parallelism in two layers: the bottom layer is the execution of sequential Simpl programs extended with a synchronization primitive `Await` over shared variables; and the top layer is the parallel execution of the bottom layer programs by means of a parallel composition operator. While existing Rely-Guarantee methods are mechanized for reasoning about abstract specification languages [9, 10], our method goes one step further and covers most of the features of the C99 language. CSimpl is able to express system language features such as exceptions, procedures, and dynamic programming, among others.

(2) We define a compositional semantics of Rely-Guarantee for CSimpl. We also provide a set of inference rules for the Rely-Guarantee proof system and we prove that they are sound w.r.t. the semantics. The rich expressibility of CSimpl means that the number of inference rules of the Rely-Guarantee proof system is much higher and their complexity is significantly increased. The CSimpl semantics, the Rely-Guarantee proof system specification and its soundness proof comprise more than 15k lines of proof and specification in Isabelle/HOL and Isar[3].

(3) As a case study, we specify in CSimpl two XtratuM [3] services for queuing inter-partition communication and we prove the correctness of an invariant on the queuing communication structure. Inter-partition communication is the mechanism used to implement information flow and is critical in proving event-based non-interference. Xstratum is a separation micro-kernel for space and time partitioning of applications. In its latest version XtratuM support multi-core architectures, allowing one to run appli-

[3] Due to space reasons we only show some excerpts of the semantics and proofs, the whole model can be found at: http://securify.sce.ntu.edu.sg/MicroVer/CSimpl.zip

```
type_synonym 's bexp = "'s set"
datatype ('s, 'p, 'f) CSimpl.com =
  Skip | Throw | Basic "'s ⇒ 's" | Spec "('s × 's) set"
  | Seq "('s ,'p, 'f) com" "('s,'p, 'f) com"
  | Cond "'s bexp" "('s,'p,'f) com"  "('s,'p,'f) com"
  | While "'s bexp" "('s,'p,'f) com" | Call "'p"
  | DynCom "'s ⇒ ('s,'p,'f) com"
  | Guard "'f" "'s bexp" "('s,'p,'f) com"
  | Catch "('s,'p,'f) com" "('s,'p,'f) com"
  | Await "'s bexp" "('s,'p,'f) Simpl.com"
datatype ('s,'f) xstate = Normal 's | Abrupt 's | Fault 'f | Stuck
type_synonym('s,'p,'f) config = "('s,'p,'f)com  × ('s,'f) xstate"
type_synonym ('s,'p,'f) body = "'p ⇒ ('s,'p,'f) com option"
type_synonym('s,'p,'f) par_Simpl = "('s,'p,'f)CSimpl.com  list"
```

Fig. 1. Syntax and state definition of the CSimpl Language

cations in parallel in multiple cores. Using our new Rely-Guarantee proof system, we prove that the specification of the inter-partition communication services correctly introduces and removes messages in the communication channel. The specification and the proofs comprise 3500 lines of formalization. This is to the best of our knowledge the first attempt on the verification of separation microkernels targeting multi-core architectures. Other works such as [15, 16] verify functional correctness and non-interference for sequential micro-kernels, and the work in [2] focuses on the verification of sequential applications using the ARINC standard.

2 CSimpl Language

2.1 Simpl Overview

Schirmer introduces in [12] a verification framework for imperative sequential programs developed in Isabelle/HOL. The verification framework includes a generic imperative language, called Simpl, which is composed of the necessary constructors to capture most of the features present in common sequential languages, such as conditional branching, loops, abrupt termination and exceptions, assertions, mutually recursive functions, expressions with side effects, and nondeterminism. Additionally, Simpl can express memory related features like the memory heap, pointers, and pointers to functions. The Simpl verification framework also includes a Floyd/Hoare-like logic to reason about partial and total correctness, and on top of it, the framework implements a verification condition generator (VCG) to ease the verification process.

In order to capture all the aspects of abrupt termination, assertions, and function calls, the program state `'s` in Simpl is modelled in Isabelle/HOL as a datatype `xstate` (shown in Fig. 1), which is composed of four different constructors: `Normal 's`, representing a regular execution; `Fault 'f`, representing a failed assertion; `Abrupt 's`, representing an exceptional state; and `Stuck`, representing a state where a call to a non-defined function is made. Additionally, the semantics requires an environment Γ containing procedure definitions, i.e., is a partial function from the set `'p` of procedure names to the body of the procedures. Both features regarding the state and procedures definitions are used in CSimpl.

2.2 CSimpl Syntax

The syntax of CSimpl is shown in Fig. 1. CSimpl extends Simpl by adding two constructors for concurrency: `Await`, which takes two parameters `cond` and `body`, and Parallel Composition. `Await` allows synchronization of processes under the boolean condition `cond` and then it atomically executes `body`, which is a pure sequential Simpl program. This allows us to use Hoare logic for sequential programs and the original Simpl VCG in the verification of the atomic blocks. Parallel composition happens at the top layer (root program), and it can not be nested with other constructors like in the approach followed [6]. Therefore, a parallel program launches n sequential programs that are executed concurrently and that do not create new concurrent threads. A parallel program is defined as a list of sequential programs. Since we are aiming the verification of programs without dynamic creation of process, this approach is not a problem for our goal and simplify the mechanization.

CSimpl's syntax, following the syntax of Simpl, is defined in terms of states, of type `'s`; a set of fault types, of type `'f`; and a set of procedure names of type `'p`. The constructor `Skip` indicates program termination; `Seq s1 s2`, `Cond b c1 c2`, and `While b c` are respectively the standard constructors for sequential, conditional, and loop statements. `Throw` and `Throw c1 c2` are the complements for abrupt termination of programs of `Skip` and `Seq c1 c2`, and they allow to model exceptions. `Call p` invokes a procedure where `p` is a procedure name; `Guard f g c` represents assertions, where c is executed if the guard `g` holds in the current state, fault of type `'f` is raised otherwise. Finally, `Spec r` and `DynCom cs` respectively introduce a nondeterministic behavior expressed by relation `r`, and a state dependent dynamic command transformation which is used to model blocks and functions with arguments.

2.3 CSimpl Semantics

The small step operational semantics of CSimpl is a predicate inductively defined based on an environment for procedures Γ and a pair of program configurations `((P,s), (P',s'))` where program P in state s, transits to P' and state s'. It is represented as $\Gamma\vdash_c$(P,s) $\rightarrow$ (P',s'), where the small c indicates it is a step transition in CSimpl. A CSimpl configuration is defined as a tuple `(P,s)` where `P` is a CSimpl program and `s` is an `xstate`. A Component CSimpl configuration `(p,s)` is called `final` if `p` = `Skip` or `p` = `Throw` and there exists a state s' such that s = `Normal` s'. A `final` configuration cannot progress to another configuration.

CSimpl extends Simpl with rules for synchronization on shared variables, `Await`, and the parallel computation shown below. For space reason we only provide the small step semantics rules Await and AwaitAb for the `Await` command (Fig. 2). The rest are similar to those defined in [12].

The `Await` rules leverage Simpl's big step semantics to atomically transit from the initial configuration (p,s) to the next state t resulting from the execution of p from s and it is expressed as $\Gamma\vdash\langle p,s\rangle \Rightarrow t$. The two rules express the situation where from a current state s satisfying the synchronization condition, the atomic program in Simpl ends in a state t that can be an abrupt state as a result of an exception thrown in the sequential program for the rule `AwaitAb`, or any other possible state for the rule `Await`.

$$\frac{\begin{array}{c} s \in b \quad \Gamma_{\neg a} \vdash \langle c, Normal\ s\rangle \Rightarrow t \\ \neg(\exists t'. t = Abrupt\ t') \end{array}}{\Gamma \vdash_c (Await\ b\ p, Normal\ s) \to (Skip, t)}\ \textsc{Await} \qquad \frac{}{\Gamma \vdash_c (P, Normal\ s) \to_e (P, t)}\ \textsc{Env}$$

$$\frac{s \in b \quad \Gamma_{\neg a} \vdash \langle c, Normal\ s\rangle \Rightarrow t \quad t = Abrupt\ t'}{\Gamma \vdash_c (Await\ b\ p, Normal\ s) \to (Throw, Normal\ t')}\ \textsc{AwaitAb} \qquad \frac{\forall t'. t \neq Normal\ t'}{\Gamma \vdash_c (P, s) \to_e (P, s)}\ \textsc{Env\_n}$$

$$\frac{i < length\ Ps \quad \Gamma \vdash_c (Ps!i, s) \to (r, t)}{\Gamma \vdash_p (Ps, s) \to (Ps[i := r], t)}\ \textsc{Par} \qquad \frac{}{\Gamma \vdash_p (Ps, Normal\ s) \to_e (Ps, Normal\ t)}\ \textsc{P\_Env}$$

Fig. 2. Small Step and Environment CSimpl Semantic rules

This distinction is necessary since a Simpl program can finish in an `Abrupt` state, however the small step semantics does not use the state `Abrupt`. Instead, a CSimpl program finishes in an exception state when the last configuration of a computation is a pair composed of the program `Throw`, together with a `Normal` state. Note that big step transitions use sequential Simpl programs, therefore the environment in the atomic step has to be a function from procedure names to Simpl programs, which do not contain `Await` instructions (for the same reason the body of `Await` cannot contain nested `Await` neither). $\Gamma_{\neg a}$ translates bodies of procedures in Γ into Simpl programs if they do not contain any `Await` instruction, removing from Γ those procedures containing `Await` instructions. $Ps!i$ means accessing the i element in the list Ps, whilst $Ps[i := r]$ means substitute the i element in Ps for r.

A Parallel CSimpl configuration is defined as a tuple `(Ps,s)` where `Ps` is a list of CSimpl programs and `s` is an `xstate`. Parallel CSimpl semantics is inductively defined by means of rule `PAR` in Fig. 2. A parallel configuration `(Ps,s)` transits to another parallel configuration `(Ps[i:=r], s')` when there is a program `i` in `Ps` such that $\Gamma \vdash_c$(Ps!i,s) $\to$ (r,s'). It is represented with $\Gamma \vdash_p$(Ps, s) $\to$ (Ps[i:=r], s'). Similarly to component configurations, a parallel CSimpl configuration (xs, s) is called final if $xs \neq []$ and for all $i < length\ xs. final(xs!i, s)$.

Together with the semantic representing component transitions, it is necessary to define semantics for environment transitions. They are inductively defined using rules `Env` and `Env_n` in Fig. 2, where the small e is to express that it is an environment transition. CSimpl semantics for components can transit from a `Normal` state to a different type. However it is not possible to transit from a non `Normal` state to a different type of state, i.e. $\Gamma \vdash_p (P, Stuck) \to (P', Normal\ t)$. Moreover, the component semantics always transits from a configuration `(p,s)` with $\mathtt{p} = \mathtt{Skip}$ and $\nexists \mathtt{s}'. \mathtt{s} = \mathtt{Normal}\ \mathtt{s}'$ to a final transition `(Skip,s)`. Therefore, the environment at the sequential layer needs to model this behaviour in rules `Env` and `Env_n` in order to make the semantics at the parallel layer compositional. Environment transitions at the parallel level are defined in such a way that they can transit from a `Normal` state to another `Normal` state as shown in rule `P_ENV` in Fig. 2.

3 Rely-Guarantee for CSimpl

Rely-Guarantee [6] extends the specification of a program with two relations R and G characterizing, respectively, how the environment interferes with the program (Rely) and how the program modifies the environment (Guarantee). Therefore a specification for the verification of parallel systems using Rely-Guarantee is composed of four elements: precondition, postcondition, rely, and guarantee.

In contrast to the previous approaches of Rely-Guarantee [9], a state `xstate` in CSimpl could be of multiple forms. This requires to modify the definitions of computation, parallel computation and conjoin, since these definitions need to consider the environment for procedures. Additionally, it is necessary to modify the definitions for commitment, assumption, validities for components and parallel execution to consider the environment, the set of `Fault` states that have been checked to be non reachable, the state definition considering different types of states, and the dual postcondition for `Normal` and `Abrupt` states. With regard to the proof system itself, a total of 8 new rules have been added to the work in [9] to deal with all the language constructors present in Simpl. Finally, soundness of the axiomatic rules for the proof system w.r.t. the specification of validity is proven. Soundness of the rely guarantee proof system is considerably more complex and larger than the work in [9]. While the work in [9] consists of around 2300 lines of proofs and specification, the current work consists of more than 13000 lines of proofs and specification. In particular, the multiple forms of states considerably increase the complexity of the proofs.

3.1 Definition of Computation for CSimpl

The formal validity of a rely-guarantee tuple in this work is based on the definition of computation, which is the set of all possible sequences of configurations resulting of transiting the component or the environment, starting from an initial configuration.

Definition 1 (Sequential Component Computation). *A computation is a tuple (Γ, confs) where Γ is an environment for procedures and confs is a list of sequential configurations. The set of all possible computation cptn is inductively defined as follows:*

- *(Γ, [(P,s)]) $\in$ cptn*
- *if $\Gamma\vdash_c$(P,s) $\rightarrow_e$ (P,t) and (Γ,(P, t)#xs) $\in$ cptn then (Γ,(P,s)#(P,t)#xs) $\in$ cptn*
- *if $\Gamma\vdash_c$(P,s) $\rightarrow$ (Q,t) and $\Gamma\vdash_c$(P,s) $\rightarrow$ (Q,t); (Γ,(Q, t)#xs) $\in$ cptn then (Γ,(P,s)#(Q,t)#xs) $\in$ cptn*

Definition 2 (cp Γ P s). *The set of possible computations of an environment for procedures Γ starting from an initial configuration (P,s) is the set (Γ,l) such that $l!0 = (P,s)$ and $(\Gamma,l) \in$ cptn.*

Definition 3 ($\propto$). *conjoin [14] represented by $\propto$, defines an equivalence relation between a parallel computation p of n CSimpl components and a list clist of n component computations, where for all $i < n$. $clist!i = (\Gamma_i, cptn_i)$. ($\Gamma$, p)$\propto$ clist iff:*

- *for all $i < n$, the length of $cptn_i$ is equal to the length of p and $\Gamma_i = \Gamma$.*
- *for all $i < n$ and $k <$ length p, $(cptn_i)!k = (c_i^k, s_i^k)$ and $p!k = (cs, s)$ with $cs!i = c_i^k$ and $s = s_i^k$.*
- *for all k such that Suc $k <$ length p, if $\Gamma \vdash_p p!k \rightarrow_e p!(Suc\ k)$, then $\forall i < n\ \Gamma_i \vdash (cptn_i)!k \rightarrow_e (cptn_i)!k$; if $\Gamma \vdash_p p!k \rightarrow p!(Suc\ k)$ then $\exists i.i < n$ where $\Gamma_i \vdash (cptn_i)!k \rightarrow (cptn_i)!(Suc\ k)$ and $\forall j.\ j \neq i\ \Gamma_j \vdash (cptn_j)!k \rightarrow_e (cptn_j)!(Suc\ k)$.*

The last condition of conjoin states that for any step k in p, if k is an environment step in p, then k is also an environment step in all $cptn_i$; and if it is a component step, then there is some $cptn_i$ where k is a component step and for any other $cptn_j$, with $j \neq i$, k is an environment step. Using conjoin and the set of computations and parallel computations we prove the compositionality of the semantic of CSimpl:

Theorem 1 (CSimpl Compositionality).

$$xs \neq [] \Longrightarrow par\_cp\ \Gamma\, xs\ s = \{(\Gamma_1, c).\Gamma_1 = \Gamma \wedge \\ \exists clist.(length\ clist) = (length\ xs) \wedge (\Gamma, c) \propto clist\}$$

Theorem 1 states that given a parallel configuration (xs, s) such that xs is not empty, then for any parallel computation (Γ, c) starting from (xs, s) there is a list of component computations $clist$ with the same length of xs and $(\Gamma, c) \propto clist$. That is, the execution of a parallel number of components $c_0 \ldots c_n$ can be expressed as the execution of one sequential component $i < n$ where the execution of any component $j \neq i$ is simulated by a component environment transition. The set of parallel computations `par_cp P s` Γ is defined similarly to `cp` using parallel configurations. The right and left implications of the equality in theorem 1 are proven first by induction on the parallel computation and the by cases on the type of parallel and component events using conjoin.

3.2 Validity of Formulas for Rely-Guarantee in CSimpl

Based on the Rely-Guarantee definitions, we define the validity of a Rely-Guarantee tuple from the set of all possible computations from an initial configuration. This uses the notions of *assumption* of preconditions and the environment, and *commitment* of the component and the postcondition.

Definition 4 (assum(pre,rely)). *The assumption of a predicate pre and an environment relation rely for an environment of procedures Γ is the set of component computations $(\Gamma, cptn)$ such that $cptn!0 = Normal\ s$ and $s \in pre$, and for any step transition in the computation $\Gamma \vdash_c (cptn)!k \rightarrow_e (cptn)!(Suc\ k)$ where $Suc\ k < length\ cptn$, $cptn!k = (p_k, s_k)$, and $cptn!(Suc\ k) = (p_{k+1}, s_{k+1})$ then $(s_k, s_{k+1}) \in rely$*

Assum represents the set of component computations $(\Gamma, cptn)$ such that the state component of the initial configuration of the computation is a `Normal` state satisfying *pre* and for any transition of the environment in the computation the pair of the states of the configurations in the transitions belongs to the rely relation.

To take advantage of automatic methods such as model checking, and following the original notion of validity for Hoare triples in Simpl, the commitment assumes that the last configuration in a computation does not end in a Fault state belonging to the set F, which is a set of non-reachable states previously calculated using external tools. Then the commitment is the set of computations such that component transitions belong to the *guarantee* relation, and that their last configuration are final (therefore with the program state equal to `Skip` or `Throw`) with the state component belonging to q or a.

Definition 5 (comm(guar,(q,a)) F). *The commitment of a pair of predicates (q,a), a relation guar, and a set of* `Fault` *states F, for an environment of procedures Γ is the set of component computations $(\Gamma, cptn)$ such that if cptn $l!(length\ l-1) = (l_p, l_s)$ and there is not a state f such that $l_s = $* `Fault` *$f$ and $f \in F$ then for any component transition in the computation $\Gamma \vdash_c (cptn)!k \rightarrow (cptn)!(Suc\ k)$ where $k < length\ cptn$, $cptn!k = (p_k, s_k)$, and $cptn!(Suc\ k) = (p_{k+1}, s_{k+1})$ then $(s_k, s_{k+1}) \in guar$, and if l is final then $l_s = $* `Normal` *$l'_s$ and if $l_p = $* `Skip` *then $l'_s \in q$ and if $l_p = $* `Throw` *then $l'_s \in a$*

Definition 6 (com_validity). *Validity of a specification of a component P w.r.t. a precondition p, postcondition (q,a), a rely relation R, a guarantee relation G, an environment of procedures Γ and a set F of Fault states, represented as $\Gamma \models_{/F} P\ sat[p,R,G,q,a]$ iff for all s, $cp\ \Gamma\ P\ s \cap assum(p,R) \subseteq comm(G(q,a))\ F$*

Following [12], we use a set of procedure specifications Θ that are used during the procedure verification. The procedure specifications Θ, is a tuple which terms represent a procedure name and its specification in terms of precondition, rely and guarantee relations, and postcondition. Note that procedures in specifications belonging to Θ do not need to match the procedures defined in the environment Γ.

Definition 7 (com_cvalidity). *CValidity of a specification of a component P w.r.t. a precondition p, postcondition (q,a), a rely relation R, a guarantee relation G, an environment of procedures Γ, a specification of procedures Θ, and a set F of Fault states, represented as $\Gamma, \Theta \models_{/F} P\ sat[p,R,G,q,a]$ iff if $\forall (c, p', R', G', q', a') \in \Theta.\ \Gamma \models_{/F} (Call\ c)\ sat[p', R', G', q', a']$ then $\Gamma \models_{/F} P\ sat[p,R,G,q,a]$*

The notion of validity for parallel computations is defined similarly using the definitions of computation, assumption, and commitment for parallel programs. We omit these definitions due to space reasons.

3.3 Inference Rules of the Proof System

The Rely-Guarantee proof system for CSimpl extends the previous mechanization of the logic in [9] with eight more inference rules. There are a total of fifteen rules, one for each language constructor, plus the consequence rule. Fig. 3 shows those rules that are either new or substantially changed w.r.t. the work in [9]. Rules `Skip`, and `Throw` are added to handle program termination for normal and abrupt termination respectively. Since `Skip` deals with normal termination it requires the normal postcondition to be stable w.r.t. the rely relation, whilst in the case of `Throw` is the abrupt postcondition which has to be stable w.r.t. the rely relation. Similarly, Catch is the complement of the sequential rule for abrupt termination. In CSimpl, composition of programs can finish on an abrupt state without executing the second program. Hence it is necessary stability of the abrupt postcondition w.r.t. the *rely* relation. Similarly, the `CATCH` rule requires stability of the normal postcondition with rely.

The Await rule requires Hoare satisfiability of the sequential program representing its body. Since the Hoare program can finish in either a normal state or an abrupt state, it is necessary that both postconditions are stable. The precondition should also

$$\frac{\begin{array}{l} Sta\ p\ R \quad Sta\ q\ R \quad p \subseteq \{s. f\ s \in q\} \\ \forall s\ t. s \in p \wedge (t = f\ s) \longrightarrow (Normal\ s, Normal\ s) \in G \end{array}}{\Gamma, \Theta \vdash_{/F}\ Basic\ f\ \mathbf{sat}\ [p,R,G,q,a]}\ \textsc{Basic}$$

$$\frac{\begin{array}{l} Sta\ p\ R \quad Sta\ q\ R \\ p \subseteq \{s.(\forall t.(s,t) \in r \longrightarrow t \in q) \wedge (\exists t.(s,t) \in r)\} \\ \forall s\ t. s \in p \wedge (s,t) \in r \longrightarrow (Normal\ s, Normal\ s) \in G \end{array}}{\Gamma, \Theta \vdash_{/F}\ Spec\ r\ \mathbf{sat}\ [p,R,G,q,a]}\ \textsc{Spec}$$

$$\frac{\begin{array}{l} Sta\ a\ R \\ \forall s.(Normal\ s, Normal\ s) \in G \end{array}}{\Gamma, \Theta \vdash_{/F}\ Throw\ \mathbf{sat}\ [a,R,G,q,a]}\ \textsc{Throw} \qquad \frac{\begin{array}{l} Sta\ q\ R \\ \forall s.(Normal\ s, Normal\ s) \in G \end{array}}{\Gamma, \Theta \vdash_{/F}\ Skip\ \mathbf{sat}\ [q,R,G,q,a]}\ \textsc{Skip}$$

$$\frac{\begin{array}{l} Sta\ p\ R \quad Sta\ q\ R \quad\quad Sta\ a\ R \\ \forall V. \Gamma_{\neg a}, \vdash_{/F} (p \cap b \cap \{V\}) c \\ \quad \{s.(Normal\ V, Normal\ s) \in G\} \cap q, \\ \quad \{s.(Normal\ V, Normal\ s) \in G\} \cap a \end{array}}{\Gamma, \Theta \vdash_{/F}\ Await\ b\ c\ \mathbf{sat}\ [p,R,G,q,a]}\ \textsc{Aw} \qquad \frac{\begin{array}{l} \Gamma, \Theta \vdash_{/F}\ c\ \mathbf{sat}\ [p \cap g,R,G,q,a] \\ Sta\ (p \cap g) R \quad \forall s.(Normal\ s, Normal\ s) \in G \end{array}}{\Gamma, \Theta \vdash_{/F}\ Guard\ f\ g\ c\ \mathbf{sat}\ [p \cap g,R,G,q,a]}\ \textsc{Gd}$$

$$\frac{\begin{array}{l} \Gamma, \Theta \vdash_{/F}\ c1\ \mathbf{sat}\ [p,R,G,q,r] \\ \Gamma, \Theta \vdash_{/F}\ c2\ \mathbf{sat}\ [r,R,G,q,a] \\ Sta\ (p \cap g) R \quad Sta\ (a \cap g) R \\ \forall s.(Normal\ s, Normal\ s) \in G \end{array}}{\Gamma, \Theta \vdash_{/F}\ Catch\ c1\ c2\ \mathbf{sat}\ [p,R,G,q,a]}\ \textsc{Catch} \qquad \frac{\begin{array}{l} \Gamma, \Theta \vdash_{/F}\ c1\ \mathbf{sat}\ [p,R,G,r,a] \\ \Gamma, \Theta \vdash_{/F}\ c2\ \mathbf{sat}\ [r,R,G,q,a] \\ Sta\ (p \cap g) R \quad Sta\ (a \cap g) R \\ \forall s.(Normal\ s, Normal\ s) \in G \end{array}}{\Gamma, \Theta \vdash_{/F}\ Seq\ c1\ c2\ \mathbf{sat}\ [p,R,G,q,a]}\ \textsc{Seq}$$

$$\frac{\begin{array}{l} \Gamma, \Theta \vdash_{/F}\ c\ \mathbf{sat}\ [p \cap g,R,G,q,a] \\ Sta\ (p \cap g) R \quad f \in F \\ \forall s.(Normal\ s, Normal\ s) \in G \end{array}}{\Gamma, \Theta \vdash_{/F}\ Guard\ f\ g\ c\ \mathbf{sat}\ [p,R,G,q,a]}\ \textsc{G} \qquad \frac{\begin{array}{l} \Gamma, \Theta \vdash_{/F}\ the(\Gamma\ c)\ \mathbf{sat}\ [p \cap g,R,G,q,a] \\ Sta\ (p \cap g) R \quad c \in dom\ \Gamma \\ \forall s.(Normal\ s, Normal\ s) \in G \end{array}}{\Gamma, \Theta \vdash_{/F}\ Call\ c\ \mathbf{sat}\ [p,R,G,q,a]}\ \textsc{C}$$

$$\frac{\begin{array}{l} \forall s \in p. \Gamma, \Theta \vdash_{/F}\ c\ s\ \mathbf{sat}\ [p \cap g,R,G,q,a] \\ Sta\ p\ R \quad \forall s.(Normal\ s, Normal\ s) \in G \end{array}}{\Gamma, \Theta \vdash_{/F}\ DynCom\ c\ \mathbf{sat}\ [p,R,G,q,a]}\ \textsc{DynCom}$$

$$\frac{\begin{array}{l} \forall i{<}xs. R \cup (\bigcup j \in \{j. j{<}xs \wedge j \neq i\}.(Guar(xs!j))) \subseteq Rely(xs!i) \\ (\bigcup j{<}length\ xs.(Guard(xs!j))) \subseteq G \quad p \subseteq (\bigcap i{<}length\ xs. Pre(xs!i)) \\ (\bigcap j{<}length\ xs.(Post(xs!j))) \subseteq q \quad (\bigcup j{<}length\ xs.(Abr(xs!j))) \subseteq a \\ \forall i{<}xs. \Gamma, \Theta \vdash_{/F}\ Com(xs!i)\ \mathbf{sat}\ [Pre(xs!i), Rely(xs!i), Guar(xs!i), Post(xs!i), Abr(xs!i)] \end{array}}{\Gamma, \Theta \vdash_{/F}\ xs\ \mathbf{SAT}\ [p,R,G,q,a]}\ \textsc{Comp}$$

Fig. 3. Rely-guarantee Proof Rules for CSimpl

to be stable to remain unchanged under environment transitions. Since every component transition has to belong to the guarantee relation, we add this constraint into the Hoare triple, binding the initial states from the precondition to the final states of both the normal and abrupt postconditions.

The rest of rules can be deduced intuitively from their semantics adding stability of the precondition for non-terminal commands, e.g., `if` for branching and `call` for non-recursive procedure calls; and adding also stability of the normal postcondition for commands modifying the state.

`Sta p R` defines stability of a predicate `p` with respect to a relation `R`. It states that if a state s satisfies `p`, then for any state s' such that $(s, s') \in$ `R`, s' must also satisfy `p`.

Finally the `COMP` is the rule for parallel composition. To apply compositionality, the rule is applied over a tuple *xs* composed of a sequential component *Com* and rely-guarantee specification, i.e. *Pre*, *Rely*, *Guarantee*, *Post* for *Com*. The rule follows [9] taking abrupt termination into consideration, since this is an exception state, and not all the individual computations may be in an exception state. Therefore, whilst we require that the intersection of all component postconditions is included in the postcondition of the parallel program, for abrupt termination we only require that the union of abrupt postconditions is in the parallel program.

3.4 Soundness of the Proof System

We prove that the set of inference rules in the proof system is sound w.r.t. CSimpl semantics. The proof is carried out in two steps, first we prove that the inference rules for single components are sound. Then we show that the compositional rule for parallel programs is also sound.

lemma `SeqRG_sound:`
"Γ,Θ $\vdash_{/F}$ `c sat [p, R, G, q,a]` $\Longrightarrow$ Γ,Θ $\models_{/F}$ `c sat [p, R, G, q,a]`"

This is proved by induction on the inference rules. Axioms, i.e., those rules without assumptions on the proof system induction, are proven based on the notion of stability and the fact that any computation starting from them only has one component step. Therefore we prove that the stability rule preserves the precondition under any environment step. We then show that the component step preserves the commitment.

The semantics for computation makes it cumbersome to prove the soundness for those CSimpl constructors whose semantic is recursively defined, such as `Seq`, `Catch`, and `While`. Soundness for these constructors are proven using a modular notion of computation [14] and the equivalence of both types of computation. The modular computation serializes the recursive specification of computation for the CSimpl constructors. This alternative semantics for computation unfolds the computation of CSimpl constructors. Soundness for these constructors is proven based on the different cases these rules provide. The modular computation for CSimpl extends the one provided in [9] with rules for the new language constructors, and new rules for `seq` and `while`, considering that the program in a final configurations can be `Skip` or `Throw`. The constructors `If` and `Call`, for non-recursive function calls, are proven similarly to the axioms based on the existence of a first component step for non-final configurations. After applying the component transition, we prove the correctness by the inductive step.

Recursive procedure calls require to consider the maximum number of nested function calls invoked by an execution and we do not currently provide a rule for them. `cptn` serializes the small step semantics, and it is not enough to prove soundness of recursive procedure calls. Nevertheless, it is possible to provide such a rule for recursive procedure calls, by extending with a parameter *n* the modular computation, which we call `cptn_mod_nest`. *n* represents the limit of nested procedures for which the computation is valid. Also, the semantics for validity must be extended to express that a formula is valid when it invokes at least *n* nested function calls by intersecting the assumptions in `com_validity` with the set of modular computations with limit *n*. Soundness of recursive procedure calls can be proven similarly to [12], supported by the equality $(\Gamma, l) \in cptn = \exists n.(n, \Gamma, l) \in cptn\_mod\_nest$ and monotonicity of *cptn_mod_nest* in *n*.

Finally we prove the soundness for the parallel composition of programs. The proof for Theorem 2 is immediate after proving Lemma 1, which is proven using Theorem 1 for compositionality of CSimpl semantics and validity definition.

Lemma 1 (parallel_sound).

$$\begin{array}{l}\forall \texttt{i<len xs.}\\ \quad \texttt{R}_p \cup (\bigcup \texttt{j}\in\{\texttt{j. j < len xs} \wedge \texttt{j} \neq \texttt{i}\}\texttt{. (G (xs ! j)))} \subseteq \texttt{(R (xs ! i))} \Longrightarrow\\ (\bigcup \texttt{j<len xs. (G (xs ! j)))} \subseteq \texttt{G}_p \Longrightarrow \texttt{p} \subseteq (\bigcap \texttt{i<len xs. (P (xs ! i)))} \Longrightarrow\\ (\bigcap \texttt{i<len xs. (Q (xs ! i)))} \subseteq \texttt{q} \Longrightarrow (\bigcup \texttt{i<len xs. (A (xs ! i)))} \subseteq \texttt{a} \Longrightarrow\\ \forall \texttt{i<len xs. } \Gamma\texttt{,}\Theta \models_{/\texttt{F}} \texttt{C (xs !i) sat [P (xs !i), R (xs ! i), G (xs ! i),}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{Q (xs ! i),A (xs ! i)]} \Longrightarrow\\ \Gamma\texttt{,}\Theta \models_{/\texttt{F}} \texttt{ParCom xs SAT [p, R}_p\texttt{, G}_p\texttt{, q,a]}\end{array}$$

Theorem 2 (par_rgsound).

$$\Gamma\texttt{,}\Theta \vdash_{/\texttt{F}} \texttt{Ps SAT [p, R, G, q,a]} \Longrightarrow \Gamma\texttt{,}\Theta \models_{/\texttt{F}} \texttt{(ParCom Ps) SAT [p, R, G, q,a]}$$

4 Case Study

We apply the proof system for the specification and the verification of two XtratuM services for inter-partition communication. The XtratuM separation micro-kernel [3] provides spatial and temporal partitioning of applications. In a separation micro-kernel, different partitions are executed in separated memory domains, and the only allowed communication among partitions is by means of static dedicated channels explicitly defined between two or more partitions. XtratuM provides services to manage partitions, communicate partitions through inter-partition channels, partitions health-monitoring, and a static cyclic scheduler. In this case study we provide a very abstract CSimpl specification of the services to send and receive messages using queuing channels in a parallel architecture, where the XtratuM micro-kernel is executed in several cores of a multi-core processor. Using the Rely-Guarantee proof system introduced in section 3 we prove: (1) that the services correctly introduce and remove elements in the queues associated with each communication channel and (2) that the number of elements in the queues do not exceed the channel maximum capacity. The specification and proofs are comprised of more than 3500 lines of specification.

4.1 Queuing Inter-partition Communication Description

Queuing inter-partition communication services let partitions to escape from the isolated environment that XtratuM provides, allowing them to send and receive messages to/from other partitions using communication channels by means of dedicated ports assigned to partitions. A communication channel is an entity storing the communication data and the source and destination ports involved in the communication.

XtratuM implements two types of communication: sampling and queuing communication. While the former only allows to store one message, and it is multicasting, i.e., a channel has one source port and a list of destination ports. The latter allows to store many messages in a bounded buffer implemented as a queue, and only allows peer to peer communication, i.e., the channel has one source port and one destination port. Channels and ports are classified according to the type of communication. Therefore, a channel and a port can be of type sampling or queuing, and a sampling channel can only allocate sampling ports, and vice-verse. The services have as input a port to/from which the message is sent/received, and the message to be send in the case of the sending service. Prior to modifying the queue, the services check whether the input values are consistent, e.g., the port which receives the message belongs to the partition, or it is a source or destination port depending on the invoked service. In the present case study we will focus on queuing communication.

4.2 State and Specification Definition

The state definition provides global and local variables. Global variables represent those variables shared by multiple instances of the micro-kernel, they hold the data for inter-partition communication, partitions, and the partition scheduler. Since we are targeting only queuing inter-partition communication, the components for the scheduler and partitions contain the necessary information for those services. The scheduler is highly abstracted and only contains information about the partition that is currently being executed, and therefore invoking the service; partitions only contain the list of assigned ports to the partition. The communication datatype includes the specification of channels and ports. A channel is defined as a datatype with the two possible types of channels, having as parameters the source and destination ports, and the message shared between the partitions, for which the queue is abstracted in the model as a multiset. The queuing channel also has as parameter the maximum size of the channel buffer. Messages are modelled just as an abstract entity.

```
datatype port = Port  port_id port_name  port_direction
datatype channel =
  Sampling_C chan_id port_id "port_id set" "Message option"
 |Queuing_C chan_id port_id port_id max_buffer_size "Message multiset"
record com = ports :: "port_id ⇀ port" channels :: "chan_id ⇀ channel"
record vars = p_' :: "part_id ⇀ partition"   c_' :: "com"
            s_' :: "scheduler list"   l_' ::"locals list"
```

In the model, ´l, ´c, ´s, and ´p access the locals, communication, scheduler, and partition component of the state, respectively. Local variables for each process are a structure with the necessary variables for the input and output parameters of the services. One of the limitations of Rely-Guarantee is that the relations lose track of the sequence of executed operations. To solve this, verification of the concurrent increment of a variable, or adding/removing elements from a set like in this example, requires using additional variables that help to track the changes [14]. To support the verification, local variables of processes include an auxiliary variable of type `Message option` that is initialized to None, and when a message is correctly sent or received the model assigns it to the auxiliary variable. Our state abstracts and map XtratuM global structures `xmcCommChannelTab`, and `xmcCommChannelPorts` into the components of `´c` and `xmcPartition` into `´p` respectively.

The parallel execution of services is modelled parametrically on the number of processes, which is defined as a fixed natural number within a Isabelle/HOL locale [7]. Each service is modelled as a procedure that is also parametrized by the process being executed; this allows that each specific procedure only accesses its local variables. The function Γ is generated by assigning a unique name for each service using a fold higher order function and assigning to each parametrized service name the corresponding parametrized body of the service. The parametrized event *receive* is shown below.

```
definition receive_q_message_i where "receive_q_message_i i ≡
(IF (¬ (ex_port_id ´c ((pt ((´l)!i))))) ∨
    (¬ (port_q (the ((ports ´c) (pt (´l!i)))))) ∨
    (¬ (port_dest (the ((ports ´c) (pt (´l!i)))))) ∨
    (¬ port_in_part  ´p ((´s)!i) (the ((ports ´c) (pt (´l!i))))) THEN
      ´l :== ´l[i:=((´l!i)(|ret_msg := None|))]
 ELSE AWAIT True
   IFs port_empty (pt ((´l)!i)) ´c  THEN
    ´l :==s ´l[i:=((´l!i)(|ret_msg := None|))]
   ELSE
    ´l :==s ´l[i:=((´l!i) (|ret_msg := port_get_msg (pt (´l!i))´c |))] ;;s
    ´c :==s  port_rem_msg  (pt (´l!i))  (the (ret_msg (´l!i))) ´c ;;s
          ´l :==s ´l[i:=((´l!i) (|aux_msg := (ret_msg (´l!i))  |))]
   FI FI)
```

The events abstract the low level behaviour of the Xtratum functions into three stages: parameters checking, lock of mutex, and insertion/extraction to/from the queue. Validation of correctness of the model w.r.t. the implementation is carried out at this stage by inspecting the code. For the `ReceiveQueuingPort` model, the event first checks that the accessed port exists in the current communication state, that it is a queuing and destination port, and that the partition that it belongs to the partition being executed. If any parameter is not valid then the service finishes returning `None`. If the parameters are correct then it performs the operations over the channel queue after checking whether the queue is not empty for event *receive*, or not full for event *send*. The statements in the body of the Await statement are $\texttt{IF}_s$ and $\texttt{:==}_s$. This is because the body of the `Await` is a sequential Simpl program, and it is necessary to provide a different syntax sugar than that used for CSimpl statements. Accessing shared variables, i.e., checking the queue size and modifying the queue of the input port, is done using an atomic block to ensure mutual exclusion. The other event is modeled similarly.

4.3 Verification

For the parallel verification it is first necessary to specify the rely and guarantee relations for the receive and send services. We show the rely relation, the guarantee relation is similar to this, only differing in that local variables for any process *j* different than *i* will not be modified.

```
definition Rely where
"Rely_Send_Receive B i≡  {(x,y). (∃x1 y1.
   x=Normal x1 ∧ y=Normal y1 ∧ (l_' x1)!i = (l_' y1)!i ∧ s_' x1 = s_' y1 ∧
   ports (c_' x1) =  ports (c_' y1) ∧ p_' x1 = p_' y1 ∧
   (x1 ∈ Invariant B ⟶ y1 ∈ Invariant B))  } "
```

Since parallel programs do not change others programs' local variables, the *i* element in the list of local variables is not changed by the rely. Also, ports, partitions, and the scheduler are not changed by any service, therefore the rely relation does not change them. Finally, if the initial state of the relation preserves the invariant, it also preserves the shape of the channel's queues, then so does the final state of the relation.

The invariant establishes consistency of the port and channel structures that must be preserved by the services. Its most important specification is `channel_spec` which preserves the specification of the queue for every defined channel in the state.

```
definition channel_spec  where "channel_spec B ≡
  ⦃ ∀c_id c. (channels ´c) c_id = Some c ⟶
    chan_get_msgs c = (B c_id + chan_sent_msgs c_id ´c ´l) -
      chan_rec_msgs c_id ´c ´l ∧
    (size (chan_get_msgs c) ≤ chan_get_max_bufs c) ∧
    chan_rec_mes c_id ´c ´l) ⊆# B c_id + chan_sent_msgs c_id ´c ´l ⦄"
```

`channel_spec` checks that the multiset modelling the queue for each defined *ch_id* is equal to its initial value, which is given by `B c_id`; those messages correctly sent are pushed into the queue by the service; and that the received messages are popped out of the queue. `chan_sent\rec_msgs` gets for each *ch_id* the multiset with the auxiliary variables different than `None`, meaning that the service has modified the queue for that channel. Also, for consistency of the multiset, the invariant needs to ensure that removed messages are a subset of the added messages.

Lemma 2 (Send_Rec_Correct).

```
n>0 ⟹ Γ,{} ⊢/{} (COBEGIN SCHEME [0 ≤ i < n]
  (ex_service i, pre_i B i, Rely B i, Guar B i, Post_Arinc B, ⦃ True ⦄)
  COEND) SAT [Pre_Arinc B, {(x,y). (∃x1 y1. x=Normal x1 ∧ y=Normal y1 ∧
              x1 = y1)}, ⦃ True ⦄, Post_Arinc B, ⦃ True ⦄]
```

`ex_service` is a sequence of nested *ifs* controlling the call to the services, each *if* guarded by a local variable that indicates which service is invoked in each parallel process. In the parallel program, the rely relation indicates that the parallel environment does not change the state, being therefore a closed system, i.e., there is not any environment at the parallel level. The guarantee relation is the universal set in which everything can be modified. The precondition `Pre_Arinc B` defines the invariant and auxiliary variables initialization to `None`. The precondition for each process `pre_i B i` sets the initial value for the auxiliary variable, the initial values of the channel queues, and it defines the invariant that is preserved by the postcondition for the normal termination Post_Arinc B. The abrupt postcondition is the universal set since we do not have any abrupt termination in this specification. The postcondition is the same for the parallel specification and for the components.

The proof obligations for the parallel rule are proved immediately after unfolding the definitions of the precondition, postcondition, and rely and guarantee relations. After applying the parallel rule on the parallel execution of the *n* components, it is necessary to prove that the parametrized `execute_service` satisfies the postcondition using the rely-guarantee rules for components. Once the `conditional` and `call` rules have been applied on `execute_service`, only the proof of the verification of each service body is left. Both send and receive services are similarly proven.

To prove the body of the services, it is necessary to apply the conditional rule to generate the proof obligations for the execution of two branches of the `if`. The first corresponds to the case in which the service is not invoked with the appropriate parameters and is immediately proven after apply the `Basic` rule since it does not modify any channel or auxiliary variable. For the second branch, after invoking `Await`, the sequential Simpl program representing its body is automatically unfolded using Simpl's VCG. The resulting goal, now without any embedded Simpl specification, is solved by proving that the state after removing or inserting a message from/to the channel associated to the input port, and after assigning the removed/inserted message to the auxiliary variable, satisfies `channel_spec`. We use some auxiliary lemmas to prove it: first, that the modification of the auxiliary variable in a component does not modify the sets `chan_sent_msgs` and `chan_rec_msgs` for any channel other than the one associated to the port the service access; second, that the modification of a variable only modifies one of these sets. Using these auxiliary lemmas the postcondition is proven immediately by applying the properties over multisets.

5 Conclusions and Future Work

In this work we have presented CSimpl, a framework for specifying concurrent programs and verifying their partial correctness using Rely-Guarantee. This framework allows us to specify programs written in a large subset of the C language. Currently we are working also on axiomatic separation rules for the proof system following works on separation logic and rely guarantee [13, 5]. This will help to cope with local variables and to hide global variables, thus improving scalability of the approach. There are, however, some aspects where this framework can be improved. First, we can introduce deadlock freedom and weak total correctness, which enable us to reason about termination of programs. Second, we can provide VCG tactics to achieve a higher level of automation. Currently, the language supports annotation to provide loop invariant, but the soundness of annotated rules is yet to be proven. Third, it is also desirable to have completeness of the proof system to introduce properties proven at the language and semantics levels. The complexity of proving completeness make us to consider this as future work. Finally, the current proof system do not include a rule for recursive procedure calls, but our framework can be easily extended to support it, with minimal modifications on the rules already proven.

References

1. Armstrong, A., Gomes, V.B.F., Struth, G.: Algebraic principles for rely-guarantee style concurrency verification tools. In: Proceedings of 19th International Symposium on Formal Methods (FM). pp. 78–93 (2014)
2. de la Cámara, P., del Mar Gallardo, M., Merino, P.: Model extraction for arinc 653 based avionics software. In: Proceedings of 14th International SPIN Workshop on Model Checking Software. pp. 243–262 (2007)
3. Carrascosa, E., Coronel, J., Masmano, M., Balbastre, P., Crespo, A.: Xtratum hypervisor redesign for leon4 multicore processor. SIGBED Rev. 11(2), 27–31 (Sep 2014)
4. Coleman, J.W., Jones, C.B.: A structural proof of the soundness of rely/guarantee rules. Journal of Logic and Computation 17(4), 807–841 (Aug 2007)
5. Feng, X.: Local rely-guarantee reasoning. SIGPLAN Not. 44(1), 315–327 (Jan 2009)
6. Jones, C.B.: Development Methods for Computer Programs including a Notion of Interference. Ph.D. thesis, Oxford University (Jun 1981)
7. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales: A sectioning concept for isabelle. In: Proceedings of 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs). pp. 149–165. Springer (1999)
8. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an os kernel. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP). pp. 207–220. ACM, New York, NY, USA (2009)
9. Nieto, L.P.: The rely-guarantee method in Isabelle/HOL. In: Proceedings of the 12th European Conference on Programming (ESOP). pp. 348–362. Springer-Verlag (2003)
10. Nipkow, T., Nieto, L.P.: Owicki/gries in isabelle/hol. In: Proceedings of Second International Conference on Fundamental Approaches to Software Engineering (FASE). pp. 188–203 (1999)
11. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs i. Acta Informatica 6(4), 319–340 (1976)
12. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. Ph.D. thesis, Technischen Universitat Munchen (2006)
13. Vafeiadis, V., Parkinson, M.: A Marriage of Rely/Guarantee and Separation Logic, pp. 256–271. Springer Berlin Heidelberg (2007)
14. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects of Computing 9(2), 149–174 (1997)
15. Zhao, Y., Yang, Z., Sanán, D., Liu, Y.: Event-based formalization of safety-critical operating system standards: An experience report on arinc 653 using event-b. In: Proceedings of IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). pp. 281–292 (Nov 2015)
16. Zhao, Y., Sanán, D., Zhang, F., Liu, Y.: Reasoning about information flow security of separation kernels with channel-based communication. In: Proceedings of 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 791–810 (2016)