Verification of Configurable SRA Systems

Alessandro Cimatti[1], Alberto Griggio[1], Christian Lidström[1], Gianluca Redondi[1], and Dylan Trenti[1,2]

[1] Fondazione Bruno Kessler, Trento, Italy
{cimatti,griggio,clidstrom,gredondi,dtrenti}@fbk.eu
[2] University of Udine, Italy trenti.dylan@spes.uniud.it

Abstract. Many digital systems are designed as collections of asynchronous processes orchestrated by a domain-specific scheduler. The verification of such *scheduler-restricted asynchronous* systems (SRA) is challenging due to process-process and process-scheduler interactions. In this paper, we tackle the problem of verifying *configurable* SRA. A configurable SRA describes an unbounded family of possible SRA, each resulting from an instantiation satisfying given configuration constraints; our goal is proving at once that every legal instantiation of a configurable SRA is correct. We propose a contract-based, deductive verification approach that combines (i) compositional proof rules that abstract the scheduler to prove top-level invariant properties, (ii) automatic summarizations of the methods invoked by the scheduler, (iii) simplification with respect to the nature of the space of configurations. The approach is grounded in (object-oriented) first order logic, requires reasoning over quantified statements, and leverages the Dafny software verifier as a backend. An experimental evaluation on industrial case studies demonstrates that the framework scales effectively and enables practical reasoning about complex parameterized behaviors.

1 Introduction

Many modern digital systems are designed as the combination of interacting processes coordinated by a domain-specific scheduler to implement a cyclic execution pattern. Within this design pattern, which we refer to as Scheduler-Restricted Asynchronous systems (SRA), the scheduler orchestrates process execution in discrete phases, without preemptions, so that the traditional interleaving model of concurrency is restricted to a result in a structured duty cycle. The SRA paradigm is adopted in various domains, including embedded control and systems on chip, with the prominent example of SystemC [16], and has been the subject of significant research efforts in formal verification [12, 7, 18, 28, 21].

In this work, we focus on *configurable* SRA systems, that are compact, parameterized descriptions of unbounded families of SRA. A configurable SRA $\mathcal{S}$ defines a set of concrete SRA; for each configuration $\mathcal{C}$ satisfying given configuration constraints Γ, there is a corresponding SRA $\mathcal{S}[\mathcal{C}]$ resulting from a (legal) instantiation of the parameters with the object and interconnections specified by $\mathcal{C}$.

Configurable design is widely adopted, for example in software product lines, firmware for heat pump control or generic railways control procedures, for its ability to factor out the commonalities of the possible products. The W-development process [23] is composed of a first phase, where the general traits of the domain are designed and analyzed, and several application-specific phases, where the characteristics of each product are chosen.

Our goal is to perform *parameterized verification*, proving that a given generic, quantified specification φ holds for all possible instantiations, i.e. for all $\mathcal{C}$ such that $\mathcal{C} \models \Gamma$, $\mathcal{S}[\mathcal{C}] \models \varphi$. This contrasts with the simpler task of verifying each concrete SRA $\mathcal{S}[\mathcal{C}]$ separately, where the processes and their interconnections are fixed. The advantage of parameterized verification is that the effort is carried out, at the configurable SRA level, earlier in the development process, so that the generic application can be certified once and for all. The verification of each concrete SRA $\mathcal{S}[\mathcal{C}]$ is limited to checking that $\mathcal{C} \models \Gamma$.

We propose a deductive verification approach for the parameterized verification of configurable SRA systems. We adopt an object-oriented modeling language. Classes represent process types and are equipped with a state machine, whose transitions are associated with guards and effects described as methods in an imperative-style language. Distinguishing features include the ability to represent configurations, to quantify over unbounded collections and domain-specific types, and to define a generic scheduling policy and generic properties.

Our main technical contributions are a contract-based, deductive approach that combines the following ingredients: (i) in order to prove top-level invariant properties, a set of compositional proof rules are used to abstract the details of the scheduling policy and decompose global properties into per-class obligations; (ii) an automated summarization of the methods invoked by the scheduler, which supports an abstraction from the implementation details; and (iii) model simplification with respect to the nature of the space of configurations, in order to reduce the complexity of the verification conditions.

The verification framework was implemented on top of Dafny [22], whose language is very suitable to express the features of configurable SRA systems. We automatically generate both the Dafny models as well as the contracts for the process implementations. We then apply our compositional strategy to prove that a given specification φ holds for all the possible valid configurations using the automatically-generated implementation contracts and a user-supplied (quantified) invariant, by discharging a sequence of proof obligations using the Dafny prover.

This paper generalizes our previous case-study works [9, 10] to a domain-independent framework for configurable SRA systems. In particular, we introduce a set-based formalization, automatic generation of implementation contracts, and a general compositional verification strategy.

We carried out an experimental evaluation over benchmarks from industrial railway control systems and embedded system control. The analyzed models comprise tens of thousands of lines of code and automatically generated contracts. All of these were automatically proved by Dafny to hold on the corre-

sponding implementations. We also demonstrate the impact of the simplifications driven by the configuration constraints, that were proved correct, and we established multiple system level safety properties for *all possible configurations*. The results demonstrate that the approach is able to benefit from the expressiveness of deductive verification without having to deal with manual annotations sacrificing automation and scalability.

Outline. Section 2 reviews the related work. Section 3 introduces the syntax and semantics of configurable SRA systems. Section 4 formalizes the problem and presents the compositional verification strategy. Section 5 describes the tool chain, and Section 6 presents the experimental evaluation. Section 7 concludes with future work. Additional material (example, contract-generation algorithm, and detailed experimental data) is available in the appendix.

2 Related Work

Software Product Lines (SPLs) study the design of families of products, each defined by a selection of features [6]. The SPL verification problem amounts to proving the correctness of (every product in) the family [29, 27, 19]. Differently from our work, SPLs typically focus on models of software systems, not on processes orchestrated by a scheduler. More importantly, feature models in SPLs are typically restricted to finite sets (and not unbounded families) of products.

Various works deal with the verification of SystemC [28, 14, 7], a well known design language for SRA systems. Tasche et al. [28] propose a deductive verification framwork where VerCors [5] is used as a backend to verify SystemC programs. The works in [14, 7] present fully algorithmic approaches based on explicit-state model checking and on the ESST extension with software model checking techniques [13]. The work in [12] discusses the verification of railways interlockings as SRA systems with a domain-specific scheduler. All these works focus on verifying a single SRA, composed by a finite number of statically arranged threads, rather than an unbounded family of systems.

Also related are the deductive verification frameworks for parameterized systems such as Ivy [25, 1, 26, 30]. Their models are directly expressed as logical formulae with uninterpreted functions and quantifiers. This is a crucial difference, as our framework allows the users to write *programs* in a control-logic language, with native notions of execution cycle and synchronization primitives. This design choice is important for industrial adoption, where engineers are more comfortable writing code than quantified first-order specifications. We also automatically extract declarative specifications in the form of contracts from the code, similarly to [17].

Cutoff results [4] establish that the verification of unbounded families can be reduced to checking a finite number of instances. These results typically are limited to fixed topologies with strong structural constraints. In contrast, our framework can deal with system topologies described by a set of configuration constraints, that provide greater modeling flexibility.

Somewhat related is the IronFleet framework [20], that leverages Dafny for mechanically verifying the safety and liveness of practical distributed system implementations. IronFleet starts with a global abstract specification, with a proof strategy based on refinement between state machines. Our work deals with a collection of component programs, and relies on compositional contract-based reasoning. Finally, IronFleet's case studies are primarily distributed systems, whereas we focus on SRA systems.

Our previous papers [9, 10] are direct predecessors of this work. They focus on specific railway systems and toolchain instances, while in this paper we present a general framework with improved methodology. Section 6 quantifies the impact of these methodological changes.

3 Configurable SRA Systems

Informally, a configurable Scheduler-Restricted Asynchronous (SRA) system consists of a set of class declarations together with a scheduler description that coordinates instances' executions. Classes serve as templates for generating processes, expressed as extended finite state machines (EFSMs). Each class defines local behavior through transitions, consisting of a guard (a boolean expression), an effect (a program statement), and a specific *scheduling phase* in which the transition can be executed. The scheduler is also defined as an EFSM that coordinates the execution of class instances, as depicted in Figure 1, and whose locations correspond to scheduling phases. The definitions of both the process classes and the scheduler are *parameterized* in terms of (unbounded) sets of related objects. An instantiation of a system is then given by a *configuration*, specifying the concrete set of objects that are controlled by the scheduler and their mutual connections.

Scheduler transitions are divided into self-loop transitions (from phase p to itself), in which all process instances execute one of their transitions marked with p (if any), and phase change transitions, in which the internal state of the scheduler is updated, but no process transition is executed. A sequence of scheduler transitions starting from a designated *initial phase* l_0 and ending in a designated *final phase* l_f then forms a *scheduling cycle*. A *run* of the system consists of a sequence of scheduling cycles.

3.1 Syntax

Type System The language provides basic types (`Int`, `Bool`), finite enumeration sorts, user-defined class types C, and immutable set types $\texttt{Set}\langle C\rangle$. Two additional special sorts model timing and events: `Timer` and `Event`; timers count the number of global execution cycles, while events are special boolean fields. We also define a special enumeration type `PhaseEnum` that represents the scheduler's locations (phases), with distinguished initial and final values.

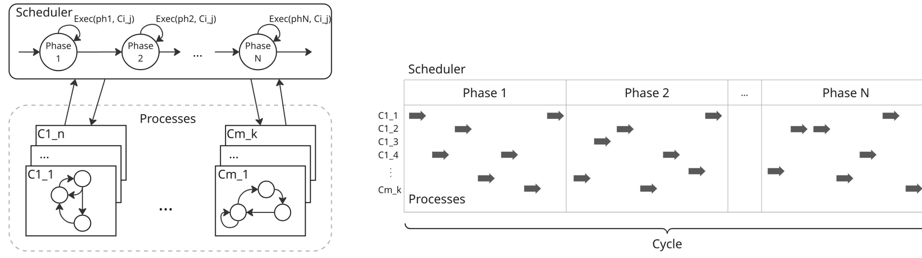

Fig. 1: SRA: architectural view (left) and dynamic view (right).

Types

$\sigma ::= \texttt{Int} \mid \texttt{Bool} \mid \mathit{Enum}$
$\tau ::= \sigma \mid C \mid \texttt{Timer} \mid \texttt{Event} \mid \texttt{Set}\langle C\rangle$

Expressions

$$
\begin{aligned}
e &::= c \mid x \mid x.f \mid e_1 \pm e_2 \mid e_1 * e_2 \\
&\mid\ \textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2 \mid |e| \\
b &::= \texttt{true} \mid \texttt{false} \mid e_1 = e_2 \mid e_1 < e_2 \\
&\mid\ \neg b \mid b_1 \wedge b_2 \mid x \in e \mid e_1 \subseteq e_2 \mid e_1 \,!!\, e_2 \\
&\mid\ e_1 \cup e_2 \mid (\forall x :: x \in e.\ b) \mid (\exists x :: x \in e.\ b)
\end{aligned}
$$

Statements

$$
\begin{aligned}
S &::= x := e; \mid x := *; \mid \textbf{if } b \textbf{ then } S \textbf{ else } S \\
&\mid\ \forall x :: x \in E\ \{x.f := e;\} \\
&\mid\ \texttt{assume } b; \mid \texttt{assert } b; \mid S; S \mid \varepsilon
\end{aligned}
$$

Fig. 2: Types, expressions and statements.

Classes A *class* C is defined by a class declaration that specifies the structure and behavior of a finite state machine template. Each class declaration consists of variable declarations, parameter declarations and transition declarations. Variable declarations ($\texttt{var}\ x : \sigma$) introduce mutable scalar fields, some of which can be tagged as `input`. Moreover, each class has a special `location` variable of a finite enumeration type (`LocationEnumC` for class type C) representing the locations of the EFSM. Parameter declarations ($\texttt{param}\ p : \sigma$) introduce immutable constants, and set declarations ($\texttt{set}\ s : \texttt{Set}\langle D\rangle$) introduce immutable collections of objects, where D is any user-defined class type. The *class signature* Σ_C of a class C is the set of symbols introduced by its declaration. Each class also declares a finite set of *transitions*. A transition specifies a guard-effect pair that defines when and how state changes occur, and is given by a tuple $(l_{start}, G, l_{end}, \mathcal{E}, p)$, where l_{start} and l_{end} are class locations, G is a boolean expression (the guard), $\mathcal{E}$ is a statement (the effect), and p is a phase enumeration value indicating during which scheduler phase the transition is eligible for execution. Every class contains three mandatory methods: an `init` method for setting initial variable values, an `exec` method for local execution (Section 3.2), and a `tick` method for updating timers.

Expression and statement language. Guards and effects in transitions are specified with expressions and statements in the language defined in Figure 2. Basic expressions include constants (c), variables (x), field access ($x.f$), and set car-

dinality ($|e|$). Boolean expressions extend standard logical operators with set membership ($x \in e$), subset relations ($e_1 \subseteq e_2$), disjointness assertions ($e_1 \,!!\, e_2$), and constrained quantification where $\forall x :: x \in e.\ b$ and $\exists x :: x \in e.\ b$ require e to have set type. We omit the description of the full type rules for brevity. Quantified assignments $\forall x :: x \in E\ \{x.f := e;\}$ are restricted to simple field assignments: E must be of set type, f must be a mutable scalar field of class C, e is an expression that may reference x (i.e., no nested quantified assignments, method invocations, or other statements are allowed).

The language enforces several restrictions. The distinguished variable `location` cannot be assigned in transition effects: location changes occur implicitly when transitions execute. `Event` variables can only be assigned to `true` in effects; they are reset to `false` automatically when consumed by transitions. Moreover, events are restricted syntactically: a guard is a conjunction of an event-free expression and a list of event variables. Variables declared as `input` are externally controlled and cannot be assigned in effects.

Scheduler Signature and Transitions The scheduler signature Σ_{Sched} extends the union of all class signatures with additional symbols of the form $\texttt{All}_{C_i}$ of type $\texttt{Set}\langle C_i \rangle$ interpreted as the set of instances of C_i in the configuration, a variable `phase` of type `PhaseEnum` for the scheduler's current location, and a boolean variable `executed` for each class instance.

Scheduler transitions define the global execution steps of the system. Each transition is specified by a guard, a start location p, and a target location p'. The guard is a Σ_{Sched}-expression, referencing only events, timers, and the `executed` flags of instances.

Configuration constraints describe structural restrictions on admissible configurations (e.g., disjointness relations or cardinality bounds). They are formulas over Σ_{Sched} that mention only immutable symbols (set-valued fields and constant parameters).

Definition 1 (Configurable SRA System). *A configurable Scheduler-Restricted Asynchronous system $\mathcal{S} = (\{C_1, \ldots, C_k\}, Sched, \Gamma)$ consists of a finite set of class declarations $\{C_1, \ldots, C_k\}$, a scheduler description Sched, and a set of configuration constraints Γ over the immutable symbols.*

3.2 Operational Semantics

Definition 2 (Configuration). *A* configuration $\mathcal{C}$ *for a finite set of classes $\{C_1, \ldots, C_k\}$ consists of: (i) a finite* universe $\mathcal{U}_i = \{o_{i,1}, \ldots, o_{i,n_i}\}$ *of object instances for each class C_i (domain for sort C_i); and (ii) an interpretation for all set-valued fields of C_i (interpretation for set predicates) and for all parameter fields of C_i (constant values) for each instance $o \in \mathcal{U}_i$.*

A configuration is the equivalent of a first-order structure, providing universes for each class and interpretations for set-valued fields and parameter fields (the non-mutable part of the system).

Local Semantics The local semantics define how individual class instances behave within a given configuration. The local execution follows a *small-step* semantics: each invocation of the `exec` method on a single instance constitutes one atomic step. The `exec` method takes a phase parameter that determines which category of transitions to consider, implementing the local execution through the following algorithm:

1. Filter all class transitions to include only those matching the specified phase;
2. Select the first transition whose guard is enabled in the current local state, according to the order in which transitions were declared in the class;
3. If an enabled transition is found: execute its effect statement sequence, update the control location to the target location, and reset to `false` every event variable that the occured in the transition's guard;
4. If no enabled transition exists: perform a *stutter step* where the local state remains unchanged.

Global Execution Semantics The global execution model coordinates class instances through the *scheduler*, following a *big-step* semantics.

Definition 3 (Global State). *Given a configuration $\mathcal{C}$ with universe $\mathcal{U}$, a* global state *S is a function that assigns: (i) for each instance $o \in \mathcal{U}$, an interpretation for all scalar fields and set-valued fields of o; and (ii) a location $S(\texttt{phase}) : \texttt{PhaseEnum}$, and a boolean value $S(\texttt{executed})$ for each instance.*

The *initial global state of the system* is established by initializing all instances and setting the scheduler to its initial location: each instance o's `init` method sets its scalar fields to their declared initial values, timers to `inactive`, events to `false`, and `executed` to `false`.

Global Execution. The effect of a scheduler transition from phase p to p' depends on its type:

– **Self-loop transitions** ($p = p'$): The scheduler invokes the `exec` method of each instance in an arbitrary order, passing p as the argument, and sets the `executed` flag of each instance to `true`. Note that local execution of an instance may read and modify both its own fields and fields of other instances; therefore, the order in which instances are executed can affect the resulting post-state.
– **Non-self-loop transitions** ($p \neq p'$): The scheduler updates `phase` to p' and resets `executed` to `false` for all instances. If the target is the final location, the scheduler additionally invokes each instance's `tick()` method to update timers.

We denote a scheduler transition from global state S to S' as $S \xrightarrow{p \to p'} S'$. A visual representation of the global execution is given in Figure 1, with the scheduler top coordinating the local execution of class instances within a cycle.

Definition 4 (Scheduling Cycle). *A* scheduling cycle *is a sequence of global states $S_0, S_1, \ldots, S_n$ such that:*

- $S_0(\mathtt{phase}) = l_0$ *(the cycle starts at the initial location)*
- *For each $i < n$, we have $S_i \xrightarrow{p_i \to p_{i+1}} S_{i+1}$ for some scheduler transition*
- $S_n(\mathtt{phase}) = l_f$ *(the cycle ends at the final location)*

After completing a cycle (reaching l_f), the scheduler transitions back to l_0 (setting `phase` accordingly) to begin the next cycle; in such transitions, input variables of all instances can be externally updated, while all other fields remain unchanged. A *run* of the system consists of a sequence of scheduling cycles, starting with the initial global state. An example of an SRA system is provided in Appendix A.

4 Compositional Verification of Configurable SRA

We start by formalizing the verification problem for configurable SRA systems and then present a compositional verification strategy to solve it. A single configurable SRA system describes an unbounded family of possible SRA, each resulting from an instantiation satisfying its configuration constraints. Our goal is proving at once that for every legal instantiation, at the end of every scheduling cycle, a given global property holds.

4.1 Verification Problem

Global properties φ are safety assertions over the global state expressed as boolean Σ_{Sched}-expressions. They may refer to both immutable and mutable instance fields to capture invariants such as mutual exclusion or consistency requirements.

Definition 5 (Parameterized Verification Problem). *Given a configurable SRA system $\mathcal{S}$ (with constraints Γ) and a global property φ, the* parameterized safety verification problem *is to prove that, for all configurations $\mathcal{C}$ that satisfy all elements in Γ, for each state S in a run of the system such that $S(\mathtt{phase}) = l_f$, the property holds: $\mathcal{C}, S \models \varphi$.*

In this paper we focus on properties checked at the end of scheduling cycles (i.e., when $\mathtt{phase} = l_f$), but the same compositional obligations can be adapted to properties required at other phases or over all reachable states.

4.2 Compositional Verification

Our deductive verification strategy decomposes global reasoning about the scheduler into a collection of local method contracts, combined through first-order entailment checks. Rather than modeling the scheduler explicitly, we reason compositionally: each class method is verified in isolation, while global correctness is obtained by proving that these local effects preserve a global invariant *Inv*.

Local Contracts A *local contract* for a method m of class C is a Hoare triple $\{Pre\}\ m\ \{Post\}$ where Pre and $Post$ are Σ_C-expressions. Moreover, $Post$ may reference pre-state values via the `old` operator: for any expression e, $\texttt{old}(e)$ denotes the value of e immediately before the method executes. The contract is *valid* if, for every configuration $\mathcal{C}$ satisfying Γ and every instance o of C, whenever o's state satisfies Pre before executing m, it satisfies $Post$ afterwards.

In the following, we will focus on specific contracts that are used in our compositional argument. In particular, for each class C and for each phase p of the scheduler, we search for Σ_C-expressions I_C, $T_C(p)$, and K_C such that the following contracts are valid:

$$\{\texttt{true}\}\ \texttt{init}()\ \{I_C\} \qquad \{\texttt{true}\}\ \texttt{exec}(p)\ \{T_C(p)\} \qquad \{\texttt{true}\}\ \texttt{tick}()\ \{K_C\}$$

These contracts can be generated automatically from class definitions and verified independently using standard program verification techniques.

Global Entailment Checks Using the local contracts above, we formulate a set of entailment checks showing that, for every configuration satisfying Γ, a global invariant Inv is preserved by every scheduler transition and implies φ at the end of each cycle. Sub-expressions wrapped in $\texttt{old}(\cdot)$ are evaluated in the pre-state (before the transition); all checks can be compiled to first-order validity queries dischargeable by SMT solvers. We write `All` for the union of all $\texttt{All}_{C_i}$.

Self-loop execution preserves Inv. For each self-loop scheduler transition at phase p, the scheduler invokes $\texttt{exec}(p)$ on every instance in an *arbitrary* order. To show that Inv is preserved by the whole interleaving, we reduce the problem to a *single-instance* check: it suffices to show that executing any one instance preserves Inv, then an induction on the number of executed instances results in preservation for the full interleaving.

For a single-instance step we need to know what holds for an instance c just before c executes. The global scheduler guard $G_{p\to p}$ holds at the start of the self-loop, but it may be invalidated by earlier executions of other instances. We therefore introduce a *local condition* g'_p—a per-instance Σ_C-expression that approximates the relevant information from the global guard and remains valid throughout the interleaving. Concretely, g'_p must satisfy two side-conditions:

(a) *Establishment:* $\Gamma \models \big(Inv \wedge G_{p\to p}\big) \Rightarrow c.g'_p$.
(b) *Stability* (for every class D and $d \neq c$ of type D): $\Gamma \models \big(\texttt{old}(Inv \wedge c.g'_p) \wedge d.T_D(p) \wedge d.\texttt{executed}\big) \Rightarrow c.g'_p$.

Given a valid g'_p, the main preservation check states that executing a single instance c of class C under g'_p preserves Inv:

$$\Gamma \models \Big(\texttt{old}\big(Inv\ \wedge\ \texttt{phase} = p\ \wedge\ c.g'_p\big)\ \wedge\ c.T_C(p)\ \wedge\ c.\texttt{executed}\Big)\ \Rightarrow\ Inv$$

In practice, a common choice for g'_p is $\neg c.\texttt{executed}$ (instance c has not yet executed in phase p), which is trivially stable under other instances' executions.

We then check additional simpler preservation conditions for the initialization, for phase transitions, as well as the implication of φ at the end of each cycle:

Initialization establishes Inv.

$$\Gamma \models \Big(\forall c \in \texttt{All}.\ \big(c.I_C \ \wedge\ \neg c.\texttt{executed}\big) \ \wedge\ \texttt{phase} = l_0\Big) \ \Rightarrow\ \mathit{Inv}$$

Phase transitions preserve Inv. For each scheduler edge $p \to p'$ with $p \neq p'$ and guard $g_{p \to p'}$:

- *Non-final transition ($p' \neq l_f$).* The scheduler updates its control location and resets all `executed` flags:

$$\Gamma \models \big(\texttt{old}(\mathit{Inv} \ \wedge\ \texttt{phase} = p \ \wedge\ g_{p \to p'}) \ \wedge\ \texttt{phase} = p' \ \wedge \\ \forall c \in \texttt{All}.\ \neg c.\texttt{executed}\big) \ \Rightarrow\ \mathit{Inv}$$

- *Final transition ($p' = l_f$).* The scheduler additionally invokes $\texttt{tick}()$ on each instance:

$$\Gamma \models \begin{pmatrix} \texttt{old}(\mathit{Inv} \ \wedge\ \texttt{phase} = p \ \wedge\ g_{p \to l_f}) \ \wedge\ \texttt{phase} = l_f \\ \wedge\ \forall c \in \texttt{All}.\ \big(\neg c.\texttt{executed} \ \wedge\ c.K_C\big) \end{pmatrix} \ \Rightarrow\ \mathit{Inv}$$

- *Reset ($l_f \to l_0$).* External inputs may change while non-input fields are preserved. Let $\text{InputChange} \equiv \bigwedge_{\text{non-input } f} \forall c \in \texttt{All}.\ c.f = \texttt{old}(c.f)$.

$$\Gamma \models \big(\texttt{old}(\mathit{Inv} \ \wedge\ \texttt{phase} = l_f) \ \wedge\ \text{InputChange} \ \wedge\ \texttt{phase} = l_0\big) \ \Rightarrow\ \mathit{Inv}$$

Inv implies the safety property. At the end of every cycle the invariant must imply the target property:

$$\Gamma \models \big(\mathit{Inv} \ \wedge\ \texttt{phase} = l_f\big) \ \Rightarrow\ \varphi$$

Theorem 1 (Compositional Soundness). *If all local contracts are valid and all global entailment checks succeed for an invariant Inv, then φ holds at the end of every scheduling cycle in every run of $\mathcal{S}$, for all configurations satisfying Γ.*

Proof sketch. The proof is by induction on scheduler transitions. Initialization establishes *Inv* by the initialization entailment check and valid local `init` contracts; for preservation, we consider both phase-change scheduler transitions and self-loop scheduler transitions, where the latter are further decomposed into individual instance transitions (with an inner induction over the execution interleaving), using the corresponding entailment checks together with valid local `exec` and `tick` contracts; finally, at states with $\texttt{phase} = l_f$, φ follows from *Inv* by the last entailment check. The full proof is provided in Appendix C.

More generally, the problem of finding a suitable invariant can be reduced to the (undecidable) problem of safety verification of parameterized systems [3].

5 Implementation in Dafny

The framework described in this paper has been implemented as a toolchain built on top of the Dafny verification system [22]. Dafny was chosen as it is a mature tool, which natively supports the combination of object-orientation, imperative statements, and quantified first-order logic, allowing a natural formalization of our models. The toolchain takes as input a description of a parameterized SRA system written in a high-level control-logic language, and generates a SysML description, executable C code, and a Dafny model for verification.

The implementation builds on an already existing toolchain. In previous work [9], the framework focused on showing the correctness of the generated executable C code, and as such the generated Dafny was made to resemble this implementation. Components were connected through lists, and assignments were performed by iteration over the lists. The new implementation instead uses the set-theoretic formalization presented in Sec. 3, and quantified assignments.

Furthermore, safety properties were previously proven to be inductive for each individual transition separately. Instead they are now proven over the `exec` method of each class, as described in Sec. 4.

5.1 Automatic Contract Generation

Local contracts for methods can be generated automatically from the class definitions. Thanks to the restrictions in our input language, this process is relatively straightforward. Due to lack of space, we cannot include the full details here, but we report them in Appendix B. Here, we only sketch the procedure for generating contracts for the `exec` method. At a high level, this involves the following three steps: (1) symbolic transformation of transition effects, (2) encoding of individual transitions, and (3) combination of the latter for the execution contract. For each transition effect $\mathcal{E}$, we compute a symbolic effect formula $\text{Effect}_{\mathcal{E}}$ that captures the state update performed by executing $\mathcal{E}$. This is done by forward symbolic execution, which computes a symbolic map $M_{\mathcal{E}}$ associating each variable with an expression over pre-state values. Assignments update the corresponding map entry; conditionals yield conditional expressions; and quantified assignments generate lambda expressions for function fields. The effects of individual transitions are then combined together to obtain the contract for the `exec` method as a disjunction of the possible transitions, in which individual transition guards are augmented with constraints enforcing the transition priority (a transition t is to be executed only if there exists no other t' declared before t in the class definition whose guard evaluates to true).

5.2 Optimization: Quantifier grounding

If configuration constraints include constraints of the form $|s| = k$ for set-valued field s, we can use this information to rewrite quantified expressions and statements into equivalent, simpler forms. Specifically, universal and existential quantifiers over s can be expanded into conjunctions or disjunctions over the k elements of s. A similar encoding can be applied in case of bound constraints of

the form $|s| \leq k$. This transformation can be applied both at the level of code (rewriting quantified statements in method bodies) and at the level of contracts (e.g., quantified postconditions), resulting in quantifier-free verification conditions that are easier for SMT solvers to handle. Furthermore, we can prove that these rewrites result in equivalent specifications, assuming that the constraints hold. Thus, when proving satisfaction of the quantifer grounded specifications, and the safety properties, the same holds for the original specifications.

We implement this optimization for constraints $k \leq 1$ in the Dafny encoding. We keep the original set-valued fields in the ghost state (i.e., state only accessible to specifications) and generate new object-valued fields (which are possibly nullable for bounded constraints) as the grounded version. We generate both the original and grounded version of every specification, over the respective fields. Then, we add lemmas stating the equivalence of the two versions, with two types of assumptions: (i) the sizes of the set-valued fields are according to the given constraints, and (ii) the object of each set-valued field equals the respective object-valued field. Such assumptions are added for every field-pair occuring in the specifications whose equivalence are being proven. For example, for a quantifier grounded nullable field s (i.e., now an object), we add also the original field as ghost, s_{ghost}. Then, for a grounded predicate p referencing s we generate also the original, non-grounded p_{ghost} referencing s_{ghost}, and a lemma stating, where $|s_{\text{ghost}}| \leq 1 \in \Gamma$:

$$\Gamma \wedge (|s_{\text{ghost}}| = 0 \Rightarrow s = \text{null}) \wedge (|s_{\text{ghost}}| = 1 \Rightarrow \{s\} = s_{\text{ghost}}) \models p \iff p_{\text{ghost}}.$$

6 Experimental Evaluation

Experimental Setup We evaluate the implementation of our framework on benchmarks from previous work on SystemC verification [28] and from industrial case studies obtained within a collaboration with the Italian railway infrastructure operator RFI[3]. The verified applications were designed by the RFI signaling engineers using the AIDA environment [2, 11], while formal verification experts interacted with the toolchain described in this paper. The cases used in this work have been presented in previous application papers [9, 10], where verification was performed with a preliminary version of the toolchain. The present results were obtained with a consolidated and generalized version of the framework, with the following differences. First, collections of objects are represented as sets (as well as lists); second, the toolchain implements a more general proof strategy, that is not limited to the AIDA domain specific scheduling policy, and that has been proved correct in the previous sections; third, the quantifier grounding transformation provably results in equivalent specifications (Section 5.2).

We evaluate the toolchain on three different aspects: verification of *local contracts* (i.e. all the methods satisfy the automatically generated candidate contracts), *safety properties*, and *quantifier grounding proofs*. We report the number

[3] Due to the properietary nature of the applications, the case studies cannot be publicly shared.

Table 1: Verification statistics for local contracts

Sys.	Version	# Ass.	Tot. time	Tot. RC	Exec. time	Exec. RC	Guard time	Guard RC	Eff. time	Eff. RC
RCS	Set	105	2s	2.72M	0s	1.55M	0s	0.16M	1s	1.65M
RPS	List	994	4233s	9436.56M	350s	724.29M	3465s	7034.48M	396s	1617.89M
	List+QG	998	925s	2529.53M	403s	671.75M	60s	188.41M	442s	1617.18M
	Set	963	729s	2154.06M	345s	675.18M	7s	21.37M	356s	1401.94M
	Set+QG	963	620s	1851.96M	234s	388.14M	5s	13.7M	362s	1401.69M
SCL	List	408	1001s	3814.77M	97s	346.38M	1s	2.4M	895s	3446.88M
	List+QG	411	924s	3528.9M	31s	90.37M	1s	1.6M	886s	3418.88M
	Set	399	969s	3763.57M	82s	317.41M	1s	1.91M	880s	3425.74M
	Set+QG	399	910s	3518.73M	22s	80.44M	1s	1.5M	880s	3418.78M

Table 2: Verification statistics summary for safety properties

Sys.	Prop.	**List**				**List+QG**				**Set**				**Set+QG**			
		Trans.		Exec.		Trans.		Exec.		Trans.		Exec.		Trans.		Exec.	
		Time	RC	Time	RC	Time	RC	Time	RC	Time	RC	Time	RC	Time	RC	Time	RC
RPS	Sum	16708s	42544M	755s	1899M	6195s	15508M	514s	1195M	12888s	33769M	797s	1908M	4096s	10064M	497s	1187M
	Max	2224s	5580M	116s	277M	370s	863M	72s	188M	1851s	4866M	92s	226M	238s	555M	59s	134M
SCL	1	13s	42M	11s	32M	12s	39M	13s	38M	12s	41M	12s	36M	11s	38M	13s	35M

of assertions, and the performance measured in terms of time elapsed and of Dafny RC[4]. The stated size of Dafny encodings includes the specifications.

We compare: (i) the current set-based formalization against the previous list-based formalization, (ii) the quantifier grounded encoding against the non-optimized one, (iii) the current proof strategy of proving safety properties over the `exec` method of each class, against the previous strategy with proofs over individual transitions. All the experiments were performed on a cluster of identical nodes (AMD EPYC 7413 24-Core Processor, 96CPU, 1.0TB RAM), with each verification task being assigned 8GB of RAM, and using Dafny 4.11.0.

Experimental Results The verification results for local contracts of all systems are summarized in Table 1. The *total* time and resource count (RC) denotes verification of all methods against their contracts. The *execute* column is the verification of only the `exec` method against its contract. The *guard* and *effect* columns show only the verification of guards and effects, respectively. Note that the set formalization contain no guard methods. The results for safety properties are summarized in Table 2. The *transition* columns are the results of verifying properties over each of the transitions, whereas the *execute* columns are the results for verifying directly over the `exec` method (which executes the individual transitions). The detailed results are reported in Appendix D.

Generalized Robot Controller System The Robot Controller System (RCS) is a SystemC case study proposed in [28], where a controller decides the movement based on the inputs of a *fixed number of sensors*. The property states that the controller never decides to move in a direction in which an obstacle is perceived. We generalize the RCS to include an unbounded number of sensors. The

[4] The Dafny resource count (RC) is a measure of the number of steps taken to complete the proof. RC is deterministic for a specific Dafny version.

Dafny encoding is made up of about 800 LoC. Verification of the safety property, which was not inductive and so was manually extended to form the invariant, took 18s and used 63.26M RC. The full details of the example can be found in Appendix A.

Railways Protection System (RPS) The RPS [9] is an industrial safety-critical system, developed by RFI, that ensures that workers can safely access railway lines for maintenance. We verify the control logic of the RPS, which is responsible for authorizing workers' requests to access the line based on the state of the railway network. The verification statistics are given for four different sets of generated codebases, each between 11k and 14k LoC (with the list-based ones on the higher end).

The safety properties express the fact that authorization cannot be given in the presence of unsafe inputs. We succesfully verify the same 21 safety properties as in [9], in addition to four new ones obtained from the railway engineers designing the system. All but four of the properties were already inductive over the state machine transitions. The non-inductive properties were manually extended to form the invariant. The summary gives total time and resource count for all properties, as well as the highest for a single property.

In the quantifier grounded version, verifying the 516 lemmas stating the equivalence between quantifier grounded and non-optimized versions of specifications took 60s (with a total of 179M RC).

Signal Control Logic (SCL) The SCL benchmark concerns the control logic for a railway signal system. The SCL was originally described in [10] to demonstrate the use of model checking techniques implemented in nuXmv [8] for the generation of invariants and counterexamples on manually generated finite abstractions of the SCL. As such, no direct comparison with the techniques proposed here is possible. We prove that at the end of each control cycle the signal aspects (colors) are set correctly.

Each generated code base is about 6k LoC. In the quantifier grounded version, verification of the 210 equivalence lemmas took 4s (with a total of 8M RC).

Discussion From the experimental results we can draw the following conclusions. First, the approach is applicable in practice to prove safety properties for all system configurations. Second, the approach scales to realistic case studies. The automated summarization of methods into candidate contracts is effective in reducing the manual effort required for verification, and requires reasonable computational effort. Third, the set-theoretical formalization is superior to the list-based one in terms of verification performance, hiding unnecessary details. Similarly, the quantifier grounding optimization significantly reduces verification time and resource consumption in most cases, although the effect is more dramatic for less optimized verification tasks (e.g., lists without grounding). Finally, the verification of safety properties over the `exec` method is significantly faster

than over each transition. We conjecture that this is due to a better usage of the underlying SMT solver in the `exec` version, which can reuse information across multiple branches instead of restarting for each transition. In general however, the careful control of the Dafny proof engine will require further investigation.

7 Conclusion

We presented a unified framework for the verification of configurable systems expressed as a collection of process types orchestrated by a domain-specific scheduler. Each process type is described as an extended finite-state machine, with actions attached to transitions expressed as imperative code. First-order quantified expressions and statements enable the direct logic-based representation of the space of possible configurations, with variable number of processes and complex interconnections. The verification approach is compositional and contract-based: proving that a global safety property holds at the end of every scheduling cycle is reduced to checking local contracts on each process type. The approach is implemented on top of Dafny, and relies on the automatic generation of contracts and annotations in order to reduce the manual effort required for verification. The experimental evaluation, carried out on several real-world case studies, demonstrated the generality, the effectiveness and the scalability of the approach.

There are several directions for future work. First, we intend to further improve automation. Currently, the decomposition of the global verification, the extraction of contracts from methods implementations, the application of configuration-specific optimizations and the compilation into Dafny all require no user intervention. However, in general the user may have to provide additional inductive invariants for the properties of interest. While in our case studies the invariants were often straightforward extensions of the desired properties, this remains a manual step. A promising direction is to leverage parameterized model checking techniques for automatic invariant discovery, e.g. [26, 31], possibly extended with abstraction strategies similar to [15, 24]. Our preliminary experiments in this direction show encouraging results, and we plan to develop a fully automated pipeline in future work. On a different direction, we intend to extend our approach to consider more general classes of properties, such as safety properties spanning multiple scheduling cycles and liveness properties. Finally, an important direction for future work will concern the application of formal or semi-formal techniques (e.g. equivalence checking, translation validation, or co-simulation) to establish the functional equivalence between the models used for verification and the actual (automatically generated) C implementations deployed in production.

References

1. Alberti, F., Ghilardi, S., Sharygina, N.: A framework for the verification of parameterized infinite-state systems. Fundam. Informaticae **150**(1), 1–24 (2017). https://doi.org/10.3233/FI-2017-1458, https://doi.org/10.3233/FI-2017-1458
2. Amendola, A., Becchi, A., Cavada, R., Cimatti, A., Griggio, A., Scaglione, G., Susi, A., Tacchella, A., Tessi, M.: A model-based approach to the design, verification and deployment of railway interlocking system. In: ISoLA (3). Lecture Notes in Computer Science, vol. 12478, pp. 240–254. Springer (2020)
3. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Information Processing Letters **22**(6), 307–309 (1986). https://doi.org/https://doi.org/10.1016/0020-0190(86)90071-2, https://www.sciencedirect.com/science/article/pii/0020019086900712
4. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability in parameterized verification. SIGACT News **47**(2), 53–64 (2016). https://doi.org/10.1145/2951860.2951873, https://doi.org/10.1145/2951860.2951873
5. Blom, S., Huisman, M.: The vercors tool for verification of concurrent programs. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8442, pp. 127–131. Springer (2014). https://doi.org/10.1007/978-3-319-06410-9"˙9, https://doi.org/10.1007/978-3-319-06410-9_9
6. Böckle, G., Pohl, K., van der Linden, F.: A framework for software product line engineering. In: Software Product Line Engineering, pp. 19–38. Springer (2005)
7. Campana, D., Cimatti, A., Narasamdya, I., Roveri, M.: An analytic evaluation of systemc encodings in promela. In: SPIN. Lecture Notes in Computer Science, vol. 6823, pp. 90–107. Springer (2011)
8. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 334–342. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9"˙22
9. Cavada, R., Cimatti, A., Griggio, A., Lidström, C., Redondi, G., Scaglione, G., Tessi, M., Trenti, D.: Automated parameterized verification of a railway protection system with dafny. In: Piskac, R., Rakamarić, Z. (eds.) Computer Aided Verification. pp. 364–376. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-98685-7"˙17
10. Cavada, R., Cimatti, A., Griggio, A., Lidström, C., Redondi, G., Tessi, M., Trenti, D.: Formal analysis of a railway signaling block designed in AIDA. In: ter Beek, M.H., Dutilleul, S.C., Lecomte, T. (eds.) Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - 6th International Conference, RSSRail 2025, Pisa, Italy, November 26-28, 2025, Proceedings. Lecture Notes in Computer Science, vol. 16236, pp. 303–312. Springer (2025). https://doi.org/10.1007/978-3-032-10762-6"˙23, https://doi.org/10.1007/978-3-032-10762-6_23
11. Cavada, R., Cimatti, A., Griggio, A., Susi, A.: A formal IDE for railways: Research challenges. In: SEFM Workshops. Lecture Notes in Computer Science, vol. 13765, pp. 107–115. Springer (2022)
12. Cimatti, A., Giunchiglia, F., Mongardi, G., Romano, D., Torielli, F., Traverso, P.: Formal verification of a railway interlocking system using model checking. Formal Aspects Comput. **10**(4), 361–380 (1998)

13. Cimatti, A., Narasamdya, I., Roveri, M.: Software model checking with explicit scheduler and symbolic threads. Log. Methods Comput. Sci. **8**(2) (2012). https://doi.org/10.2168/LMCS-8(2:18)2012, https://doi.org/10.2168/LMCS-8(2:18)2012
14. Cimatti, A., Narasamdya, I., Roveri, M.: Software model checking systemc. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **32**(5), 774–787 (2013)
15. Clarke, E.M., Talupur, M., Veith, H.: Proving ptolemy right: The environment abstraction framework for model checking concurrent systems. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 33–47. Springer (2008)
16. Committee, S.S.: Ieee standard for standard systemc language reference manual. IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011) pp. 1–618 (2023). https://doi.org/10.1109/IEEESTD.2023.10246125
17. Ghosal, S., Jonsson, B., Rümmer, P.: An active learning approach to synthesizing program contracts. In: Ferreira, C., Willemse, T.A.C. (eds.) Software Engineering and Formal Methods - 21st International Conference, SEFM 2023, Eindhoven, The Netherlands, November 6-10, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14323, pp. 126–144. Springer (2023). https://doi.org/10.1007/978-3-031-47115-5"̇8, https://doi.org/10.1007/978-3-031-47115-5_8
18. Große, D., Drechsler, R.: Formal verification of LTL formulas for systemc designs. In: ISCAS (5). pp. 245–248. IEEE (2003)
19. Gruler, A., Leucker, M., Scheidemann, K.D.: Modeling and model checking software product lines. In: FMOODS. Lecture Notes in Computer Science, vol. 5051, pp. 113–131. Springer (2008)
20. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving safety and liveness of practical distributed systems. Commun. ACM **60**(7), 83–92 (2017). https://doi.org/10.1145/3068608, https://doi.org/10.1145/3068608
21. Herber, P., Hünnemeyer, B.: Formal verification of systemc designs using the BLAST software model checker. In: ACES-MB@MoDELS. CEUR Workshop Proceedings, vol. 1250, pp. 44–53. CEUR-WS.org (2014)
22. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4"̇20, https://doi.org/10.1007/978-3-642-17511-4_20
23. Li, J., Li, Q., Li, J.: The w-model for testing software product lines. In: ISCSCT (1). pp. 690–693. IEEE Computer Society (2008)
24. McMillan, K.L.: Eager abstraction for symbolic model checking. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 191–208. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3"̇11, https://doi.org/10.1007/978-3-319-96145-3_11
25. McMillan, K.L., Padon, O.: Ivy: A multi-modal verification tool for distributed algorithms. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 190–202. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8"̇12, https://doi.org/10.1007/978-3-030-53291-8_12

26. Redondi, G., Cimatti, A., Griggio, A., McMillan, K.L.: Invariant checking for smt-based systems with quantifiers. ACM Trans. Comput. Log. **25**(4), 1–37 (2024). https://doi.org/10.1145/3686153, https://doi.org/10.1145/3686153
27. Scaletta, M., Hähnle, R., Steinhöfel, D., Bubel, R.: Delta-based verification of software product families. In: GPCE. pp. 69–82. ACM (2021)
28. Tasche, P., Monti, R.E., Drerup, S.E., Blohm, P., Herber, P., Huisman, M.: Deductive verification of parameterized embedded systems modeled in systemc. In: Dimitrova, R., Lahav, O., Wolff, S. (eds.) Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14500, pp. 187–209. Springer (2024). https://doi.org/10.1007/978-3-031-50521-8"̇9, https://doi.org/10.1007/978-3-031-50521-8_9
29. Thüm, T., Schaefer, I., Hentschel, M., Apel, S.: Family-based deductive verification of software product lines. In: GPCE. pp. 11–20. ACM (2012)
30. Wilcox, J.R., Feldman, Y.M.Y., Padon, O., Shoham, S.: mypyvy: A research platform for verification of transition systems in first-order logic. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14682, pp. 71–85. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9"̇4, https://doi.org/10.1007/978-3-031-65630-9_4
31. Yao, J., Tao, R., Gu, R., Nieh, J.: Duoai: Fast, automated inference of inductive invariants for verifying distributed protocols. In: Aguilera, M.K., Weatherspoon, H. (eds.) 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022. pp. 485–501. USENIX Association (2022), https://www.usenix.org/conference/osdi22/presentation/yao

A Extended Example: Robot Controller System

We present an example of a configurable SRA system modeling a robot controller with multiple sensors, inspired by a SystemC case study [28]. The system consists of a controller class and a sensor class, where the controller reads sensor states and decides on movement direction. The scheduler has four phases: `Sense` (the intial phase), `Act`, `Reset`, and `End`.

A.1 Class Definitions

We start by defining the classes for the controller and sensors, along with their local transitions. **Sensor Class:**

```
class Sensor {
  var location : SensLoc        // {Ready, Go, NoGo}
  var executed : Bool
  input obstacle : Bool         // external input
  event processed : Event       // set by controller, treated as a boolean

  transition senseGo = (Ready, !obstacle, Go, { }, Sense)
  transition senseNoGo = (Ready, obstacle, NoGo, { }, Sense)
  transition resetGo = (Go, processed, Ready, { }, Act)
  transition resetNoGo = (NoGo, processed, Ready, { }, Act)
}
```

The `Sensor` class detects obstacles and transitions between control states: `Ready` (waiting to sense), `Go` (no obstacle), or `NoGo` (obstacle detected). It has an event `processed` set by the controller to acknowledge the sensor reading. The initialize method (not shown) sets the location to `Ready`.

Controller Class:

```
class Controller {
  var location : CtrlLoc        // {Idle, Moving}
  var executed : Bool
  var direction : Direction     // {Forward, Stop, Left, Right}
  set leftSensors : Set<Sensor>
  set rightSensors : Set<Sensor>
  set allSensors : Set<Sensor> //{ leftSensors + rightSensors }

  transition actRight = (Idle,
    (exists s in leftSensors : s.location == NoGo) &&
    (forall s in rightSensors : s.location == Go),
    Moving, { direction := Right;
      forall s in allSensors { s.processed := true; } }, Act)
  transition actLeft = (Idle,
    (exists s in rightSensors : s.location == NoGo) &&
    (forall s in leftSensors : s.location == Go),
```

```
    Moving, { direction := Left;
      forall s in allSensors { s.processed := true; } }, Act)
  // similar transition for actForward and actStop, omitted for brevity
  transition reset = (Moving, true, Idle, { }, Reset)

}
```

The `Controller` class owns sets of sensors and reads their state via quantified expressions. It decides on movement direction based on sensor readings and signals sensors when it has processed their state. The initialize method (not shown) sets the location to `Idle` and direction to `Stop`.

Scheduler. The scheduler in this example has a set of locations $\texttt{PhaseEnum} = \{\texttt{Sense}, \texttt{Act}, \texttt{Reset}, \texttt{End}\}$, with initial phase `Sense` and final phase `End`. The scheduler transitions are defined as follows:

1. *Self-loop on* ***`Sense`*** $(p = p' = \texttt{Sense})$, with guard $G_1 \equiv \forall x \in \texttt{All}.\ \neg x.\texttt{executed}$ This transition triggers the local execution of all instances in the `Sense` phase. Since at the start of each cycle $\texttt{executed} = \texttt{false}$ for all instances, the guard is satisfied. After execution, each instance's `executed` flag is set to `true`, disabling the guard.
2. *Phase change from* ***`Sense`*** *to* ***`Act`*** $(p = \texttt{Sense},\ p' = \texttt{Act})$, with guard $\neg G_1$. This transition fires once all instances have executed in the `Sense` phase.
3. *Self-loop on* ***`Act`*** $(p = p' = \texttt{Act})$, with guard: $G_2 \equiv \forall x \in \texttt{All}.\ \big(\neg x.\texttt{executed}\big) \vee (\exists s \in \texttt{All}_{\texttt{Sensor}}.\ s.\texttt{processed})$ This transition repeatedly invokes local execution in the `Act` phase. The guard ensures that the self-loop continues as long as either some instance has not yet executed in this phase or there remain sensors with a pending `processed` event, forcing the system to iterate until all events are consumed.
4. *Phase change from* ***`Act`*** *to* ***`Reset`*** $(p = \texttt{Act},\ p' = \texttt{Reset})$, with guard $\neg G_2$. This transition fires once all instances have executed in the `Act` phase and no pending events remain.
5. *Self-loop on* ***`Reset`*** $(p = p' = \texttt{Reset})$, with guard $G_3 \equiv \forall x \in \texttt{All}.\ \neg x.\texttt{executed}$ This transition triggers local execution for the `Reset` phase.
6. *Phase change from* ***`Reset`*** *to* ***`End`*** $(p = \texttt{Reset},\ p' = \texttt{End})$, with guard $\neg G_3$

A scheduling cycle thus consists of the transition sequence $S_0 \xrightarrow{\texttt{Sense}\to\texttt{Sense}} S_1 \xrightarrow{\texttt{Sense}\to\texttt{Act}} S_2 \xrightarrow{\texttt{Act}\to\texttt{Act}} \cdots \xrightarrow{\texttt{Act}\to\texttt{Reset}} S_j \xrightarrow{\texttt{Reset}\to\texttt{Reset}} S_{j+1} \xrightarrow{\texttt{Reset}\to\texttt{End}} S_n$, where the self-loop on `Act` may iterate multiple times until all events are processed.

Configuration constraints. The system's structural constraints are given by the following configuration constraints:

$\Gamma_1 \equiv \forall c :: c \text{ in } \mathtt{All}_{\mathtt{Controller}}.\ c.\mathtt{leftSensors} \mathbin{!!} c.\mathtt{rightSensors}$

$\Gamma_2 \equiv \forall c_1 :: c_1 \text{ in } \mathtt{All}_{\mathtt{Controller}}.\ \forall c_2 :: c_2 \text{ in } \mathtt{All}_{\mathtt{Controller}}.\ c_1 \neq c_2 \Rightarrow$
$(c_1.\mathtt{leftSensors} \cup c_1.\mathtt{rightSensors}) \mathbin{!!} (c_2.\mathtt{leftSensors} \cup c_2.\mathtt{rightSensors})$

$\Gamma_3 \equiv \forall c :: c \text{ in } \mathtt{All}_{\mathtt{Controller}}.\ |c.\mathtt{leftSensors}| \geq 1$

$\Gamma_4 \equiv \forall c :: c \text{ in } \mathtt{All}_{\mathtt{Controller}}.\ |c.\mathtt{rightSensors}| \geq 1$

$\Gamma_5 \equiv \forall c :: c \text{ in } \mathtt{All}_{\mathtt{Controller}}.\ c.\mathtt{leftSensors} \cup c.\mathtt{rightSensors} = \mathtt{allSensors}$

We denote with Γ the set $\{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4, \Gamma_5\}$. These constraints ensure every controller has at least one sensor on each side, that left and right sensor sets are disjoint within each controller, and that no sensor is shared between different controllers.

Global Safety Property. The system must satisfy the following safety property:

$Prop \equiv \forall c :: c \text{ in } \mathtt{All}_{\mathtt{Controller}}.\ ((\exists s :: s \text{ in } c.\mathtt{leftSensors}.\ s.\mathtt{obstacle}) \Rightarrow c.\mathtt{direction} \neq \mathtt{Left})$

This property states that if any left sensor has detected an obstacle, the controller is not moving left. In our semantics, this property must hold in all reachable states at the end of each scheduler cycle. In fact, the property can also be violated during intermediate phases within cycles.

A.2 Configuration Example

A simple configuration $\mathcal{C}$ for the example consists of:

- One controller instance: $\mathcal{U}_{\text{Controller}} = \{c_1\}$
- Three sensor instances: $\mathcal{U}_{\text{Sensor}} = \{s_L, s_{R1}, s_{R2}\}$
- Set field interpretations: c_1.leftSensors $= \{s_L\}$, c_1.rightSensors $= \{s_{R1}, s_{R2}\}$, and c_1.allSensors $= \{s_L, s_{R1}, s_{R2}\}$

This configuration clearly satisfies the configuration constraints Γ.

A.3 Execution Scenario

We illustrate a sample execution scenario over the above configuration $\mathcal{C}$. Initially, the controller c_1 is in the `Idle` state, its direction is set to `Stop`, and `executed` is `false`. All sensors, namely s_L, s_{R1}, and s_{R2}, are in the `Ready` state, each with `executed` set to `false`, and their `processed` event is false. The initial scheduler phase is `Sense`.

For the external inputs, suppose that the left sensor detects an obstacle, i.e., s_L.obstacle = true, while both right sensors detect no obstacle, so s_{R1}.obstacle = false and s_{R2}.obstacle = false.

The execution proceeds in scheduler transitions. The first transition is the self-transition of the `Sense` phase, which execute the local transitions of all instances in the `Sense` phase. In particular, we have:

- *Sensors:*
  - s_L detects an obstacle (s_L.obstacle = true), so it executes `senseNoGo` transition: `Ready` → `NoGo`
  - s_{R1} detects no obstacle (s_{R1}.obstacle = false), so it executes `senseGo` transition: `Ready` → `Go`
  - s_{R2} detects no obstacle (s_{R2}.obstacle = false), so it executes `senseGo` transition: `Ready` → `Go`
- *Controller:* Stutters (no transitions in `Sense` phase)

Moreover, each instance's `executed` flag is set to `true`. Then, the next scheduler transition is from `Sense` phase to `Act` phase, which updates the scheduler global phase and resets `executed` to `false` for all instances. This is followed by the `Act` phase self-transition, where again all instances execute their local transitions in the `Act` phase. For example, we have:

- *Controller:* The controller c_1 is in `Idle` state. It evaluates its transition guards for the local execution:
  - `actRight` guard: $(\exists s \in \{s_L\} : s.\text{location} = \texttt{NoGo}) \wedge (\forall s \in \{s_{R1}, s_{R2}\} : s.\text{location} = \texttt{Go})$
  - This evaluates to: $(s_L.\text{location} = \texttt{NoGo}) \wedge (s_{R1}.\text{location} = \texttt{Go} \wedge s_{R2}.\text{location} = \texttt{Go})$, which is true in the current state.
- *Effect:* Controller executes `actRight`:
  - Sets c_1.direction := `Right`
  - Moves to c_1.location := `Moving`
  - Sets s_L.processed := true, s_{R1}.processed := true and s_{R2}.processed := true
- *Sensors:* Check for `processed` events:
  - s_L has processed = true, executes `resetNoGo`: `NoGo` → `Ready`
  - s_{R1} has processed = true, executes `resetGo`: `Go` → `Ready`
  - s_{R2} has processed = true, executes `resetGo`: `Go` → `Ready`

All sensors are back in `Ready` state and have their `processed` event reset to false. Note that if we had switched the execution order within the `Act` phase, the sensors would have stuttered, and the events would have remained true, triggering an additional act phase. Instead, we now take the scheduler transition from `Act` phase to `Reset` phase, followed by the `Reset` phase self-transition; here, only the controller has a transition: it executes `reset` transition: `Moving` → `Idle`.

At the end of Cycle 1, the controller has decided to move right and returned to `Idle` state. The direction variable retains the value `Right`. Before starting Cycle 2, the external inputs (sensor obstacle values) may change: we might start in a situation where the controller is moving right, and right sensors are detecting obstacles, forcing the controller to change direction at the end of cycle 2.

A.4 Local Contracts

We report an example for a valid local contract for the `Controller`'s `exec` method in the `Act` phase. Let $\mathit{leftNoGo} \equiv (\exists s \in \texttt{leftSensors}.\ s.\texttt{location} = \texttt{NoGo})$ and $\mathit{rightGo} \equiv (\forall s \in \texttt{rightSensors}.\ s.\texttt{location} = \texttt{Go})$. A valid postcondition capturing the `actRight` case is:

$$\begin{aligned} T_{\texttt{Controller}}(\texttt{Act}) \equiv\ & \big(\texttt{old}(\texttt{location}) = \texttt{Idle} \wedge \texttt{old}(\mathit{leftNoGo}) \wedge \texttt{old}(\mathit{rightGo})\big) \\ & \wedge \big(\texttt{location} = \texttt{Moving} \wedge \texttt{direction} = \texttt{Right} \\ & \quad \wedge\ (\forall s \in \texttt{All}_{\texttt{Sensor}}.\ s.\texttt{processed} = (\text{if } s \in \texttt{allSensors} \text{ then } \texttt{true} \\ & \qquad \text{else } \texttt{old}(s.\texttt{processed}))))\big) \\ \vee\ \ldots\ & \quad \text{(other transitions handled similarly)} \end{aligned}$$

B Automatic Contract Generation

B.1 Initialization Contract

The initialization contract I_C for a class C directly reflects the semantics of the `init` method. It is defined as a conjunction of equalities of the form $v = e$, where v is a field of the class and e is a constant expression.

B.2 Execute contracts

Contracts for `exec` methods are generated in three steps: (1) symbolic transformation of transition effects, (2) encoding of individual transitions, and (3) combination of the latter for the execution contract.

Step 1: Symbolic transformation of transition effects. For each transition effect $\mathcal{E}$, we compute a symbolic effect formula $\text{Effect}_{\mathcal{E}}$ that captures the state update performed by executing $\mathcal{E}$. This is done by forward symbolic execution, which computes a symbolic map $M_{\mathcal{E}}$ associating each variable with an expression over pre-state values. Assignments update the corresponding map entry; conditionals yield conditional expressions; and quantified assignments generate lambda expressions for function fields. Method calls occurring in $\mathcal{E}$ are inlined by substituting their bodies, so that the transformation operates on a single expanded statement sequence.

A detailed description of this transformation is provided in Algorithm 1. It maintains a *symbolic map* M that tracks the cumulative effect of assignments, handling sequential composition by chaining updates. Given an effect E of a class C, the transformation initializes $M_0(v) = v_{\text{old}}$ for each of the variables v that are fields of C and are assigned to in the effect, and $M_0(f) = \lambda y.w_{\text{old}}(y)$ for each variable w that is updated by a quantified assignment in the effect, then computes the final map $M_E = \textsc{Transform}(E, M_0)$. The $\textsc{Subst}$ function substitutes all variable occurrences using the current symbolic map.

Algorithm 1 Effect Transformation: TRANSFORM(Statement S, Map M)

1: **Input:** Statement S, Symbolic map M
2: **Output:** Updated symbolic map M'
3:
4: **if** S is $v := e$ **then**
5: $M'(v) \leftarrow \textsc{Subst}(e, M)$
6: $M'(w) \leftarrow M(w)$ for all $w \neq v$
7: **else if** S is $v := *$ **then**
8: $M'(v) \leftarrow \varepsilon$ ▷ havoc: no constraint on v
9: $M'(w) \leftarrow M(w)$ for all $w \neq v$
10: **else if** S is **if** b **then** S_1 **else** S_2 **then**
11: $M_1 \leftarrow \textsc{Transform}(S_1, M)$
12: $M_2 \leftarrow \textsc{Transform}(S_2, M)$
13: **for** each variable v **do**
14: $M'(v) \leftarrow$ **if** $\textsc{Subst}(b, M)$ **then** $M_1(v)$ **else** $M_2(v)$
15: **end for**
16: **else if** S is $\forall x \in E_{set}\{x.f := e\}$ (quantified assignment) **then**
17: $M'(f) \leftarrow \lambda y.$**if** $y \in E_{set}$ **then** $\textsc{Subst}(e[x \mapsto y], M)$ **else** $M(f)(y)$
18: $M'(w) \leftarrow M(w)$ for all $w \neq f$
19: **else if** S is $S_1; S_2$ **then**
20: $M_{temp} \leftarrow \textsc{Transform}(S_1, M)$
21: $M' \leftarrow \textsc{Transform}(S_2, M_{temp})$
22: **end if**
23: **return** M'

The resulting effect formula is defined as:

$$\text{Effect}_{\mathcal{E}} \equiv \bigwedge_{v \in \text{modified(self)}} v = M_{\mathcal{E}}(v) \wedge \bigwedge_{w \in \text{modified(other)}} \forall x :: x \in \texttt{All}.\ x.w = M_{\mathcal{E}}(w),$$

where $M_{\mathcal{E}}(v)$ and $M_{\mathcal{E}}(w)$ are the symbolic expressions computed for fields of the current object and for fields of other objects accessed via set-valued references, respectively.

Remark 1 (Strongest Postcondition). Existentially quantifying the pre-state variables in $\text{Effect}_{\mathcal{E}}$ yields a formula equivalent to the strongest postcondition of the effect $\mathcal{E}$.

Step 2: Encoding of transitions. Each class transition $t = (l_{\text{start}}, G, l_{\text{end}}, \mathcal{E}, p)$ is encoded as a transition formula Trans_t:

$$\begin{aligned}\text{Trans}_t \equiv\ & \texttt{old}(\text{location} = l_{\text{start}} \wedge \widehat{G}_t) \\ & \wedge \texttt{Effect}_{\mathcal{E}} \wedge \text{location} = l_{\text{end}} \\ & \wedge \texttt{executed} \\ & \wedge \texttt{ResetEvents}(G) \wedge \texttt{Unchanged}.\end{aligned}$$

The predicate $\widehat{G}_t$ is the *extended guard*, which enforces transition priority:

$$\widehat{G}_t \equiv G \wedge \bigwedge_{t' \prec t,\ t'.\text{start}=l_{\text{start}},\ t'.\text{end}=l_{\text{end}}} \neg G_{t'}.$$

Here $t' \prec t$ indicates that transition t' is declared before t in the class definition. Since transitions are evaluated in declaration order, t may fire only if its own guard holds and all higher-priority guards with the same source and target locations are false. The remaining components of Trans_t serve auxiliary purposes:

- `ResetEvents`(G) resets to `false` any event fields referenced in the guard G;
- `Unchanged` is an expression asserting that all variables not modified by E retain their pre-state values.

Step 3: Execution Contracts. For a given phase p, the execution contract $Exec_C^p$ for class C is defined as the disjunction of all transition formulas associated with phase p, augmented with a stuttering case:

$$Exec_C^p \equiv \bigvee_{\substack{t \in \text{transitions}(C) \\ t.\text{phase}=p}} \text{Trans}_t \ \vee \ \text{Stutter}.$$

The predicate Stutter asserts that all state variables remain unchanged and applies when no transition in phase p is enabled, i.e., when all corresponding guards are false.

B.3 Tick Timer Contracts

The tick-timer contract K_C for a class C is also generated automatically. Its effect consists of decrementing all active timers declared in the class by one and updating the state of any timer that reaches zero (e.g., by marking it as `inactive`). Also this expression can be derived trivially from the class definition.

C Proof of Theorem 1

Here we provide a proof of Theorem 1 (Compositional Soundness), clarifying the connection to the local contracts and global entailment checks in Section 4.2.

Theorem 1 (Compositional Soundness). Given a configurable SRA system $\mathcal{S}$, a set of configuration constraints Γ, and a global property φ, if all local contracts are valid and all global entailment checks in Section 4.2 succeed for an invariant Inv, then φ holds at the end of every scheduling cycle in every run of $\mathcal{S}$, for all configurations satisfying Γ.

Proof. Fix an arbitrary configuration $\mathcal{C}$ such that $\mathcal{C} \models \Gamma$. We first prove that the invariant *Inv* holds in all reachable global states. We then use this fact to establish the safety property φ.

Invariant preservation. We proceed by induction on the length of a run, showing that for every reachable global state S, we have $\mathcal{C}, S \models \mathit{Inv}$.

Base case. Let S_0 be the initial global state produced by system initialization. For each instance o in $\mathcal{C}$, the local state of o in S_0 is a valid post-state of the `init` method. By validity of the initialization contracts, I_C holds for all instances in S_0.

Moreover, by construction, $S_0(\texttt{phase}) = l_0$ and all instances satisfy $\neg\texttt{executed}$. Hence S_0 satisfies the antecedent of the initialization check ("Initialization establishes *Inv*"). By the validity of this check, we conclude $\mathcal{C}, S_0 \models \mathit{Inv}$.

Inductive step. Assume that $\mathcal{C}, S \models \mathit{Inv}$ for some reachable global state S. Let S' be any state reachable from S by one scheduler transition. We distinguish cases according to the form of the scheduler transition.

Case 1: Scheduler self-loop. Suppose $S(\texttt{phase}) = p$ and the scheduler takes the self-loop $p \to p$. This corresponds to an arbitrary interleaving of `exec`(p) invocations.

We model this execution as a sequence

$$S = \sigma_0 \to \sigma_1 \to \cdots \to \sigma_n = S',$$

where each step $\sigma_i \to \sigma_{i+1}$ corresponds to executing `exec`(p) on a single instance o_i.

We prove by induction on i that for all $0 \leq i \leq n$:

1. $\mathcal{C}, \sigma_i \models \mathit{Inv}$, and
2. for every instance c that has not yet executed in phase p at σ_i, $\mathcal{C}, \sigma_i \models c.G'$.

Base case ($i = 0$). By the outer induction hypothesis, $\mathcal{C}, S \models \mathit{Inv}$. Since the self-loop is taken, S also satisfies the scheduler guard $g_{p \to p}$. By the self-loop preservation check, condition (a) (establishment of G'), we obtain $\mathcal{C}, \sigma_0 \models c.G'$ for all instances c. Thus both claims hold for σ_0.

Inductive step. Assume the claims hold for σ_i, with $i < n$. The state σ_{i+1} is obtained by executing `exec`(p) on instance o_i and setting o_i`.executed` to `true`.

By validity of the local execution contract, `exec`(p) satisfies $T_C(p)$. Since the antecedent of the self-loop main preservation entailment holds at σ_i, its validity yields $\mathcal{C}, \sigma_{i+1} \models \mathit{Inv}$.

Moreover, for any instance $c \neq o_i$ that has not yet executed in phase p, the self-loop preservation check, condition (b) (stability) applies, with d instantiated as o_i. Thus $\mathcal{C}, \sigma_{i+1} \models c.G'$ holds for all such instances.

This completes the inner induction, and therefore $\mathcal{C}, S' \models \mathit{Inv}$ holds after the self-loop.

Case 2: Scheduler phase change. Suppose the scheduler transitions from phase p to phase p' with guard $g_{p \to p'}$. By the outer induction hypothesis, we have

$$\mathcal{C}, S \models \Gamma \wedge \mathit{Inv} \wedge \texttt{phase} = p \wedge g_{p \to p'}.$$

We distinguish cases depending on the target location.

Case 2a: Non-final transition ($p' \neq l_f$). In this case, S' is obtained by updating the scheduler location to p' and resetting all `executed` flags to `false`. Thus the antecedent of the phase-transition non-final entailment is satisfied. By its validity, we conclude

$$\mathcal{C}, S' \models \mathit{Inv}.$$

Case 2b: Final transition ($p' = l_f$). In this case, the scheduler transitions to the final location and additionally resets all `executed` flags to `false` and invokes `tick` on all instances. By validity of the local tick contracts, K_C holds for all instances. Thus the antecedent of the phase-transition final entailment holds in S'. By its validity, we conclude

$$\mathcal{C}, S' \models \mathit{Inv}.$$

Case 2c: Reset transition ($l_f \to l_0$). Finally, suppose the scheduler transitions implicitly from the final location l_f back to the initial location l_0. By the outer induction hypothesis, we have

$$\mathcal{C}, S \models \Gamma \wedge \mathit{Inv} \wedge \texttt{phase} = l_f.$$

By construction of the reset transition, external input variables may change, while all non-input fields remain unchanged. Thus S' satisfies the predicate InputChange and $\texttt{phase} = l_0$. The antecedent of the phase-transition reset entailment therefore holds. By its validity, we conclude

$$\mathcal{C}, S' \models \mathit{Inv}.$$

This completes the inductive proof that Inv holds in all reachable states.

Establishing the safety property. Let S_i be a global state at the end of a scheduling cycle, i.e., $S_i(\texttt{phase}) = l_f$. By invariant preservation, $\mathcal{C}, S_i \models \mathit{Inv}$. Moreover, S_i is obtained by a final scheduler transition. By the validity of the safety-implication check ("*Inv* implies the safety property"), we conclude $\mathcal{C}, S_i \models \varphi$.

Since the configuration $\mathcal{C}$ and the run were arbitrary, the claim follows.

D Detailed Experiment Results

For the Railway Protection System, Table 3 shows the full verification statistics for local contracts, and Table 4 the statistics for safety properties. In Table 4, properties 1–21 are the properties verified in [9], while properties 22–25 are new.

For the Signal Control Logic, Table 5 shows the full verification statistics for local contracts.

Table 3: Verification statistics for local contracts of the RPS

Version	Class	# Ass.	Tot. time	Tot. RC	Exec. time	Exec. RC	Guard time	Guard RC	Eff. time	Eff. RC
List	1	88	1380s	2710.11M	7s	22.85M	1370s	2678.98M	1s	3.57M
	2	102	1069s	2154.83M	6s	20.47M	1059s	2123.46M	2s	5.22M
	3	155	1036s	2229.82M	11s	33.91M	1020s	2181.6M	2s	6.83M
	4	73	3s	7.94M	1s	2.0M	0s	0.46M	1s	2.5M
	5	31	1s	1.13M	0s	0.13M	0s	0.01M	0s	0.17M
	6	128	14s	48.75M	3s	11.71M	0s	0.5M	9s	31.99M
	7	415	730s	2283.95M	322s	633.21M	15s	49.47M	381s	1567.6M
	Sum	994	4233s	9436.56M	350s	724.29M	3465s	7034.48M	396s	1617.89M
List+QG	1	88	26s	78.58M	7s	20.68M	16s	50.87M	1s	3.38M
	2	102	22s	72.42M	6s	18.95M	13s	44.03M	2s	5.02M
	3	155	29s	84.4M	10s	28.54M	15s	43.23M	2s	6.62M
	4	73	3s	6.6M	0s	1.65M	0s	0.2M	1s	2.4M
	5	31	1s	1.13M	0s	0.13M	0s	0.01M	0s	0.17M
	6	128	13s	48.75M	4s	11.71M	0s	0.5M	8s	31.99M
	7	419	831s	2237.63M	377s	590.09M	16s	49.57M	427s	1567.6M
	Sum	998	925s	2529.53M	403s	671.75M	60s	188.41M	442s	1617.18M
Set	1	84	44s	117.62M	39s	105.98M	1s	3.59M	1s	3.4M
	2	98	43s	113.23M	39s	98.74M	1s	3.85M	2s	5.04M
	3	148	59s	144.85M	52s	125.79M	1s	5.0M	2s	6.62M
	4	69	3s	7.53M	1s	1.91M	0s	0.29M	1s	2.41M
	5	29	1s	1.12M	0s	0.14M	0s	0.0M	0s	0.17M
	6	126	13s	47.73M	3s	12.06M	0s	0.45M	8s	30.73M
	7	407	565s	1721.96M	211s	330.57M	3s	8.2M	342s	1353.57M
	Sum	963	729s	2154.06M	345s	675.18M	7s	21.37M	356s	1401.94M
Set+QG	1	84	5s	13.87M	2s	5.64M	1s	1.3M	1s	3.34M
	2	98	6s	17.91M	2s	7.11M	1s	1.49M	2s	4.98M
	3	148	9s	26.83M	4s	12.38M	1s	1.89M	2s	6.54M
	4	69	2s	6.34M	0s	1.49M	0s	0.18M	1s	2.36M
	5	29	1s	1.12M	0s	0.14M	0s	0.0M	0s	0.17M
	6	126	13s	47.73M	3s	12.06M	0s	0.45M	8s	30.73M
	7	407	584s	1738.14M	223s	349.33M	3s	8.4M	348s	1353.57M
	Sum	963	620s	1851.96M	234s	388.14M	3s	13.7M	362s	1401.69M

Table 4: Verification statistics for safety properties of the RPS

	List				**List+QG**				**Set**				**Set+QG**			
Prop.	Trans.		Exec.		Trans.		Exec.		Trans.		Exec.		Trans.		Exec.	
	Time	RC	Time	RC	Time	RC	Time	RC	Time	RC	Time	RC	Time	RC	Time	RC
1	475s	1458M	15s	44M	356s	856M	18s	40M	388s	991M	24s	59M	195s	480M	18s	40M
2	576s	1399M	18s	47M	333s	792M	17s	38M	379s	959M	24s	58M	215s	555M	17s	36M
3	1641s	4394M	80s	200M	161s	383M	61s	143M	1412s	4078M	91s	215M	144s	381M	59s	126M
4	579s	1368M	20s	58M	370s	863M	14s	28M	357s	888M	22s	52M	237s	528M	15s	32M
5	564s	1349M	19s	56M	351s	788M	13s	26M	381s	881M	20s	48M	205s	473M	15s	33M
6	596s	1412M	17s	43M	361s	827M	15s	29M	402s	931M	23s	56M	232s	511M	15s	31M
7	568s	1346M	21s	61M	347s	854M	11s	27M	391s	866M	23s	59M	238s	539M	18s	46M
8	567s	1348M	17s	45M	341s	851M	13s	32M	428s	1018M	21s	53M	213s	480M	16s	35M
9	562s	1315M	18s	50M	293s	728M	12s	29M	382s	886M	20s	48M	236s	524M	20s	49M
10	425s	1030M	20s	48M	312s	794M	11s	33M	308s	755M	21s	52M	164s	398M	20s	42M
11	403s	1004M	18s	50M	250s	650M	12s	29M	288s	722M	20s	49M	136s	340M	16s	34M
12	398s	987M	17s	45M	308s	796M	13s	33M	300s	750M	21s	49M	133s	323M	16s	35M
13	606s	1498M	18s	45M	303s	770M	16s	43M	332s	856M	24s	59M	184s	448M	20s	43M
14	550s	1260M	19s	49M	321s	819M	15s	39M	320s	798M	25s	60M	210s	523M	17s	37M
15	2224s	5580M	98s	233M	161s	405M	42s	106M	1592s	4490M	90s	209M	135s	348M	43s	96M
16	668s	1554M	18s	46M	305s	769M	16s	41M	331s	834M	22s	51M	177s	448M	16s	43M
17	563s	1276M	19s	48M	327s	818M	15s	38M	292s	774M	32s	75M	208s	523M	14s	37M
18	1750s	4534M	116s	277M	148s	402M	72s	188M	1703s	4460M	92s	226M	133s	343M	43s	115M
19	580s	1362M	18s	45M	270s	743M	17s	39M	305s	813M	24s	60M	208s	532M	13s	37M
20	502s	1406M	20s	44M	265s	723M	17s	37M	314s	816M	23s	57M	188s	472M	13s	38M
21	1472s	4386M	101s	249M	143s	379M	54s	117M	1851s	4866M	84s	196M	147s	383M	50s	134M
22	108s	319M	13s	29M	43s	124M	9s	16M	110s	334M	13s	29M	39s	127M	6s	16M
23	110s	319M	11s	29M	43s	124M	9s	16M	108s	334M	13s	30M	40s	127M	6s	18M
24	108s	319M	11s	29M	42s	124M	10s	16M	109s	334M	14s	31M	39s	127M	6s	16M
25	111s	319M	12s	29M	42s	124M	9s	16M	107s	334M	12s	28M	41s	127M	7s	18M
Sum	16708s	42544M	755s	1899M	6195s	15508M	514s	1195M	12888s	33769M	797s	1908M	4096s	10064M	497s	1187M
Max	2224s	5580M	116s	277M	370s	863M	72s	188M	1851s	4866M	92s	226M	238s	555M	59s	134M

Table 5: Verification statistics summary for local contracts of the SCL

Version	Class	# Ass.	Tot. time	Tot. RC	Exec. time	Exec. RC	Guard time	Guard RC	Eff. time	Eff. RC
List	1	32	7s	24.51M	0s	1.11M	0s	0.03M	6s	22.02M
	2	196	305s	1030.73M	22s	73.94M	1s	1.32M	279s	946.44M
	3	40	7s	28.08M	0s	0.82M	0s	0.25M	6s	25.46M
	4	135	682s	2730.86M	74s	270.51M	0s	0.8M	605s	2452.96M
	Sum	408	1001s	3814.77M	97s	346.38M	1s	2.4M	895s	3446.88M
List+QG	1	32	7s	24.96M	0s	1.39M	0s	0.03M	6s	22.25M
	2	198	305s	1022.16M	24s	67.43M	1s	0.92M	277s	944.87M
	3	40	7s	27.75M	0s	0.77M	0s	0.1M	6s	25.46M
	4	136	605s	2453.57M	6s	20.78M	0s	0.56M	597s	2426.29M
	Sum	411	924s	3528.9M	31s	90.37M	1s	1.6M	886s	3418.88M
Set	1	30	6s	23.67M	0s	0.92M	0s	0.02M	5s	21.41M
	2	193	298s	1030.77M	19s	75.98M	1s	1.04M	275s	944.86M
	3	38	7s	27.95M	0s	0.81M	0s	0.15M	6s	25.46M
	4	133	658s	2680.7M	62s	239.7M	0s	0.7M	593s	2434.0M
	Sum	399	969s	3763.57M	82s	317.41M	1s	1.91M	880s	3425.74M
Set+QG	1	30	6s	23.65M	0s	0.97M	0s	0.02M	5s	21.41M
	2	193	295s	1016.75M	17s	62.5M	1s	0.87M	275s	944.59M
	3	38	7s	27.73M	0s	0.77M	0s	0.08M	6s	25.46M
	4	133	601s	2450.14M	5s	16.2M	0s	0.53M	594s	2427.32M
	Sum	399	910s	3518.73M	22s	80.44M	1s	1.5M	880s	3418.78M