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,
trenti.dylan@spes.uniud.it
[2] University of Udine, Udine, Italy

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 [17], and has been the subject of significant research efforts in formal verification [7,12,19,22,29].

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)

A. Biere et al. (Eds.): IJCAR 2026, LNAI 16689, pp. 286–303, 2026.
https://doi.org/10.1007/978-3-032-32592-1_17

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 [24] 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 [23], 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 corresponding 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 Sect. 6 presents the experimental evaluation. Section 7 concludes with future work. Additional material (example, contract-generation algorithm, and detailed experimental data) is available in an extended paper [13].

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 [20,28,30]. 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 [7,15,29], a well known design language for SRA systems. Tasche et al. [29] propose a deductive verification framework where VerCors [5] is used as a backend to verify SystemC programs. The works in [7,15] present fully algorithmic approaches based on explicit-state model checking and on the ESST extension with software model checking techniques [14]. 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 [1,26,27,31]. 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 [18].

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 [21], 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 Fig. 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 `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.

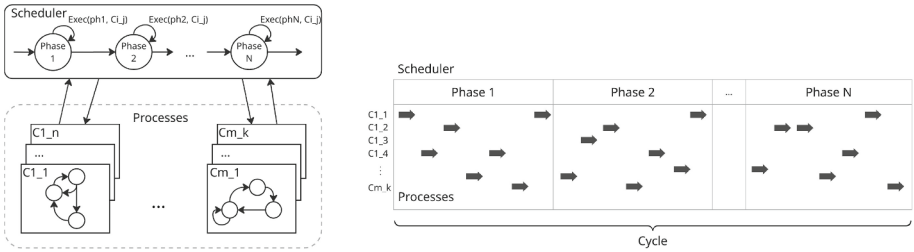

Fig. 1. SRA: architectural view (left) and dynamic view (right).

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

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

Statements

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

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 (Sect. 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 Fig. 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 Fig. 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:*

- $S_0(\texttt{phase}) = l_0$ *(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*
- $S_n(\texttt{phase}) = l_f$ *(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 the extended paper [13].

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* $S(\texttt{phase}) = l_f$*, 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 $\texttt{phase} = l_f$), 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 `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 the extended paper [13].

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 [23]. 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 Sect. 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 Sect. 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 the extended paper [13]. 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 [29] and from industrial case studies obtained within a collaboration with the Italian railway infrastructure operator RFI[1]. 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 (Sect. 5.2).

[1] 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	2 s	2.72M	0 s	1.55M	0 s	0.16M	1 s	1.65M
RPS	List	994	4233 s	9436.56M	350 s	724.29M	3465 s	7034.48M	396 s	1617.89M
	List+QG	998	925 s	2529.53M	403 s	671.75M	60 s	188.41M	442 s	1617.18M
	Set	963	729 s	2154.06M	345 s	675.18M	7 s	21.37M	356 s	1401.94M
	Set+QG	963	620 s	1851.96M	234 s	388.14M	5 s	13.7M	362 s	1401.69M
SCL	List	408	1001 s	3814.77M	97 s	346.38M	1 s	2.4M	895 s	3446.88M
	List+QG	411	924 s	3528.9M	31 s	90.37M	1 s	1.6M	886 s	3418.88M
	Set	399	969 s	3763.57M	82 s	317.41M	1 s	1.91M	880 s	3425.74M
	Set+QG	399	910 s	3518.73M	22 s	80.44M	1 s	1.5M	880 s	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	16708 s	42544M	755 s	1899M	6195 s	15508M	514 s	1195M	12888 s	33769M	797 s	1908M	4096 s	10064M	497 s	1187M
	Max	2224 s	5580M	116 s	277M	370 s	863M	72 s	188M	1851s	4866M	92 s	226M	238 s	555M	59 s	134M
SCL	1	13 s	42M	11 s	32M	12 s	39M	13 s	38M	12 s	41M	12 s	36M	11 s	38M	13 s	35M

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 of assertions, and the performance measured in terms of time elapsed and of Dafny RC[2]. 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 the extended paper [13].

[2] 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.

Generalized Robot Controller System. The Robot Controller System (RCS) is a SystemC case study proposed in [29], 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 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 18 s and used 63.26M RC. The full details of the example can be found in the extended paper [13].

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 60 s (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 4 s (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. [27,32], possibly extended with abstraction strategies similar to [16,25]. 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.

Disclosure of Interest. The authors have no competing interests to declare.

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)
2. Amendola, A., et al.: A model-based approach to the design, verification and deployment of railway interlocking system. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12478, pp. 240–254. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61467-6_16
3. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. **22**(6), 307–309 (1986)
4. Bloem, R., et al.: Decidability in parameterized verification. SIGACT News **47**(2), 53–64 (2016)
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, 12–16 May 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
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: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 90–107. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22306-8_7
8. Cavada, R., et al.: 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., et al.: 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., et al.: 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, 26–28 November 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
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., Griggio, A., Lidström, C., Redondi, G., Trenti, D.: Verification of configurable SRA systems: extended version (2026). https://doi.org/10.48550/arXiv.2605.21385. Available at https://arxiv.org/abs/2605.21385
14. 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
15. Cimatti, A., Narasamdya, I., Roveri, M.: Software model checking systemc. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **32**(5), 774–787 (2013)

16. Clarke, E., Talupur, M., Veith, H.: Proving Ptolemy right: the environment abstraction framework for model checking concurrent systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 33–47. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_4
17. Committee, S.S.: IEEE standard for standard systemic 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
18. 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, 6–10 November 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
19. Große, D., Drechsler, R.: Formal verification of LTL formulas for systemc designs. In: ISCAS (5), pp. 245–248. IEEE (2003)
20. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 113–131. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68863-1_8
21. Hawblitzel, C., et al.: Ironfleet: proving safety and liveness of practical distributed systems. Commun. ACM **60**(7), 83–92 (2017)
22. Herber, P., Hünnemeyer, B.: Formal verification of systemic designs using the BLAST software model checker. In: ACES-MB@MoDELS. CEUR Workshop Proceedings, vol. 1250, pp. 44–53. CEUR-WS.org (2014)
23. 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
24. Li, J., Li, Q., Li, J.: The w-model for testing software product lines. In: ISCSCT (1), 690–693. IEEE Comput. Soc. (2008)
25. 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, 14–17 July 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
26. 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, 21–24 July 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
27. 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)
28. Scaletta, M., Hähnle, R., Steinhöfel, D., Bubel, R.: Delta-based verification of software product families. In: GPCE, pp. 69–82. ACM (2021)

29. Tasche, P., Monti, R.E., Drerup, S.E., Blohm, P., Herber, P., Huisman, M.: Deductive verification of parameterized embedded systems modeled in systemic. In: Dimitrova, R., Lahav, O., Wolff, S. (eds.) Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, 15–16 January 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
30. Thüm, T., Schaefer, I., Hentschel, M., Apel, S.: Family-based deductive verification of software product lines. In: GPCE, pp. 11–20. ACM (2012)
31. 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, 24–27 July 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
32. 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, 11–13 July 2022, pp. 485–501. USENIX Association (2022). https://www.usenix.org/conference/osdi22/presentation/yao