

Verification of Configurable SRA Systems

Alessandro Cimatti, Alberto Griggio, Christian Lidström, **Gianluca Redondi**, Dylan Trenti

Fondazione Bruno Kessler, Trento, Italy

IJCAR 2026

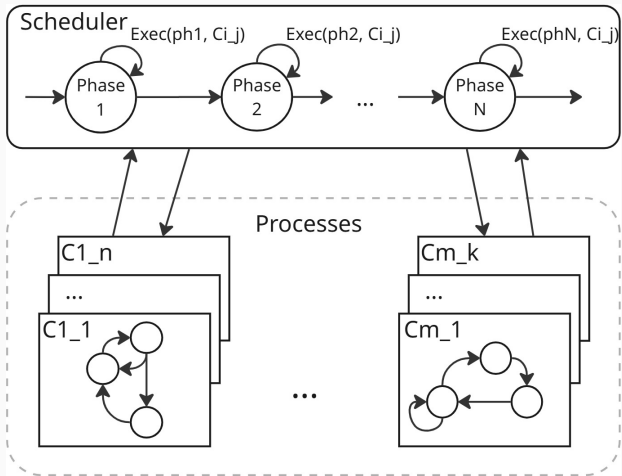
Motivation: Configurable Railway Software

- Ongoing **FBK–RFI** collaboration: build safety-critical software for railway applications (interlocking logics, ...).
- The software is **generic** (*parameterized*): the *same* logic ships to many stations — different numbers of tracks, sensors, controllers.
- The software is organized in **classes**: class instances are processes governed by a **scheduler** that orchestrates their execution (similar to systemC designs).
- We describe a **framework** for the definition and formal verification of these systems.

The Parameterized Verification Problem

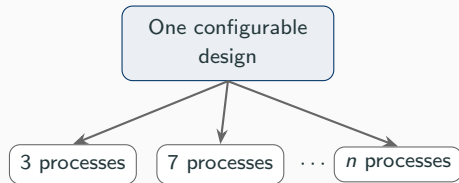
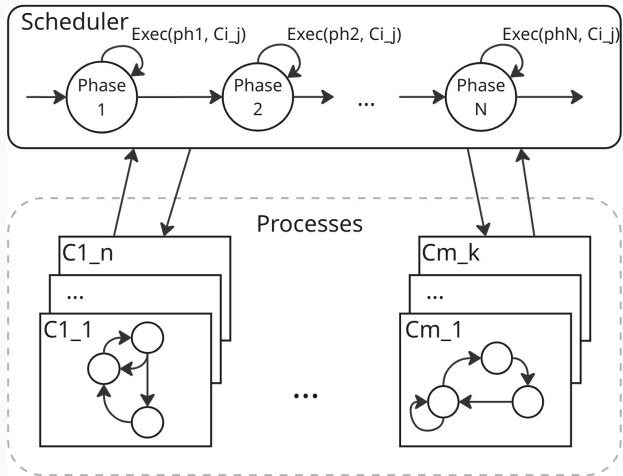
- **Goal — parameterized verification:** prove safety **once**, for every legal configuration, instead of re-verifying each deployed product.
- **Challenge:** unbounded state space, quantified specifications, domain-specific scheduling — the problem is **undecidable** in general.
- **Our approach:** a **compositional, contract-based deductive** method,
 - implemented in Dafny,
 - validated on real industrial railway logics.

Configurable SRA Systems at a Glance



An **SRA**: a **scheduler** orchestrating a *fixed* set of process instances.

Configurable SRA Systems at a Glance



A **configurable** SRA = a *family* of such systems
— same design, *different number of processes*.

An **SRA**: a **scheduler** orchestrating a *fixed* set of process instances.

Configurable SRA Model

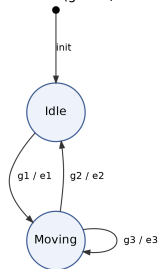
cSRA = a finite set of **Classes** + a **Scheduler** + constraints Γ .

A Class is an *EFSM template*: location, fields (Boolean, enum, integers, or constant *set-of-objects*), and transitions (guard + effect).

Related instances via **sets**:

- a transition can read / update an *unbounded* number of other instances;
- **specs are quantified**: $\forall x \in S : b \quad / \quad \forall x \in S \{x.f := e\}$.

A class instance: an EFSM (guard / effect on each transition)



Each transition carries a **guard** and an **effect**.

A Class, by Example

```
class Controller {  
  var location  : CtrlLoc    // {Idle, Moving}  
  var direction : Direction  
  set leftSensors  : Set<Sensor>  
  set rightSensors : Set<Sensor>  
  set allSensors   : Set<Sensor>  
  
  transition actRight =  
    ( Idle,  
      (exists s in leftSensors  : s.loc == NoGo) &&  
      (forall s in rightSensors : s.loc == Go),  
      Moving,  
      { direction := Right;  
        forall s in allSensors {s.processed:=true;} },  
      Act )  
}
```

A **class** = fields + transitions.

A **transition** is a tuple:

- source location (Idle)
- guard — quantified over sensor sets
- target location (Moving)
- effect — updates an unbounded set
- phase tag (Act)

A Class, by Example

```
class Controller {  
  var location  : CtrlLoc    // {Idle, Moving}  
  var direction : Direction  
  set leftSensors  : Set<Sensor>  
  set rightSensors : Set<Sensor>  
  set allSensors   : Set<Sensor>  
  
  transition actRight =  
    ( Idle,  
      (exists s in leftSensors  : s.loc == NoGo) &&  
      (forall s in rightSensors : s.loc == Go),  
      Moving,  
      { direction := Right;  
        forall s in allSensors {s.processed:=true;} },  
      Act )  
}
```

A **class** = fields + transitions.

A **transition** is a tuple:

- source location (Idle)
- guard — quantified over sensor sets
- target location (Moving)
- effect — updates an unbounded set
- phase tag (Act)

A Class, by Example

```
class Controller {  
  var location  : CtrlLoc    // {Idle, Moving}  
  var direction : Direction  
  set leftSensors  : Set<Sensor>  
  set rightSensors : Set<Sensor>  
  set allSensors   : Set<Sensor>  
  
  transition actRight =  
    ( Idle,  
      (exists s in leftSensors  : s.loc == NoGo) &&  
      (forall s in rightSensors : s.loc == Go),  
      Moving,  
      { direction := Right;  
        forall s in allSensors {s.processed:=true;} },  
      Act )  
}
```

A **class** = fields + transitions.

A **transition** is a tuple:

- source location (Idle)
- guard — quantified over sensor sets
- target location (Moving)
- effect — updates an unbounded set
- phase tag (Act)

A Class, by Example

```
class Controller {  
  var location  : CtrlLoc    // {Idle, Moving}  
  var direction : Direction  
  set leftSensors  : Set<Sensor>  
  set rightSensors : Set<Sensor>  
  set allSensors   : Set<Sensor>  
  
  transition actRight =  
    ( Idle,  
      (exists s in leftSensors  : s.loc == NoGo) &&  
      (forall s in rightSensors : s.loc == Go),  
      Moving,  
      { direction := Right;  
        forall s in allSensors {s.processed:=true;} },  
      Act )  
}
```

A **class** = fields + transitions.

A **transition** is a tuple:

- source location (Idle)
- guard — quantified over sensor sets
- target location (Moving)
- effect — updates an unbounded set
- phase tag (Act)

A Class, by Example

```
class Controller {  
  var location  : CtrlLoc    // {Idle, Moving}  
  var direction : Direction  
  set leftSensors  : Set<Sensor>  
  set rightSensors : Set<Sensor>  
  set allSensors   : Set<Sensor>  
  
  transition actRight =  
    ( Idle,  
      (exists s in leftSensors  : s.loc == NoGo) &&  
      (forall s in rightSensors : s.loc == Go),  
      Moving,  
      { direction := Right;  
        forall s in allSensors {s.processed:=true;} },  
      Act )  
}
```

A **class** = fields + transitions.

A **transition** is a tuple:

- source location (Idle)
- guard — quantified over sensor sets
- target location (Moving)
- effect — updates an unbounded set
- phase tag (Act)

A Class, by Example

```
class Controller {  
  var location  : CtrlLoc    // {Idle, Moving}  
  var direction : Direction  
  set leftSensors  : Set<Sensor>  
  set rightSensors : Set<Sensor>  
  set allSensors   : Set<Sensor>  
  
  transition actRight =  
    ( Idle,  
      (exists s in leftSensors : s.loc == NoGo) &&  
      (forall s in rightSensors : s.loc == Go),  
      Moving,  
      { direction := Right;  
        forall s in allSensors {s.processed:=true;} },  
      Act )  
}
```

A **class** = fields + transitions.

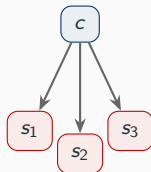
A **transition** is a tuple:

- source location (Idle)
- guard — quantified over sensor sets
- target location (Moving)
- effect — updates an unbounded set
- phase tag (Act)

Two Configurations of One Design

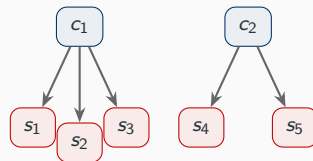
Same Controller/Sensor classes, instantiated differently:

$\mathcal{C}_1$: one controller, three sensors



$c.\text{left} = \{s_1\}, c.\text{right} = \{s_2, s_3\}$

$\mathcal{C}_2$: two controllers, disjoint sensor sets



$c_1.\text{all} = \{s_1, s_2, s_3\}, c_2.\text{all} = \{s_4, s_5\}$

A **configuration** fixes the instances of each class and how their set fields connect them.

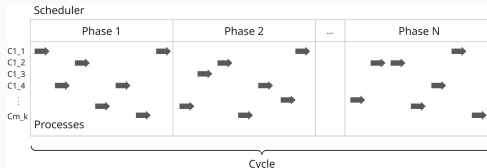
Scheduler and Cycle

The scheduler is itself an **EFSM**: its locations are the **phases** of one cycle.

Two kinds of transitions:

- **self-loop** — the scheduler calls each instance's **exec(p)** (in **arbitrary order**): it runs an enabled phase- p transition, or stutters;
- **phase-change** — the scheduler advances the phase; instances do not move.

A *cycle* = path from initial to final phase; a *run* = sequence of cycles.



One cycle, phase by phase.

Constraints and the Verification Problem

Configuration constraints Γ

First-order formulae over the *immutable* symbols; a configuration is a **model** of Γ .

$$\forall c : \text{Controller}. c.\text{leftSensors} \cap c.\text{rightSensors} = \emptyset$$

Example safety property φ — a left obstacle forbids moving left

$$\forall c. (\exists s \in c.\text{leftSensors}. s.\text{obstacle}) \Rightarrow c.\text{direction} \neq \text{Left}$$

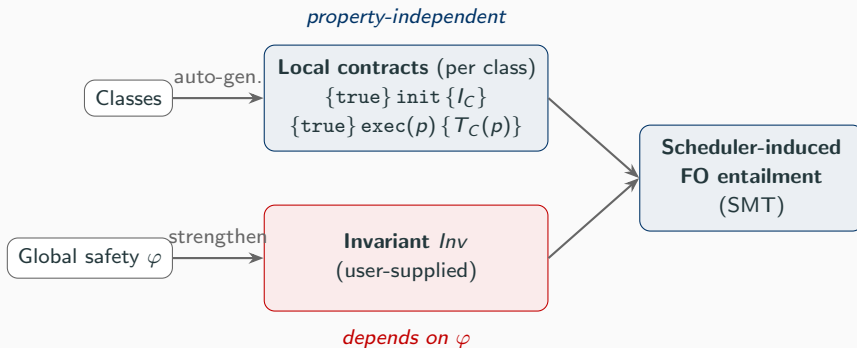
Parameterized Verification Problem

Given $\mathcal{S} = (\{C_1, \dots, C_k\}, \text{Sched}, \Gamma)$ and a global safety property φ :

$$\forall \mathcal{C} \models \Gamma, \text{ in every end-of-cycle state (phase} = l_f) : \quad \mathcal{C}, s \models \varphi.$$

Key Idea

Do not model the scheduler explicitly. Reduce *global* safety to *per-class local* contracts, combined by first-order entailment.



Proof Obligations and Soundness

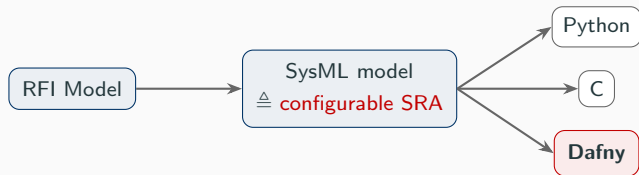
Families of **single-step** entailment checks (all assume Γ), using the auto-generated contracts I_C , $T_C(p)$, K_C :

- **Initialization:** $\bigwedge_C I_C \Rightarrow Inv$
- **Self-loop step (the key one):** a *single* exec preserves $Inv \Rightarrow$ the whole self-loop does, in any order
- **Phase change:** scheduler advances phase $\Rightarrow Inv$ preserved
- **Safety:** $Inv \wedge \text{phase} = l_f \Rightarrow \varphi$

Theorem (Compositional Soundness)

If all local contracts are valid and these checks succeed for some Inv , then φ holds at the end of every cycle, in every run, for *all* configurations $\mathcal{C} \models \Gamma$.

Built on **Dafny** — mature; natively combines OO + quantified FOL + imperative code.



Dafny checks each generated contract against the method *body*.

Two Encodings: Set vs. List in Dafny

The toolchain emits the Dafny model in **two encodings** of the same system:

Set — 1:1 with the paper's semantics.

Higher abstraction, order-independent,
faster to verify.

```
// guard: a pure function, no loop
function g() : bool reads ...
  { forall x :: x in S ==> x.f == v }

// effect: one quantified assignment
method e() modifies ...
  { forall x | x in S { x.f := e; } }
```

List — closer to the **deployed C code**.

Closer to the production artifact $\Rightarrow$
underpins eventual translation validation.

```
// guard: method, for-loop + early exit
method g() returns (b:bool) ...
  ensures forall x :: x in S ==> x.f == v
  { b:=true;
    for i:=0 to |S|
      invariant b<==>forall j::0<=j<i==>S[j].f==v
      { b := b && (S[i].f==v); if !b {break;} } }
```

For **both** encodings, contracts are **auto-generated**.

Evaluation: an Industrial Case Study

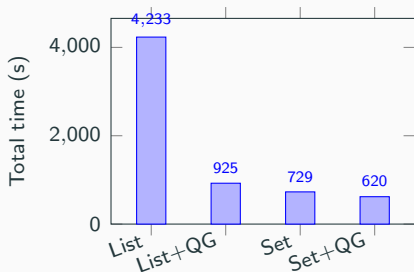
Among the applications, we applied the framework to the **RPS** — a **real industrial** railway protection system, ~ 14 kLoC.

- It authorizes track access for maintenance workers based on the state of the network.
- Designed by **RFI signaling engineers**.

Setup: four Dafny codebases (Set / List, each $\times$ QG); Dafny 4.11 on a cluster. All **auto-generated contracts proved**.

QG (quantifier grounding): when Γ bounds a set, quantifiers are unfolded to **quantifier-free** form — easier for SMT.

Evaluation: Local Contracts



RPS: all local contracts, total time.

All **auto-generated** contracts proved — no manual annotation.

- **Set** $\gg$ **List**: the set model hides list bookkeeping.
- **+QG** a large further win.

This step makes the compositional rule usable: property-independent, done *once* per system.

Evaluation: Safety Properties

We verify **25 RPS safety properties** (authorization is never granted under unsafe inputs).

Verification time per property — Set+QG (over the 25 properties)

Total 497 s

Min 6 s

Average ≈ 20 s

Max 59 s

- **21 of 25** properties already inductive; only 4 needed a strengthening invariant.
- QG correctness *itself proved*: 516 lemmas in 60 s.

Contributions

- Deductive proof rule for parameterized verification of conf SRA;
- Dafny encoding with auto-contract generation; validated on industrial railway logics.

Ongoing work

- **Model checking** in the loop, for automatic invariant discovery.
- **Cutoff theorems** for particular cases.
- **Temporally extended** specifications: beyond single-cycle safety.
- Already exposing **bugs**: model checking found genuine counterexamples.

Backup: The Crux — Self-Loop Preservation

A self-loop at phase p runs $\text{exec}(p)$ on every instance, in **arbitrary order**.

Reduction: prove that *one* instance's exec step preserves $\text{Inv} \Rightarrow$ by induction over the (arbitrarily ordered) instances, the *whole* self-loop does.

Difficulty: the condition that enabled the loop might be broken inside — so the single step cannot assume it still holds.

Fix: each instance relies only on a **local fact that stays true** throughout the loop (e.g. “it has not executed yet”).

Self-loop at phase p — instances run in **arbitrary order**:



Each step preserves $\text{Inv} \Rightarrow$ the whole self-loop preserves Inv , in any order.

Backup: Automatic Contract Generation

The toolchain **auto-generates** (in Dafny) a postcondition $T_C(p)$ summarizing $\text{exec}(p)$ — *no manual annotation*.

Transition (input):

```
transition actRight = (Idle,  
  (exists s in leftS : s.loc==NoGo) &&  
  (forall s in rightS: s.loc==Go),  
  Moving, {direction:=Right;  
    forall s in allS {s.processed:=true;}}, Act)
```

Generated $T_{\text{Controller}}(\text{Act})$:

```
( old(loc = Idle  $\wedge$   $G_{\text{actRight}}$ )  
   $\wedge$  loc = Moving  $\wedge$  direction = Right  
   $\wedge$   $\forall s:\text{Sensor}. s.\text{processed} =$   
    (if  $s \in \text{allSensors}$  then true else old( $s.\text{processed}$ ))  
   $\wedge$  framing )  $\vee \dots \vee \text{Stutter}$ 
```

Symbolic execution of effects $\rightarrow$ *disjunction* of (guard $\wedge$ effect $\wedge$ framing) over transitions + stutter.

framing: untouched fields keep their old value.

Backup: Modeling Language — Full Grammar

Types

$$\begin{aligned}\sigma &::= \text{Int} \mid \text{Bool} \mid \text{Enum} \\ \tau &::= \sigma \mid C \mid \text{Timer} \mid \text{Event} \mid \text{Set}\langle C \rangle\end{aligned}$$

Expressions

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

Statements

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

Arithmetic, Booleans, field access, if are *standard*; the **key ingredients** are the constructs over *set-valued* fields — quantified expressions and the quantified assignment (update a whole set).

Backup: Quantifier Grounding

When Γ bounds a set field ($|S| \leq k$), every quantifier over S can be **unfolded** into a finite conjunction / disjunction over its $\leq k$ elements — at both the level of quantified *statements* (method bodies) and quantified *contracts*.

Correctness is **proved, not assumed:** for every specification p , auto-generated lemmas establish $\Gamma \models p \Leftrightarrow p_{\text{grounded}}$ (under the cardinality constraints of Γ). So safety of the grounded model implies safety of the original.

Result

Quantifier-free verification conditions — much easier for the SMT solver; large speed-ups in practice.

Backup: Local-Contract Verification (Table 1)

Sys.	Version	# Ass.	Tot. time	Tot. RC	Exec. time	Exec. RC	Eff. time	Eff. RC
RCS	Set	105	2s	2.72M	0s	1.55M	1s	1.65M
RPS	List	994	4233s	9436.56M	350s	724.29M	396s	1617.89M
RPS	List+QG	998	925s	2529.53M	403s	671.75M	442s	1617.18M
RPS	Set	963	729s	2154.06M	345s	675.18M	356s	1401.94M
RPS	Set+QG	963	620s	1851.96M	234s	388.14M	362s	1401.69M
SCL	List	408	1001s	3814.77M	97s	346.38M	895s	3446.88M
SCL	List+QG	411	924s	3528.90M	31s	90.37M	886s	3418.88M
SCL	Set	399	969s	3763.57M	82s	317.41M	880s	3425.74M
SCL	Set+QG	399	910s	3518.73M	22s	80.44M	880s	3418.78M

RC = Dafny resource count. The set formalization contains no guard methods.

Backup: Safety-Property Verification (Table 2)

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

Trans. = verifying over each transition; **Exec.** = verifying over the exec method. Verifying over exec is consistently faster.