Verification of Configurable SRA Systems

Deductive and symbolic techniques for parameterized railway safety logic

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

Fondazione Bruno Kessler, Trento, Italy

WG2.3 Meeting, Southampton, September 2026

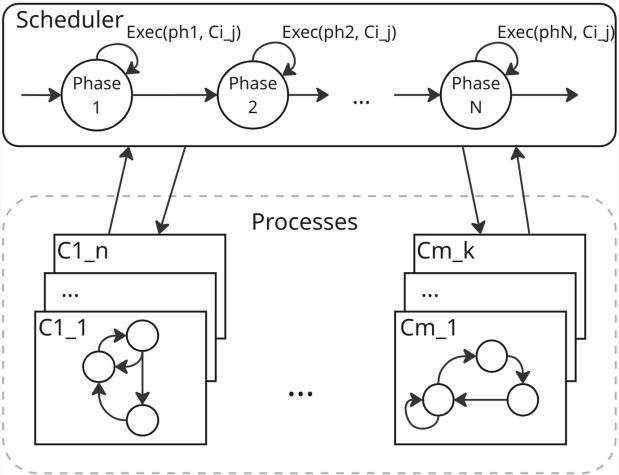
From Relays to Configurable Software

- FBK and RFI (Rete Ferroviaria Italiana) collaborate on next-generation railway interlocking and protection software, moving from legacy relay-based systems to computer-based 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.

Proving it once, for every configuration

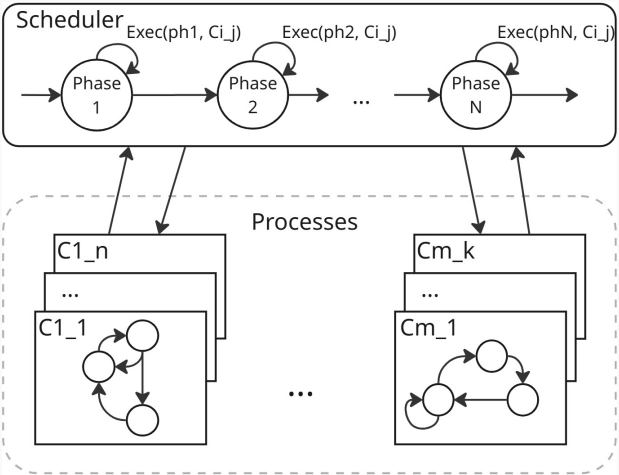
- **Goal — parameterized verification:** prove safety once, for every legal configuration, instead of re-verifying each deployed product.
- **Challenge:** unbounded state space, quantified specifications, domain-specific scheduling — the problem is undecidable in general.
- **Our approach:** a compositional,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

Configurable SRA Systems at a Glance

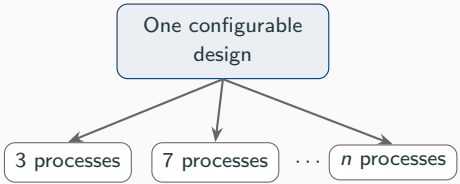


An SRA: a scheduler orchestrating a *fixed* set of process instances.

Configurable SRA Systems at a Glance



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



A configurable SRA = a *family* of such systems

— same design, *different number of processes*.

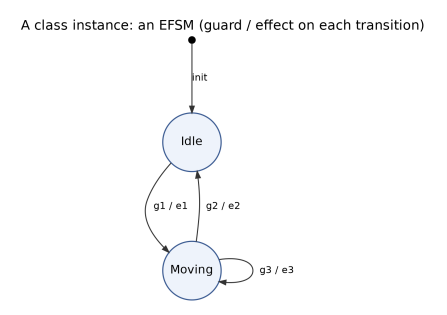
Configurable SRA Model

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: $\forall x \in S : b$ / $\forall x \in S\ \{x.f := e\}$.



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.

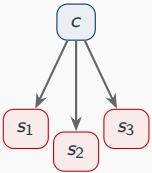
A **transition** is a tuple:

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

Two Configurations of One Design

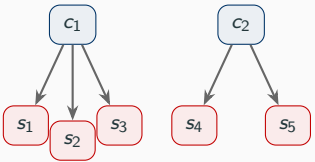
Same `Controller`/`Sensor` classes, instantiated differently:

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



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

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



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

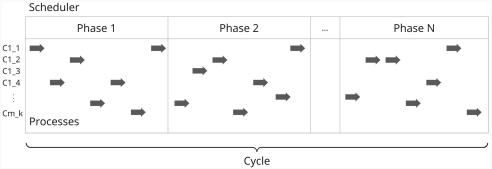
A configuration fixes the instances of each class and how their set fields connect them.

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.



One cycle, phase by phase.

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

Constraints and the Verification Problem

Configuration constraints Γ

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

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

Example safety property φ — a left obstacle forbids moving left

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

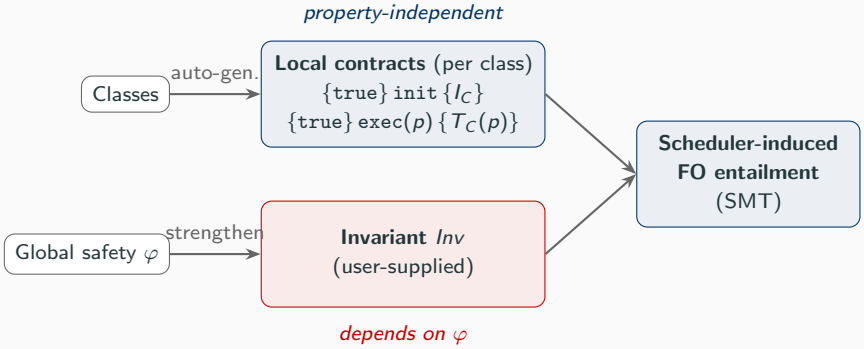
Parameterized Verification Problem

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

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

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:** $\bigwedge_C I_C \Rightarrow \mathit{Inv}$
- **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:** $\mathit{Inv} \wedge \texttt{phase} = l_f \Rightarrow \varphi$

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 $\mathcal{C} \models \Gamma$.

Self-Loop Preservation

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

Reduction: prove that *one* instance's exec step preserves $Inv \Rightarrow$ 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.

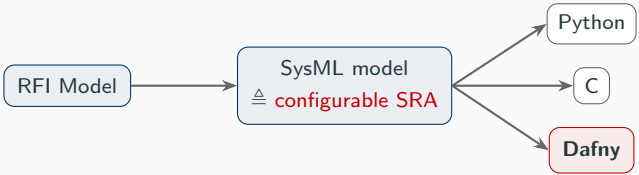
Fix: each instance relies only on a local fact that stays true throughout the loop (e.g. "it has not executed yet").

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

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

Toolchain

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 above 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 ⇒ 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.

Automatic Contract Generation

The toolchain auto-generates (in Dafny) a postcondition $T_C(p)$ summarizing $\mathtt{exec}(p)$ — *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 $T_{\mathtt{Controller}}(\mathtt{Act})$:

$$
\begin{aligned}
&\big(\ \mathtt{old}(\mathtt{loc} = \mathtt{Idle} \wedge G_{\mathtt{actRight}}) \\
&\quad \wedge\ \mathtt{loc} = \mathtt{Moving} \wedge \mathtt{direction} = \mathtt{Right} \\
&\quad \wedge\ \forall s{:}\mathtt{Sensor}.\ s.\mathtt{processed} = \\
&\qquad (\textbf{if } s \in \mathtt{allSensors} \textbf{ then } \mathtt{true} \textbf{ else } \mathtt{old}(s.\mathtt{processed})) \\
&\quad \wedge\ \textit{framing}\ \big)\ \vee\ \cdots\ \vee\ \textit{Stutter}
\end{aligned}
$$

Symbolic execution of effects → *disjunction* of (guard ∧ effect ∧ framing) over transitions + stutter.

framing: untouched fields keep their `old` value.

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.

The RPS Case Study

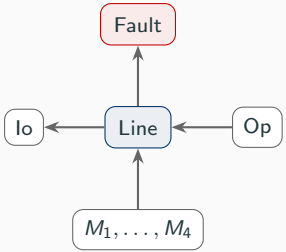
- The RPS (Railway Protection System) is one of RFI's generic applications: a configurable SRA, designed once and instantiated per railway station.
- **Purpose:** manage, in a safety-critical way, the controlled *interruption* of a railway section for maintenance work and its subsequent *release* back into service.
- Real industrial logic, designed by RFI signaling engineers, ∼14 kLoC.

Size		
8 classes	**117** guarded transitions	**164** state variables

RPS Classes

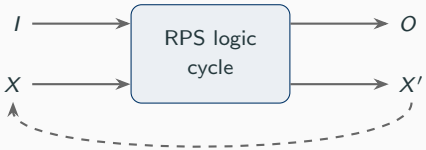
Eight classes, organized around one main class `Line`:

- **Line** — railway sections to be blocked / unblocked;
- $M_1, \ldots, M_4$ — one *management* class per kind of interruption/release procedure;
- **Io** — trackside detection points;
- **Fault** — failures and faults observed on a line section;
- **Op** — the human operator authorized to issue commands.

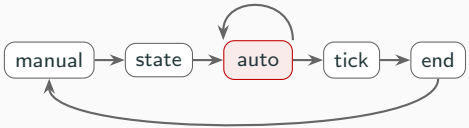


Each `Line` references its `Fault` instance and its two `Io` points; managers, `Op` and `Fault` carry a back-reference to their `Line`.

RPS Scheduler and Cycle



Inputs I + state $X \rightarrow$ outputs O + next state X'; the RPS runs as an unbounded sequence of such cycles.



Five phases per cycle; each runs instances asynchronously, exactly once (auto repeats while automatic commands are pending).

- **manual**: instances react to the manual-command inputs;
- **state**: every instance executes one state-update transition;
- **auto**: instances with a pending automatic command execute one transition each — repeated while new automatic commands are produced;
- **tick**: scheduler-managed timers advance by one;
- **end**: instances emit their outputs; the closing transition refreshes every input, opening the next cycle.

RPS Configuration Constraints

The configuration constraints Γ_{RPS} fix the topology of a deployment — how instances connect across classes — via three kinds of constraints.

- **Cardinality** — e.g. every `Line` connects to exactly two `Io` points and one `Fault` instance:

$$|\ell.io| = 2 \qquad |\ell.fault| = 1$$

- **Mutual-inverse** — the `Line`↔`Fault` references are each other's inverse:

$$\forall\, \ell{:}\mathrm{Line}.\, \forall\, a{:}\mathrm{Fault}.\ a \in \ell.fault \iff \ell \in a.line$$

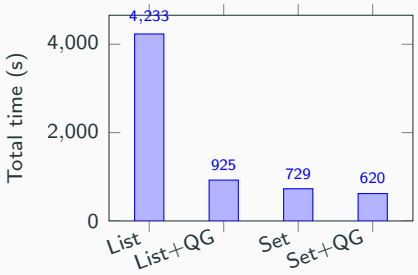
- **Disjointness** — distinct line sections share no detection points or faults:

$$\forall\, \ell, \ell'{:}\mathrm{Line}.\ \ell \neq \ell' \Rightarrow \ell.io \cap \ell'.io = \emptyset$$

Cardinality + mutual-inverse fix the local *shape* of each line section's neighbourhood; disjointness forces these neighbourhoods to be pairwise disjoint.

Evaluation: Local Contracts

Setup: four Dafny codebases (Set / List, each × QG); Dafny 4.11 on a cluster.



RPS: all local contracts, total time.

All auto-generated contracts proved — no manual annotation.

- Set ≫ List: the set model hides list bookkeeping.
- +QG a 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** $\approx$ 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.

Deductive workflow limitations

- If a property holds but is not directly inductive, strengthening the invariant is a manual task.
- Some requirements are inherently multi-cycle (bounded-response): "if A holds now, B must hold within k cycles" — out of reach for the single-step induction of Section 2.
- When a proof does not go through, the engineer cannot tell a genuine violation from a missing auxiliary lemma: Dafny gives no counterexample.

Reducing to Symbolic Model Checking

- For logics whose configuration constraints have the right structure, the parameterized problem reduces to verifying a fixed, finite system — via a cutoff construction.
- A simulation argument shows that verifying universal safety on S is equivalent to verifying a finite cutoff system $red(S)$ built from R — in both directions: safe verdicts lift up, counterexamples lift down.
- *$red(S)$ can now be verified automatically with model checking (e.g. nuXmv).*

From Parameterized to Finite: a Cutoff Reduction

Universal safety property: $\varphi \equiv \forall\, \ell{:}\mathit{Line}.\ \psi(\ell)$.

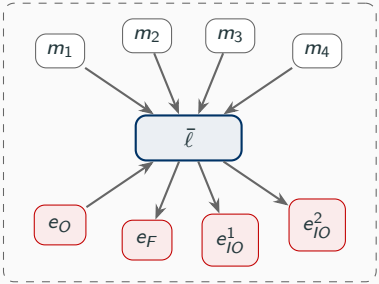
Example ("no command while in fault"):

$\forall \ell{:}\mathit{Line}, \mathtt{phase} = \mathrm{END} \wedge \ell.\mathit{faulty} \rightarrow \neg \ell.\mathit{cmd}$

Prophecy: reason about one arbitrary but fixed line section $\bar{\ell}$ instead of quantifying:

$$\forall \mathcal{C} \models \Gamma.\ \mathcal{S}[\mathcal{C}] \models \varphi \iff \forall \mathcal{C} \models \Gamma.\ \mathcal{S}[\mathcal{C}] \models \psi(\bar{\ell})$$

Grounding: under Γ_{RPS}, $\bar{\ell}$ together with the instances that can influence it forms a fixed 9-instance block R; every other line section is provably non-interfering.



The 9-instance block R: fixed size, regardless of the number of line sections in the deployment.

The Cutoff Theorem

Cutoff system $\mathcal{S}_{\mathrm{cut}}$

Asynchronous composition of the 9 grounded instances of R and the (unchanged) scheduler $\mathit{Sched}_{\mathrm{RPS}}$.

Still infinite-state: timer durations remain symbolic parameters.

Theorem (cutoff soundness & completeness)

$$\mathcal{S}_{\mathrm{cut}} \models \psi(\bar{\ell}) \iff \forall\, \mathcal{C} \models \Gamma.\ \mathcal{S}[\mathcal{C}] \models \varphi$$

- Proof: a stuttering simulation between $\mathcal{S}[\mathcal{C}]$ and $\mathcal{S}_{\mathrm{cut}}$, resting on two structural lemmas: Γ-closure (no navigation leaves R) and Γ-separation (no outside instance references R).
- Counterexamples of $\mathcal{S}_{\mathrm{cut}}$ lift to counterexamples of the original parameterized system.

Encoding the Cutoff in nuXmv

Classes and instances.

- each class → a nuXmv `MODULE`, parameterized by the references its instances carry;
- each of the 9 instances of R → one `VAR` of the corresponding module type in `main`, parameters bound by Φ_R;
- inside a module: one `VAR` per scalar field, one for the control location; set-valued fields are not stored, accessed only through the parameters;
- guarded transitions of each phase → `DEFINE` predicates on current/next values;
- bounded quantifiers over set fields (unsupported in nuXmv) → expanded into explicit finite conjunctions/disjunctions.

Scheduler.

Realized in `main` via a global `phase` variable and one Boolean flag *executed*(e) per instance. `TRANS` is a disjunction of:

- instance steps: fire an enabled instance's transition for the current phase, set its flag;
- phase advances: enabled once every instance has executed; switch phase, reset the flags (the closing advance also refreshes the inputs non-deterministically).

Size of the generated model

8 class modules **9** instances **117** grounded transitions ∼**7 000** lines of SMV

Results: nuXmv on the RPS Cutoff

Setup

nuXmv with IC3IA, 1 h timeout per property.

- **62 safety properties** verified: 46 safe, 16 unsafe; plus **26 sanity probes** (all correctly refuted).
- Multi-cycle bounded-response properties (out of reach for Dafny) encoded as ordinary invariants via a small *monitor* construction, discharged by IC3IA.
- Of the 46 safe properties, 16 already go end-to-end with *both* nuXmv and Dafny; the rest need a Dafny invariant that nuXmv *synthesizes automatically*.
- Counterexamples $\rightarrow$ post-processed into readable traces, shared with the RFI logic designer: 5 genuine bugs found and fixed, 11 requirement ambiguities clarified.

Dafny vs. Model Checking: a Trade-off

Dafny (deductive)

- General: sound for *any* conf. SRA satisfying the compositional proof rule, no cutoff needed;
- per-property, sometimes needs a manual invariant;
- single-cycle safety only;
- no witnesses on failure.

Model checking (cutoff + nuXmv)

- invariants synthesized automatically (IC3IA);
- concrete counterexamples on unsafe properties;
- reaches multi-cycle properties;
- but the cutoff is a special case: relies on closure/separation lemmas that hold for the RPS's Γ, *not* for every conf. SRA.

The two workflows share the same underlying conf. SRA encoding and are complementary, not competing.

Ongoing and Future Work: Combining Deduction and Model Checking

- Close the loop: extract the inductive invariants $I(\bar{\ell})$ synthesized by nuXmv, lift them to $\forall\, \ell.\, I(\ell)$, and feed them into the Dafny proof.
  - extraction of *candidate* invariants from underapproximations.
- Generalize the cutoff to parametric model checking, for conf. SRA where the closure/separation lemmas do not directly apply:
  - building on our own work on invariant checking for SMT-based systems with quantifiers;
  - combined with abstraction-based approaches when a direct cutoff is not available, e.g. along the lines of the *Environment Abstraction* framework for model checking parameterized concurrent systems.
- Bring multi-cycle properties within reach of the deductive encoding.

Conclusion

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.

Thank you

Backup: RPS Verification Results

ID	T	R	nuXmv	Dafny	ID	T	R	nuXmv	Dafny	ID	T	R	nuXmv	Dafny
0	s	S	1	190	21	s	S	124	?	42	m	S	2876	–
1	s	S	2	204	22	s	S	264	?	43	m	U	2291	–
2	s	S	< 1	211	23	s	S	< 1	241	44	m	U	1297	–
3	s	S	2	209	24	s	S	< 1	280	45	m	U	4459	–
4	s	S	15	?	25	s	S	< 1	255	46	m	U	3189	–
5	s	S	19	?	26	s	S	< 1	261	47	m	S	9	–
6	s	S	93	?	27	s	S	< 1	246	48	m	U	611	–
7	s	U	284	?	28	s	S	< 1	241	49	m	U	2117	–
8	s	U	861	?	29	s	S	61	?	50	m	S	1071	–
9	s	S	13	?	30	s	S	< 1	223	51	m	S	1479	–
10	s	S	62	?	31	s	S	< 1	234	52	m	S	958	–
11	s	S	145	?	32	s	S	< 1	231	53	m	S	561	–
12	s	U	449	?	33	s	S	< 1	233	54	m	S	66	–
13	s	U	263	?	34	s	S	< 1	234	55	m	S	454	–
14	s	S	19	?	35	s	S	< 1	240	56	m	S	12	–
15	s	S	32	?	36	s	S	22	?	57	m	S	38	–
16	s	S	127	?	37	m	U	1900	–	58	m	S	26	–
17	s	U	502	?	38	m	U	3038	–	59	m	S	13	–
18	s	S	238	?	39	m	U	2849	–	60	m	S	20	–
19	s	S	147	?	40	m	U	2381	–	61	m	S	15	–
20	s	S	58	?	41	m	U	2136	–					

T: single-cycle (s) / multi-cycle (m); R: safe (S) / unsafe (U); safe , genuine bug , requirement ambiguity .

Backup: From LTL to Invariants — the Monitor Construction

Multi-cycle (bounded-response) properties relate two end-of-cycle states an unbounded number of transitions apart — their natural formulation uses LTL until:

$$G\big((\texttt{phase}{=}\textsc{end} \wedge A) \rightarrow X(\neg(\texttt{phase}{=}\textsc{end})\, U\, (\texttt{phase}{=}\textsc{end} \wedge B))\big)$$

"whenever A holds at an end-of-cycle state, B holds at the next one" — the general "k-th subsequent cycle" case is analogous.

These are bounded-response, hence *safety*, properties: every violation has a finite witness. nuXmv's native liveness engines did not converge on the LTL form, so instead we verify them as invariants via a small ad-hoc *monitor*:

- a counter $c \in \{0, 1, 2\}$ and a frozen variable $\widehat{A}$, initially $c = 0$;
- the monitor non-deterministically picks a witness end-of-cycle state, latches $\widehat{A} := A$ and sets $c := 1$; on the *next* end-of-cycle transition, it sets $c := 2$.

Invariant to verify

$$(c = 2 \wedge \texttt{phase} = \textsc{end}) \rightarrow (\widehat{A} \rightarrow B)$$

Backup: Modeling Language — Full Grammar

Types

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

Expressions

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

Statements

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

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: 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.