



UNIVERSITY
OF TRENTO



SMT-based Verification of Parameterized Systems

Ph.D. Candidate: Gianluca Redondi

Advisor: Alessandro Cimatti

Co-advisor: Alberto Griggio

July 18th, 2024

Content of the talk

Introduction and Problem Statement

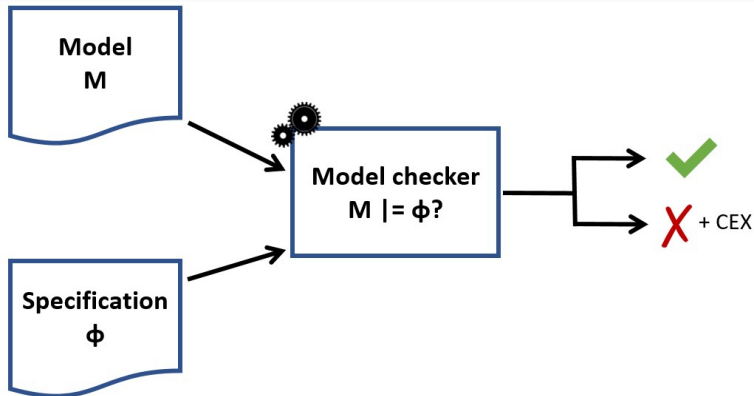
Algorithms Developed

Application in the verification of interlocking logic

Conclusions and Future Directions

Introduction and Problem Statement

Model Checking: automated verification of systems



SMT Solvers



- Satisfiability of first-order formulae with respect to various underlying theories (EUF, LIA, Arrays, etc.).
- Support for interpolation (MathSAT).
- Handling quantifiers with instantiation-based approaches (Z3).

Verification Modulo Theories

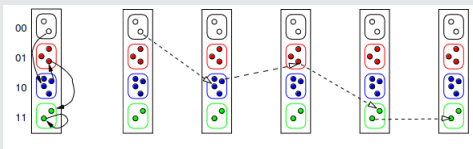


- Integration of SMT solvers within the model checking process to handle complex systems.
- Various algorithms: Bounded Model Checking (BMC), k-induction, Interpolation-based methods, IC3.

Background: IC3 and IC3IA

- **IC3** is an algorithm for checking invariants when the system and the property are defined using Boolean logic.
 - IC3 incrementally refines over-approximations of the reachable states (called frames) until a violation or a proof is found.
 - It is a complete algorithm, thus it is guaranteed to terminate.
-
- **IC3IA** is a generalization of IC3 for quantifier-free SMT theories.
 - Combines predicate abstraction and interpolation techniques.
 - For a fixed set of predicates, it is complete, but it may diverge during refinement.

Abstraction Techniques

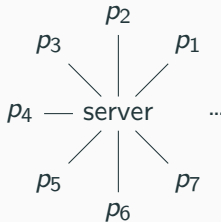


Process:

1. Create an abstract model of the system.
2. Perform model checking on the abstract model.
3. If a counterexample is found, check its validity in the concrete system.
4. Refine the abstraction if the counterexample is spurious, and repeat.

Parameterized Systems

Parameterized systems are systems in which the number of certain components is unbounded. An example is a client-server protocol with an unknown number of clients.



Typical properties of parameterized systems, such as **mutual exclusion**, should hold independently of the number of components.

A Motivating Example: Interlocking Logic



The design of an *interlocking* logic for railways is often done in a parameterized setting. Can we prove that it is safe in all possible stations?

Formalization of Parameterized Systems

Transition systems over an SMT theory:

$$S = (X, I(X), T(X, X')),$$

where:

- State variables are functions from an uninterpreted sort to some interpreted sort (such as integers).
- First-order quantifiers over uninterpreted sorts appear in the formulae defining the systems.

For example, a possible specification could be:

$$\forall i. \text{state}(i) = \text{idle} \wedge \text{ticket}(i) = 0 \wedge \text{next_ticket} = 1 \wedge \text{to_serve} = 1$$

Problem Statement

Invariant Checking Problem

Given S and ϕ , does every reachable state of the system satisfy ϕ ?

Main goal of the thesis:

Solving the invariant checking problem when S and ϕ are defined with a combination of quantifiers and SMT.

Example of property:

$$\forall i, j. (i \neq j \rightarrow \neg(\text{state}(i) = \text{crit} \wedge \text{state}(j) = \text{crit})).$$

The problem is solved if we either:

- find an **inductive invariant** for the system; or
- find a counterexample, i.e., a run leading to a violation of the property.

We developed two techniques that attempt to automatically infer **universally quantified** inductive invariants, or counterexamples in ground instances of the system (i.e., in a finite model of the uninterpreted sort).

Challenges

- For infinite-state systems, the invariant checking problem (even without quantifiers) is undecidable.
- Checking the (un)satisfiability of first-order formulae is undecidable.

However, there are some positive aspects:

- Efficient algorithms exist for model checking systems over quantifier-free theories.
- Incomplete methods can be sufficient to check the (un)satisfiability of first-order formulae.

- **Cut-off Results:** Techniques that identify a minimal instance size covering all behaviors, useful for specific systems.
- **Semi-automatic Methods:** Verification via manual abstraction or user-guided inductive invariants.
- **Symbolic Backward Reachability:** Tools like MCMT and CUBICLE use symbolic algorithms to compute inductive invariants.

Existing technique: UPDR

- **UPDR** is an algorithm for S and ϕ defined in pure first-order logic where state variables are predicates.
- UPDR builds frames using **diagrams**:
 - Given a finite model $\mathcal{M}$ over a signature Σ , compute the existential closure of all the literals that are true in the model.
 - Diagrams are the 'strongest' existential formulae to represent models.
 - UPDR is semi-complete for finding universal invariants over Σ .

Algorithms Developed

First algorithm: UPDR with Implicit Abstraction

- Combines UPDR with *index predicates*, which contain free variables.
- The original system S is abstracted with a system $\tilde{S}$ defined in uninterpreted logic.
- UPDR is used to check the property over $\tilde{S}$.
- We refine the abstraction via interpolation.

Implicit abstraction: $\tilde{S}$ is never computed explicitly!

- The UPDRIA procedure reuses most of the UPDR algorithm.
- Logical formulae are used to encapsulate the simulation relation.
- Diagrams are projected over the index predicates.

Logical encoding

Given $\mathcal{P}(I)$ a set of index predicates, we define:

- $X_{\mathcal{P}(I)}$ is a set of fresh predicates of index sort, one for each element $p \in \mathcal{P}(I)$, and with arity the number of free variables in p .
- Formula for the abstraction:

$$H_{\mathcal{P}}(X_{\mathcal{P}}, X) = \forall I. \left(\bigwedge_{p(I, X) \in \mathcal{P}} x_p(I) \leftrightarrow p(I, X) \right)$$

- Formula used for RecBlock:

$$\begin{aligned} AbsRelInd(F, \tau, \psi, \mathcal{P}) = & F(X_{\mathcal{P}}) \wedge \psi(X_{\mathcal{P}}) \wedge H_{\mathcal{P}}(X_{\mathcal{P}}, X) \wedge \\ & \tau(X, X') \wedge \neg \psi(X'_{\mathcal{P}}) \wedge H_{\mathcal{P}}(X'_{\mathcal{P}}, X') \end{aligned}$$

UPDRIA - Pseudocode

Algorithm 1: UPDR + IA

```
1 Input:  $S = (X, \iota(X), \tau(X, X')), \phi(X)$ 
2  $\mathcal{P}(I) = \{\text{set of atoms occurring in } \iota, \phi\}$ 
3 if not  $\bar{i} \wedge H_P \models \bar{\phi}$ :
4   return cex # cex in initial state
5  $F_0 = \{i\}$ 
6  $k = 1, F_k = \emptyset$ 
7 while True:
8   while  $F_k \wedge H_P \wedge \neg \bar{\phi}$  is sat:
9     # let  $\sigma$  be an  $A_I^E$  model, and  $\sigma_I$  its restriction on the index sort;
10     $\psi = \text{diag}(\sigma_I)$ 
11    if not  $\text{RecBlock}(\psi, k)$ :
12      # an abstract counterexample  $\pi$  is provided
13      if not  $\text{Concretize}(\pi)$ :
14         $\mathcal{P} = \mathcal{P} \cup \text{Refine}(\pi)$ 
15      else:
16        return unsafe
17    # if no models, add new frame:
18     $k = k + 1, F_k = \emptyset$ 
19    # propagation phase
20    for  $i = 1, \dots, k - 1$ :
21      for each lemma  $\psi \in F_i$ :
22        if  $\text{AbsRelInd}(F_i, \tau, \psi, \mathcal{P})$  is unsat:
23          add  $\psi$  to  $F_{i+1}$ 
24        if  $F_i = F_{i+1}$ :
25          Return safe
```

Algorithm 2: $\text{RecBlock}(\psi, N)$

```
1 if  $N = 0$ 
2   return False
3 While  $\text{AbsRelInd}(F_{i-1}, \tau, \neg\psi, \mathcal{P})$  is sat:
4   # let  $\sigma'$  be an  $A_I^E$  model, and  $\sigma'_I$  its restriction on the index sort;
5    $\psi' = \text{diag}(\sigma'_I)$ 
6   if not  $\text{RecBlock}(\psi', N - 1)$ :
7     return cex
8    $g = \text{Generalize}(\neg\psi, N)$ 
9   add  $g$  to  $F_1, \dots, F_N$ 
10 Return True
```

Soundness

If the algorithm terminates with safe, then $S \models \phi$. If the algorithm terminates with unsafe, then $S \not\models \phi$.

Partial Completeness

Let $\mathcal{L}$ be the set of formulae of the form $\forall \underline{i}. \psi(\underline{i})$, with $\psi(\underline{i})$ a Boolean combination of $\Sigma_I(\underline{i})$ -atoms and elements of $\mathcal{P}(\underline{i})$. If UPDRIA finds an abstract counterexample, then no inductive invariant in $\mathcal{L}$ exists for S and ϕ .

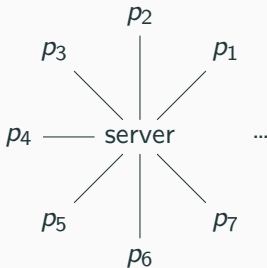
- UPDRIA relies on models for quantified formulas and interpreted theories – solvers struggle to build models.
- Interpolation procedures for quantified SMT are not well-supported.

A Python implementation is available.

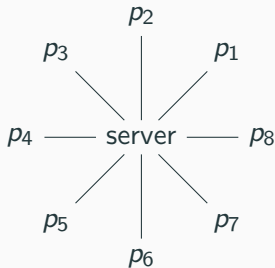
We also developed a second algorithm, following this ‘practical’ intuition in parameterized model checking: *‘Either a counterexample occurs for a small value of the parameter, or the system is safe always for the same reason (at least after a certain threshold value)’*.

A Source of Help: Ground Instances

Parameterized system



Ground instance

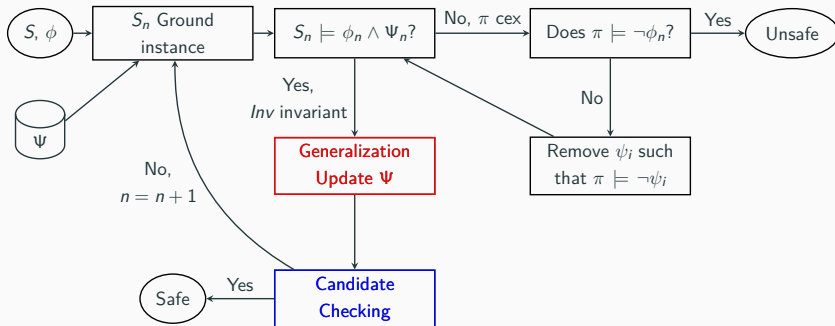


Ground instances are under-approximations of the parameterized system, but they are a useful source of heuristics, and can be model checked efficiently.

General idea: Check $S \models \phi$ in a ground instance, and then generalize the proof.

- After checking a ground instance, the model checker provides an inductive invariant.
- Our procedure attempts to lift this invariant to the parameterized (quantified) case.
- We then check if our generalization is correct or determine if a larger instance needs exploration.

Algorithm Overview



An overview of the algorithm.

Example of Generalization

- From a ground inductive invariant in CNF

$$(\text{state}(p_1) = \text{idle} \vee \text{state}(p_2) = \text{active}) \wedge$$

$$(\text{state}(p_2) = \text{idle} \vee \text{state}(p_1) = \text{active}) \wedge$$

...

- to universally quantified lemmas

$$\forall i_1, i_2. i_1 \neq i_2 \rightarrow (\text{state}(i_1) = \text{idle} \vee \text{state}(i_2) = \text{active}) \wedge$$

...

To enhance generalization, we employed:

- Minimization techniques to derive better invariants from the ground instance.
- Prioritizing the equality theory in the ground model checker before considering interpreted theories.
- Leveraging symmetries within the ground instance.

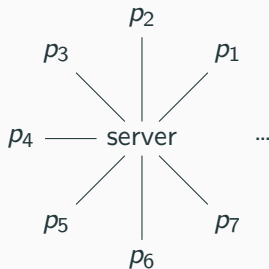
We need to check that $\phi \wedge \Psi$ is an inductive invariant for S .

- Even if the generalization is correct, we might get stuck in proving validity of the verification conditions.
- We are mainly interested in proving unsatisfiability, so incomplete but 'fast' strategies are preferred.

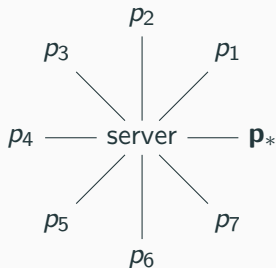
We developed two approaches for checking that the generalization is correct.

Candidate Checking via Parameter Abstraction

Parameterized system



Parameter Abstraction



Simulation given by a quantifier-free system! Still, spurious counterexamples may occur.

- A more standard approach: apply Skolemization to the verification condition and search for a set of ground terms that induce unsatisfiability.
- In the case of partial models, explore the verification conditions of a larger ground instance to obtain new terms or generalize new lemmas.
- This method is implemented on top of MathSAT with a simple instantiation strategy.

A third algorithm: UPDRIA + Lambda

We also implemented a hybrid algorithm that combines the two approaches:

- The algorithm starts with a finite instance exploration and generalizes a candidate inductive invariant as in Lambda.
- Then, UPDRIA runs to either conclude the proof or to 'fix' the invariant.

Table 1: Summary of the characteristics of the tools considered

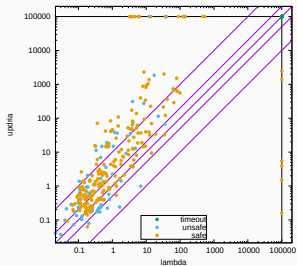
	$\mathcal{T}_E$ only Boolean	$\mathcal{T}_E$ generic	$T(X, X')$ generic	$\forall\exists$ invariants
UPDRIA	✓	✓	✓	-
LAMBDA	✓	✓	✓	-
MCMT	✓	✓	-	-
CUBICLE	✓	✓	-	-
IC3PO	✓	-	✓	✓
MyPyVY	✓	-	✓	✓

- **Mcmt benchmarks:** 238 benchmarks taken from the MCMT distribution and research papers.
- **Cubicle benchmarks:** 42 benchmarks taken from the CUBICLE distribution, with a MCMT translation available.
- **VMT:** 52 benchmarks taken from IC3PO and MYPYVY papers.
- **Trains benchmarks:** 25 benchmarks inspired by train verification problems.

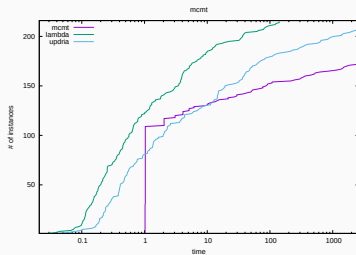
Data Comparison - Overall

	Mcmt (238 tot)	Cubicle (42 tot)	VMT (52 tot)	Trains (25 tot)
UPDRIA	158	19	27	20
UPDRIA+LAMBDA	208	30	29	24
LAMBDA	214 (12)	35 (5)	29	25 (1)
MCMT	172	24	-	-
CUBICLE	-	31 (1)	-	-
IC3PO	-	-	36 (7)	-
MyPyVY	-	-	27	-
VBS	220	36	37	25

Data Comparison - different algorithms



UPDRIA vs LAMBDA



LAMBDA, UPDRIA and MCMT

Application in the verification of interlocking logic

Railways and AIDA

In an ongoing collaboration between FBK and RFI, a tool (AIDA) is being developed for the design of future train stations:

- specs for the logic are given in **pseudo-natural language**;
- code is automatically generated for the functioning of every component of a station;
- many properties should hold **independently of the configuration!**
- a parameterized model can be obtained much earlier in the verification process:



automatic steps are in red

Towards Verification of Interlocking via Dafny

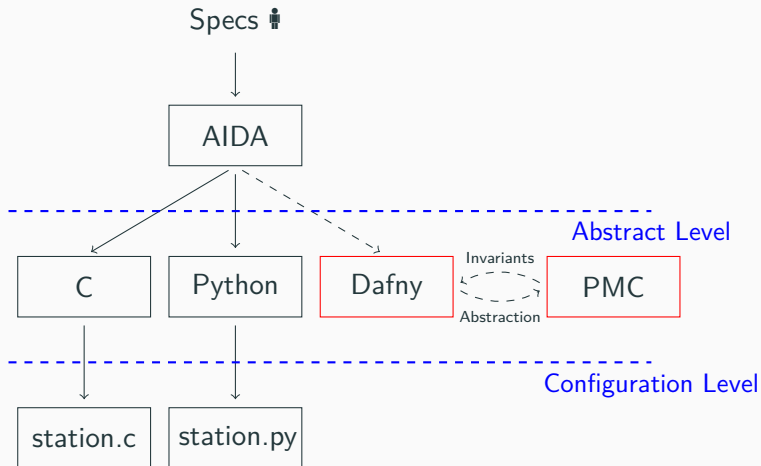
- Dafny is a programming language and verification tool designed for building correct software.
- Generation of Dafny code from AIDA seems like a relatively straightforward task.
- While proficient at checking verification conditions, Dafny struggles with inferring invariants automatically.
- **Our goal** is to explore the integration of Dafny with UPDRIA/Lambda to verify properties of interlocking logic.

We have modeled a simple version of the parameterized logic in Dafny:

- Each component has its own class with attributes mirroring the behavior of the real ones.
- Code was written to mirror the code produced by AIDA in other languages.
- Contracts ensure that the code of each method implements the desired specifications.

- Additional properties (such as ensuring that two trains never collide) were added to the Dafny models.
- Properties were not inductive: through a translation into the VMT language and the algorithms above, an inductive invariant was found and validated with Dafny.

AIDA Workflow



Main problem in verifying the actual logic: it is huge!

- More than 100 components with tens of state variables each.
- Each component is a parameterized system that communicates with others via asynchronous composition (shared variables).
- A lot of redundancy: given a property, how can we identify a set of components and the subset of variables responsible for its safety?

Our idea (sketch):

- We designed an abstraction process that eliminates variables, exploiting the syntactic shape of the logic.
- We can consider the interaction only on a subset of 'relevant' abstract components and model check it.
- We can later use a theorem prover to check if the safety proof is also valid for all other components.

Conclusions and Future Directions

- Improve algorithms to also infer invariants with quantifier alternations.
- Enhance algorithms to handle liveness properties.
- Specialize algorithms for software verification.
- Better implementation of algorithms inside nuXmv.

Current main effort:

- Prove that the parameterized design of the railway logic is safe, using also the techniques developed in the thesis.

Development of algorithms:

- CADE 2021: Universal Invariant Checking of Parametric Systems with Quantifier-free SMT Reasoning
- ATVA 2022: Verification of SMT Systems with Quantifiers
- TOCL 2024: Invariant Checking for SMT-based Systems with Quantifiers

Case studies:

- Dafny@POPL 2024: Towards the verification of a generic interlocking logic: Dafny meets parameterized model checking
- FMCAD 2024: Towards Verification Modulo Theories of asynchronous systems via abstraction refinement

SMT-based verification of parameterized systems:

- We extended the state of the art with two new algorithms.
- We propose a compositional approach for checking properties on a large number of components.
- We conducted a case study combining Dafny and the developed algorithms to prove the safety of a simplified railway interlocking logic.

Thank You

Questions?

UPDRIA - Termination

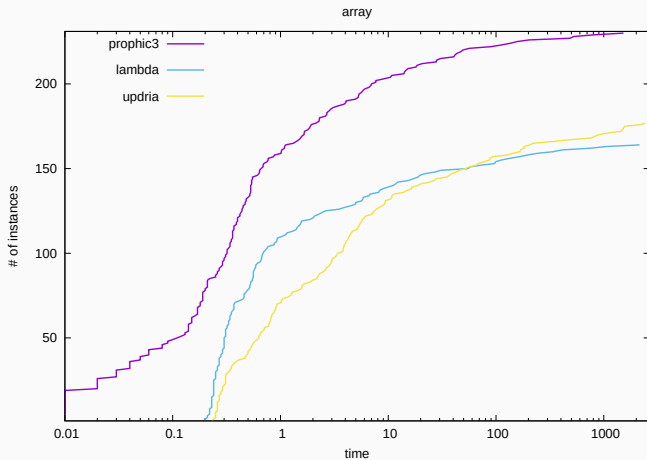
Condition for termination of UPDRIA

Given a set of predicates $\mathcal{P}$, if the states of $\tilde{\mathcal{S}}_{\mathcal{P}}$ form a well-quasi-order, then the algorithm either finds an inductive invariant or a proof of the non-existence of an inductive invariant over $\Sigma_I \cup \mathcal{P}$.

Corollary

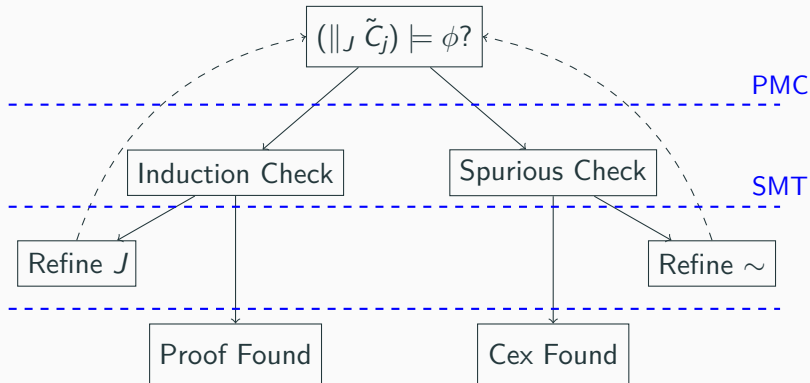
Suppose $\mathcal{T}_I$ is the theory of a linear order. Suppose also that, at any point of the procedure, the index predicates $\mathcal{P}$ contain only unary or constant predicates. Then, if there exists a universal invariant over $\Sigma_I \cup \mathcal{P}$, algorithm UPDRIA is guaranteed to find it.

Data Comparison - Array benchmarks



Algorithms on Array benchmarks

Compositional Approach: Overview



The compositional procedure

Dafny Case Study

```
datatype RouteState = idle | wait | active
class Route
{
  const usedtracks : seq<Track>
  var state : RouteState
  method activateRoute()
    modifies this
    modifies this.usedtracks.state
    requires this.state = wait
    requires ( $\forall t \bullet t \text{ in usedtracks} \implies (t.state = \text{free}$ 
       $\wedge t.unlock\_command = \text{false})$ )
    ensures this.state = active
    ensures  $\forall t \bullet (t \text{ in this.usedtracks} \implies$ 
       $t.state = \text{locked})$ 
    {
      var i := 0;
      while i < |this.usedtracks|
        invariant  $0 \leq i \leq |this.usedtracks|$ 
        invariant  $\forall t \bullet t \text{ in this.usedtracks}[..i]$ 
           $\implies t.state = \text{locked}$ 
        {
          this.usedtracks[i].state := locked;
          i := i + 1;
        }
        this.state := active;
      }
    }
  //...
}
```

Figure 1: Dafny encoding of the route component

Dafny Case Study - Part II

$$\begin{aligned} T_2 = \exists i : \text{route} \Big(& (\text{state_r}(i) = \text{wait}) \wedge \\ & (\forall t : \text{track} (\text{tracks_used}(t, i) \rightarrow \\ & \quad (\text{state_t}(t) = \text{free} \wedge \text{unlock_command}(t) = \text{false}))) \wedge \\ & (\forall j_0 : \text{route}, j_1 : \text{track} (\\ & \quad \text{state_r}'(j_0) = \begin{cases} \text{active} & \text{if } i = j_0 \\ \text{state_r}(j_0) & \text{otherwise} \end{cases}, \\ & \quad \text{state_t}'(j_1) = \begin{cases} \text{locked} & \text{if } \text{tracks_used}(j_1, i) \\ \text{state_t}(j_1) & \text{otherwise} \end{cases}, \\ & \quad \text{unlock_command}'(j_1) = \text{unlock_command}(j_1))) \Big). \end{aligned}$$