

Transaction-Guided Parameterized Model Checking in TxCubicle

Hector Suzanne & Sylvain Conchon

LMF – Université Paris-Saclay / École Normale Supérieure Paris-Saclay / CNRS

Abstract. Designing concurrent and distributed protocols is inherently difficult. Designers typically start from simple, high-level descriptions and progressively refine them into more detailed models that expose additional concurrency, communication patterns, and failure scenarios. For parameterized systems, whose correctness must hold for an arbitrary number of processes, this refinement process can quickly lead to models whose complexity overwhelms automated verification techniques.

This paper introduces TxCubicle, an extension of the Cubicle model checker with transactions that supports this progressive refinement process. Transactions are finite, possibly branching pieces of control flow involving a chosen slice of the system. Each resulting transaction path either completes as a whole or is rolled back and has no effect. These transactions can be composed, refined, split, and branched, allowing users to gradually introduce implementation details and concurrency while maintaining a manageable verification effort.

We have implemented TxCubicle by integrating transactions with Cubicle’s existing verification infrastructure. Transactions can reduce the effective state space explored during verification and guide invariant generation by exposing higher-level behavioral patterns. Our implementation preserves compatibility with Cubicle’s highly optimized symbolic algorithms and SMT-based reasoning without introducing significant overhead. Experimental results show that transactions provide a more natural and efficient alternative to ad hoc control-flow encodings, substantially accelerating invariant generation and improving verification performance even when the underlying reachable state space remains unchanged.

Keywords: Parameterized model checking · Transactions · Cubicle · TxCubicle · MCMT

1 Introduction

Distributed and concurrent protocols are designed for arbitrary numbers of processes rather than for fixed numbers of instances. This parameterization is exactly what makes such protocols attractive to build and difficult to verify automatically: safety must be ensured over an infinite family of finite-state systems [3]. Cubicle is a model checker for such parameterized concurrent systems that addresses this problem in the MCMT tradition [6,11,13,12]. It represents parameterized systems as array transition systems, whose arrays are indexed

by process identifiers, and combines SMT-based backward reachability with finite-instance forward exploration to prove safety properties of those systems [7].

1.1 High-level and low-level models

Cubicle’s system representation exposes interleaving and shared mutable state directly. Those low-level features highlight a tension between the way a protocol is designed and the way it is checked. A designer typically reasons in larger units of behavior: a request is issued, permissions are revoked if needed, access is granted, and the protocol returns to a stable point. During early design, these units are often treated as transactions or user stories. Only later are they split into smaller actions, exceptional branches, retry points, and pessimistic failure modes. In contrast, a conventional array transition system presents the verifier with a flat collection of guarded transitions. Whenever some part of the system satisfies a guard, the corresponding transition may mutate its state. This is, in practice, how distributed and concurrent systems evolve: without following a centralized scenario.

This tension means the high-level narratives that designers use during design are lost in the model. The verifier then explores partial executions and irrelevant interleavings that obscure the intended behavior. It may also have to rediscover, through SMT queries and invariant generation, relationships that are obvious at the protocol level: that a response is the continuation of a particular request, that the same process identifier is reused along an execution path, or that an intermediate state only makes sense in the context of the transaction of which it is a part. A common workaround is to encode this control flow manually inside the model by adding a finite control variable and auxiliary variables that remember the process arguments of the current path [17,19,16]. This makes models harder to read and maintain. More importantly, for MCMT-based verification, it turns structural information into additional logical state that the SMT solver must reason about explicitly.

1.2 Our contribution

We introduce TxCubicle (“Transactional Cubicle”), an extension of Cubicle in which transactions are native to both the modeling language and the verification algorithm. TxCubicle lets users connect ordinary Cubicle transitions, creating sequences and branches over chosen processes. The resulting pieces of control flow are called *transactions*: prescribed interactions in which the selected processes execute predetermined transitions in order. The path commits only if every step succeeds; otherwise, it produces no successor state and another branch or argument choice may be tried from the original state. We currently require those transactions to be cycle-free. Iteration remains available under the standard MCMT paradigm: TxCubicle systems iterate over transactions instead of individual transitions.

The underlying model remains an array transition system, but the intended control flow is made available to Cubicle directly rather than encoded through

additional mutable state. Transactions also support the iterative modeling style in which many protocols are developed. A model can first expose the optimistic paths that describe the intended protocol, then be refined by splitting transactions, adding branches, or allowing interleaving more often as intermediate states and failure modes become relevant.

Although finite acyclic transactions can be compiled into baseline Cubicle and therefore add no logical expressive power, their explicit control flow and process selection let TxCubicle guide invariant generation, backward search, or both at once: backward reachability may compute pre-images of whole transaction paths rather than individual transitions, while forward exploration and invariant generation may likewise follow transaction paths to avoid searching for possible algorithmic invariants outside user-intended traces. Since those two analyses are independent, forward exploration may follow transactions to guide a final safety proof that uses a more pessimistic and fine-grained backward search without transaction guidance. This separation is useful in practice: transaction-level executions often reveal the invariants of a protocol more directly than unconstrained executions over all low-level transitions.

1.3 Outline

We begin by presenting the Cubicle language and the TxCubicle extension in Section 2, using an academic cache model *à la* German. We then explain how (Tx)Cubicle models are checked, give formal semantics for the notion of transaction we implement, and evaluate TxCubicle and our implementation techniques against baseline Cubicle. We finally discuss the limits of our contribution and possible improvements in the conclusion.

2 TxCubicle

Cubicle is a symbolic model checker for distributed and concurrent systems. It describes models as *array transition systems* (ATSSs) and automatically proves safety properties about them. Such systems are naturally presented as collections of arbitrarily many processes, each holding a local state. The state of the whole system is then represented as arrays indexed by process identifiers. At each execution step, a tuple of process identifiers is selected nondeterministically, and if their local states match a Boolean guard, their states are updated. Execution starts from some known safe state, and Cubicle checks whether some *unsafe* state is reachable from the initial safe state.

2.1 Cubicle by example

We present TxCubicle through the example of a model for the German read/write resource sharing algorithm [8]. Here, many processes share a cache line. Either many processes have read access to the cache, or a single process has write access to it. The principle of the algorithm is presented visually in Figure 1. When a

process wants increased access, it takes the relevant solid blue arrow from the diagram, but before it does, all other processes must take the dashed red arrows, ensuring that if one process has write access, no other process has access.

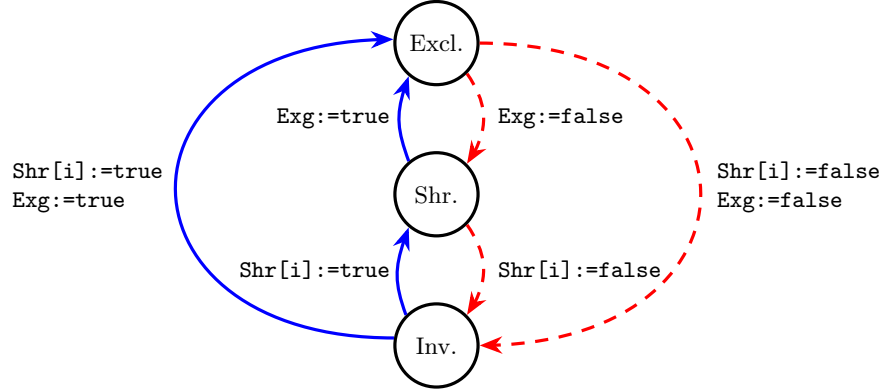


Fig. 1. Diagram for the German algorithm

Access is controlled by the **Exg** Boolean, which is true when the cache is in exclusive-write mode, and by the **Shr** array of Booleans, which records whether processes have access at all. These variables are updated according to the control flow diagram in Figure 2. Starting from **START**, either a request for shared access is created and granted, possibly after revoking exclusive access, or a request for exclusive access is created and granted, after possibly invalidating a number of shared accesses (represented as a cycle between **START** and **inv_shr**).

An idealized control flow here would repeat **inv_shr** within one transaction until **gnt_excl** becomes enabled. The current acyclicity restriction of **TxCubicle** prevents writing this self-loop. The yielding branch therefore returns to **START** after each invalidation, and the model's transition guards constrain the subsequent logical control flow in the same manner as a baseline Cubicle model would.

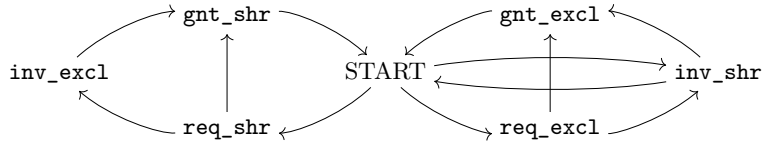


Fig. 2. Control flow diagram of our example German model

The **TxCubicle** model itself is presented in Figure 3. First, enumerations for access states and requests are defined and the system's variables and arrays

are declared (lines 1–10). The `Curcmd` and `Curptr` variables hold the current request message and its sender, and the `Cache` array holds each process’s access level. `Exg` and `Shr` are used to ensure safety as in Figure 1. The initial state of the system is described in lines 12–17 using an implicitly universally quantified variable `z`, and the `unsafe` keyword in lines 19–22 introduces the safety property to be checked. The variables `z1` and `z2` are implicitly existentially quantified and represent two distinct processes: the system is unsafe if one process has write access to the cache while another still accesses it. The rest of the model implements each of the six arrows as *transitions* and orchestrates them using TxCubicle’s transaction annotations, which we have highlighted. The level of detail in the model is increased compared to the diagram: access requests are processed explicitly through transactions, and changing the cache’s permission is not atomic or instantaneous. A basic transition has the form

transition $t(\mathbf{x})$ **requires** $\{ \varphi(\mathbf{x}) \} \{ v := t(\mathbf{x}); \dots \}.$

Here, t is the name of the transition, the process variables $\mathbf{x}$ are its arguments, the predicate $\varphi(\mathbf{x})$ is its *guard*, and the assignments $v := t(\mathbf{x})$ are (taken as a whole) its *action*. For example, in lines 24–29 of Figure 3, the transition `req_shr` is defined. It takes as argument the process identifier n that will initiate a new request. If no request message is being processed (`Curcmd=Empty`) and n has no access to the cache (`Cache[n] = Invalid`), then its new request is registered (`Curcmd:=Reqs; Curptr:=n`). The model’s runtime semantics are given by a loop. At each turn, a transition is chosen nondeterministically, and processes are assigned nondeterministically to its arguments. If this assignment satisfies the guard, then the assignments in the action are performed, and otherwise, the action is skipped.

2.2 Transactional aspects

Our extension introduces new syntax to orchestrate transitions into transactions with rollback-and-retry semantics. Transitions may be chained together by calling a transition immediately after another’s action. For example, line 30 states that after a `req_shr` transition with argument n , either `gnt_shr` must be called with argument n , thereby fulfilling the request, or a call to `inv_excl` must be made with some arbitrary argument denoted by an underscore (note that process arguments in all transitions always remain distinct). The choice between the two calls is nondeterministic. By default, all transitions may start a transaction, and the **triggered** keyword (e.g., line 49) declares that a transition can be executed only when called by another transition. Figure 3 illustrates the two ways to end a transaction. After `inv_shr`, the clause **triggers** `gnt_excl(_)` or **yields** either continues the transaction with `gnt_excl` or terminates it successfully after `inv_shr` (the transition “**yields**” control). The transition `gnt_shr` has no **triggers** block and therefore yields implicitly after its action. Overall, the annotation grammar is therefore

[**triggered**] **transition** ... [**triggers** $t(\mathbf{y})$ [or $t(\mathbf{y})^*$] [or **yields**]].

Here, **transition** ... denotes the basic transition definition of baseline Cubicle. Brackets mark optional syntax and * repetition. Each y_i is a process variable of the enclosing transition or an underscore.

Our transactions have rollback semantics. If the guard of a transition triggered in this way is not satisfied, the entire transaction is rolled back (and can be retried from the beginning, possibly taking another branch or trying another argument). For example, if another process had write access when `req_shr` was called, then the call to `gnt_shr` would fail, and `inv_excl` would be tried instead. Likewise, if the triggered `inv_excl` is given the wrong argument (a process without write access), then the entire transaction is rolled back, and the right choice can be made on a subsequent attempt.

Two caveats about iteration and transaction boundaries need explicit mention. First, chains of triggered transitions must remain finite and cannot contain cycles. This finiteness condition is needed for the path enumeration described later. Iteration is still available through Cubicle's ordinary looping semantics: repeated occurrences of the same transition are modeled as separate nondeterministic steps, with guards enabling or disabling particular transitions. Second, a TxCubicle transaction denotes a prescribed distributed interaction among several processes, which act in isolation from the rest of the system during the transaction. At the transaction level, a selected path either completes as a whole without interleaved transitions, or has no committed effect. Our notion of transaction is therefore better suited to modeling protocols and systems in which processes take part in only one transaction at a time or are naturally isolated from processes in concurrent transactions.

3 Model checking in Cubicle

We now describe the base algorithms used by Cubicle to prove or disprove the safety of models, together with their logical basis. After giving some theoretical background, we describe the *backward reachability algorithm with cubes* at the core of Cubicle and give an overview of how invariant-guessing oracles are used to speed up verification. We pay particular attention to the treatment of logical process variables. The material presented here is not novel, but serves as an essential basis for our treatment of transactions.

3.1 Cubes

Cubicle represents (sets of) states symbolically, as logical formulas in which arrays and global variables appear as free variables. Namely, model states are represented by *cubes* $\varphi(\mathbf{a})$, that is, existential formulas of the form

$$\varphi(\mathbf{a}) := \exists \mathbf{x}. \left(\bigwedge_{i \neq j} x_i \neq x_j \right) \wedge l_1(\mathbf{x}, \mathbf{a}) \wedge \cdots \wedge l_n(\mathbf{x}, \mathbf{a}),$$

```

1 type msg = Empty | Reqs | Reqe
2 type state =
3   Invalid | Shared | Exclusive
4
5 var Exg : bool
6 var Curcmd : msg
7 var Curptr : proc
8
9 array Cache[proc] : state
10 array Shr[proc] : bool
11
12 init (z) {
13   Cache[z] = Invalid
14   && Shr[z] = False
15   && Exg = False
16   && Curcmd = Empty
17 }
18
19 unsafe (z1 z2) {
20   Cache[z1] = Exclusive
21   && Cache[z2] <> Invalid
22 }
23
24 transition req_shr(n)
25 requires { Curcmd = Empty
26   && Cache[n] = Invalid } {
27   Curcmd := Reqs;
28   Curptr := n
29 }
30 triggers gnt_shr(n) or inv_excl(_)
31
32 transition req_excl (n)
33 requires { Curcmd = Empty
34   && Cache[n] <> Exclusive } {
35   Curcmd := Reqe;
36   Curptr := n
37 }
38 triggers inv_shr(_) or gnt_excl(n)
39
40 transition inv_shr(n)
41 requires { Shr[n]=True
42   && Curcmd = Reqe } {
43   Exg := False;
44   Cache[n] := Invalid;
45   Shr[n] :=False
46 }
47 triggers gnt_excl(_) or yields
48
49 triggered
50 transition inv_excl(n)
51 requires { Shr[n]=True
52   && Curcmd = Reqs
53   && Exg=True } {
54   Exg := False;
55   Cache[n] := Invalid;
56   Shr[n] := False
57 }
58 triggers gnt_shr(_)
59
60 triggered
61 transition gnt_shr(n)
62 requires { Curptr = n
63   && Curcmd = Reqs
64   && Exg = False } {
65   Curcmd := Empty;
66   Shr[n] := True;
67   Cache[n] := Shared
68 }
69
70 triggered
71 transition gnt_excl(n)
72 requires { Shr[n] = False
73   && Curcmd = Reqe
74   && Exg = False
75   && Curptr = n
76   && forall_other l.
77     Shr[l]=False } {
78   Curcmd := Empty;
79   Exg := True;
80   Shr[n] := True;
81   Cache[n] := Exclusive
82 }

```

Fig. 3. TxCubicle model of the German algorithm

where $l_1, \dots, l_n$ are literals (applied predicates, possibly negated), and $\mathbf{a}$ are the logical variables for the model's arrays.

In a cube, all process variables $\mathbf{x}$ are existentially quantified and pairwise distinct; dually, a set of literals can be seen as a cube by quantifying all its process variables and adding the required pairwise inequalities to the set. Accordingly, we denote the cube φ over the state $\mathbf{a}$ as either an existentially quantified formula with explicit distinctness constraints, $\varphi(\mathbf{a})$, as above, or as its unquantified conjunctive clause with implicit distinctness constraints, $\varphi(\mathbf{a}, \mathbf{x})$.

3.2 MCMT

Cubicle is based on the *Model Checking Modulo Theory* framework [4,5]. In MCMT, each transition t has an update relation $\mathcal{R}_t(\mathbf{a}_{old}, \mathbf{a}_{new}, \mathbf{x})$ and a guard predicate $\mathcal{G}_t(\mathbf{a}_{old}, \mathbf{x})$, where $\mathbf{a}_{old}$, $\mathbf{a}_{new}$, and $\mathbf{x}$ denote the old state, the new state, and the transition's arguments, respectively. This makes it possible to syntactically compute a symbolic pre-image of a cube $\varphi(\mathbf{a}_{new}, \mathbf{x})$ under a transition $t(\mathbf{y})$. When φ is a cube, the formula

$$pre_t(\varphi(\mathbf{a}_{new}, \mathbf{x}), \mathbf{y}) := \mathcal{G}_t(\mathbf{a}_{old}, \mathbf{y}) \wedge \mathcal{R}_t(\mathbf{a}_{old}, \mathbf{a}_{new}, \mathbf{y}) \wedge \varphi(\mathbf{a}_{new}, \mathbf{x})$$

is its logical pre-image under the transition t . Taking the disjunctive normal form of this pre-image, one can extract a set of conjunctive clauses, each of which can be seen as a cube whose processes are $\mathbf{x} \cup \mathbf{y}$. Therefore, pre-images of cubes under a transition are sets of cubes. This generalizes to pre-image functions $pre_{\mathcal{T}}(\varphi)$ over whole transition systems $\mathcal{T}$.

Cubicle iterates those pre-image computations until the system is proved safe or unsafe. A system is unsafe iff the iterated pre-images of the unsafe cube $\mathcal{U}(\mathbf{a})$ under the system's transitions intersect the initial condition $\mathcal{I}(\mathbf{a})$. In short, we can reduce the model checking problem for array transition systems to a purely logical *backward reachability problem*:

$$\text{The system } (\mathcal{I}, \mathcal{T}, \mathcal{U}) \text{ is unsafe} \iff \left(\bigvee_{n \in \mathbb{N}} pre_{\mathcal{T}}^n(\mathcal{U}) \right) \wedge \mathcal{I} \text{ is satisfiable}$$

The arbitrary-depth pre-images required to decide backward reachability by purely logical means are either too expensive or too complex to be computed for useful models. The more efficient algorithm Cubicle implements proceeds by directly manipulating cubes and using the just-defined cube-level pre-image functions.

3.3 Backward reachability with cubes

The backward-reachability algorithm used in baseline Cubicle is listed on the left of Figure 4; the right-hand algorithm is discussed later. Its main primitive is the cube-level pre-image function $\mathbf{pre}(t, \varphi)$, which is bundled together with an SMT

solver capable of deciding the satisfiability ($\text{sat}(\varphi)$) and validity ($\text{valid}(\varphi)$) of first-order formulas over arrays.

We fix a model $(\mathcal{I}, \mathcal{T}, \mathcal{U})$ as before, and begin by declaring a set of visited cubes V (initially empty, line 1) and a queue of cubes to visit Q (initially containing only $\mathcal{U}$, line 3). Then, as long as Q is not empty, we take a cube φ from it (line 4). If that cube intersects $\mathcal{I}$ according to an SMT solver, then the system is unsafe (line 5). If φ implies the disjunction of all already visited cubes V , then we have already visited all states represented by φ , and we can skip it (line 6). Then, we iterate over all transitions $t(\mathbf{x})$ and possible mappings $\sigma = [\mathbf{x} \mapsto \mathbf{y}]$ from formal arguments of $t(\mathbf{x})$ to actual process identifiers (lines 7–9), and compute for each of them the pre-image of φ under the instantiated transition $t\sigma$ (line 11). Those pre-images are added to the queue (line 12). Once pre-images have been processed for all instances, we add the current cube to the visited set (line 13). Once Q is empty, we know we have seen all iterated pre-images of the unsafe condition and that none intersects the initial condition. The system is therefore safe (line 14).

To build σ , we use the auxiliary function $\text{all_instances}(t(\mathbf{y}), \varphi)$, which computes a list of argument tuples $\mathbf{y}$ suitable for t using the process variables from the current cube φ and a generator for fresh variables. This requires care, as the total number of processes in the system is arbitrary and safety must be proven for all system sizes. Nevertheless, without loss of generality, processes not appearing in φ have arbitrary state and are therefore identical up to logical equivalence. This allows us to implement all_instances by generating all relevant tuples: for a transition t with n arguments and a cube φ with m existential variables, this reduces to computing all n -tuples made from the m preexisting variables and n fresh ones.

The algorithm relies exclusively on the SMT solver to determine the safety and novelty of the currently processed cube. This can be costly, so tight coupling with the SMT solver is required (for example, to progressively update $\mathcal{V}\mathcal{V}$ without recomputation), and care must be taken to minimize the number of solver calls. Furthermore, the number of instantiations of process arguments for each transition contributes significantly to the branching factor of the algorithm.

3.4 Invariant search

For many parameterized protocols, the limiting factor in provability is the discovery of invariants strong enough to make backward reachability converge. Cubicle addresses this problem by using finite executions as a guide. Small instances of the system are run forward via a symbolic representation, and the states they reach are used as oracles to generate those invariants [18,1,15].

Cubicle invariants are expressed in negative form: they are cubes that over-approximate those actually computed by backward reachability. If such an approximation is stable and does not intersect the initial states, it can replace all the cubes falling under it. Those approximations often converge faster than the cubes they replace, since they ignore nontrivial but irrelevant changes in states. Cubes can readily be approximated by simply deleting literals from them. Then,

<pre> 1 Q := empty_queue() 2 V := empty_set() 3 push(Q, U) 4 while $\varphi := \text{pop}(Q)$: 5 if $\text{sat}(\varphi \wedge I)$ then return unsafe 6 if $\text{valid}(\varphi \Rightarrow \bigvee V)$ then continue 7 for $t(x) \in \mathcal{T}$: 8 for $y \in \text{all_instances}(t(x), \varphi)$: 9 $\sigma := [x \mapsto y]$ 10 11 $\psi := \text{pre}(t\sigma, \varphi)$ 12 push(Q, ψ) 13 push(V, φ) 14 return safe </pre>	<pre> 1 Q := empty_queue() 2 V := empty_set() 3 push(Q, U) 4 while $\varphi := \text{pop}(Q)$: 5 if $\text{sat}(\varphi \wedge I)$ then return unsafe 6 if $\text{valid}(\varphi \Rightarrow \bigvee V)$ then continue 7 for $T(z) \in \text{paths}(\mathcal{T})$: 8 for $y \in \text{all_instances}(T(z), \varphi)$: 9 $\sigma := [z \mapsto y]$ 10 11 $\psi := \varphi$ 12 for $t \in \text{rev_steps}(T(z))$: 13 $\psi := \text{pre}(t\sigma, \psi)$ 14 push(Q, ψ) 15 push(V, φ) 16 return safe </pre>
(a) Cubicle	(b) TxCubicle

Fig. 4. Backward reachability algorithms for (Tx)Cubicle (changes highlighted in yellow)

for each backward cube, each subset of its set of literals is a valid candidate for invariant generation, and smaller subsets are more general. More general invariants are more likely to converge, but are also more likely to intersect the initial state or one of its images, which would make them useless.

To select good approximants, Cubicle implements a *forward exploration pass*. Before the backward search starts, Cubicle instantiates the system with a fixed number of processes and explores its reachable states symbolically, up to a depth bound. The resulting set of cubes is used as a source of negative evidence against bad approximations: if approximating a cube causes it to intersect any part of the forward-explored state space, then that approximant is rejected.

4 Model checking with transactions

To verify a TxCubicle model, one cannot simply assume that transitions may fire at any time their guards are satisfied. Since transactions restrict the control flow of the model, its safety may rely on particular transitions, involving particular processes, occurring in sequence. Accordingly, the first step of model checking is to determine which fragments of control flow the model encodes in its transactions.

4.1 Sequentialization

A TxCubicle model can be seen as a control-flow graph, and each run of the model as a path in that graph. Nodes are transitions, and edges are calls to transitions. Formally, the graph of a model has one node for each transition $t(x)$ of the model, and each call in a `triggers` clause is an edge: there is an edge $t(x) \xrightarrow{t'(y)} t'(x')$ between the definitions of $t(x)$ and of $t'(x')$ for each call of the form “ `transition  $t(x)$  ... triggers ... or  $t'(y)$  ...` ”.

We tag nodes as *initial* unless they are declared as **triggered**, and as *final* if they either have no **triggers** clause or have a **triggers ...or yields** clause. Once such a graph is generated, standard graph algorithms can be used to determine its set of paths from initial to final vertices. To ensure that this computation finishes, we require the graph to be acyclic, which makes the set of paths finite.

For example, the control-flow graph in Figure 2 is a compact form of the graph of the German model in Figure 3. The nodes of the figure are the transitions, and each edge of the control flow is a call (whose label is elided in the figure). The extra node START encodes final and initial nodes: edges from START tag their targets as initial, and edges to START tag their sources as final. The acyclicity condition here translates to “each cycle must pass through START”. In other words, START is a transaction boundary rather than an ordinary transition vertex, so such cycles do not create infinite transaction paths. Consequently, transaction-level safety is checked after each invalidation. From that graph, the set of generated paths is

$$\begin{array}{l}
\text{req_shr}(n) \xrightarrow{\text{gnt_shr}(n)} \text{gnt_shr}(n) \\
\text{req_shr}(n) \xrightarrow{\text{inv_excl}(_)} \text{inv_excl}(m) \xrightarrow{\text{gnt_shr}(_)} \text{gnt_shr}(n) \\
\text{req_excl}(n) \xrightarrow{\text{gnt_excl}(n)} \text{gnt_excl}(n) \\
\text{req_excl}(n) \xrightarrow{\text{inv_shr}(_)} \text{inv_shr}(m) \xrightarrow{\text{gnt_excl}(_)} \text{gnt_excl}(n) \\
\text{req_excl}(n) \xrightarrow{\text{inv_shr}(m)} \text{inv_shr}(m) \\
\text{inv_shr}(m) \\
\text{inv_shr}(m) \xrightarrow{\text{gnt_excl}(_)} \text{gnt_excl}(n)
\end{array}$$

4.2 Tracking processes

Each path in the graph of the model corresponds to a sequence of transition calls of the form:

$$\begin{array}{ccccc}
\text{transition } t_1(x_1) & \text{transition } t_2(x_2) & & \text{transition } t_n(x_n) \\
\{\dots\} & \rightarrow \{\dots\} & \rightarrow \dots \rightarrow & \{\dots\} \\
\text{triggers } t_2(y_1) & \text{triggers } t_3(y_2) & & \text{yields}
\end{array}$$

where each element of an argument tuple y_i is either taken from x_i or replaced by an underscore. The next step for model checking in TxCubicle is to determine all possible simultaneous instantiations of the x_i arguments compatible with the y_i . To do so, we first ignore the underscores and pass them through unchanged. We set $\sigma_1 = id$ and, for $1 \leq i < n$, define $\sigma_{i+1} = \sigma_i \circ [x_{i+1} \mapsto y_i]$. For $1 \leq i \leq n$, let $z_i = x_i \sigma_i$. It follows that $z_1 = x_1$ and $z_{i+1} = y_i \sigma_i$. Replacing each transition t_i by $t_i \sigma_i$, we obtain the following path, equivalent to the original, in which all the process variables are now arguments of the first transition:

$$\begin{array}{ccccc}
\text{transition } t_1(z_1) & \text{transition } t_2(z_2) & & \text{transition } t_n(z_n) \\
\{\dots\} & \rightarrow \{\dots\} & \rightarrow \dots \rightarrow & \{\dots\} \\
\text{triggers } t_2(z_2) & \text{triggers } t_3(z_3) & & \text{yields}
\end{array}$$

When no arguments are underscores, the path is fully normalized: it now represents a sequence of transitions over a fixed slice of the system, namely $\mathbf{z}_1$. Otherwise, normalization generates one path per choice of underscore arguments (up to logical equivalence). An underscore in the arguments $\mathbf{y}_i$ of the call $t_{i+1}(\mathbf{y}_i)$ may denote any process, including a caller argument, provided that the callee's complete actual tuple is pairwise distinct. We instantiate underscores iteratively. If $\mathbf{y}_i$ contains k underscores, we form all k -tuples over the variables already known and k fresh variables $z'_1, \dots, z'_k$, retaining exactly the choices that make the complete callee tuple pairwise distinct. The fresh variables introduced at all steps are collected in a set $\mathbf{z}'$, and the assignments satisfy $\bigcup_{i \leq n} \mathbf{z}_i \subseteq \mathbf{z}_1 \cup \mathbf{z}'$. The normalized path then has scope $\mathbf{z} = \mathbf{z}_1 \cup \mathbf{z}'$ and takes the form

$$T(\mathbf{z}) := t_1(\mathbf{z}_1) \rightarrow t_2(\mathbf{z}_2) \rightarrow \dots \rightarrow t_n(\mathbf{z}_n).$$

4.3 Semantics of transactions

Once those normal forms are computed, the logical semantics of a path follows from those of its constituent transitions. In a normalized path $T(\mathbf{z}) = t_1(\mathbf{z}_1) \rightarrow \dots \rightarrow t_n(\mathbf{z}_n)$, each transition's actual argument tuple is pairwise distinct. To that end, let

$$Distinct_T(\mathbf{z}) := \bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j < k \leq |\mathbf{z}_i|} z_{i,j} \neq z_{i,k}.$$

Note that equalities induced by named call arguments are represented by reuse of the same variable in the normalized tuples; no additional disequalities are imposed between variables that do not occur together in such a tuple. The update relation $\mathcal{R}_T(\mathbf{a}_{old}, \mathbf{a}_{new})$ of a path is defined by

$$\begin{aligned} \mathcal{R}_T(\mathbf{a}_{old}, \mathbf{a}_{new}) := & \exists \mathbf{z}. \exists \mathbf{a}_1, \dots, \mathbf{a}_{n+1}. Distinct_T(\mathbf{z}) \wedge \mathbf{a}_{old} = \mathbf{a}_1 \wedge \\ & \left(\bigwedge_{1 \leq i \leq n} \mathcal{G}_{t_i}(\mathbf{a}_i, \mathbf{z}_i) \wedge \mathcal{R}_{t_i}(\mathbf{a}_i, \mathbf{a}_{i+1}, \mathbf{z}_i) \right) \wedge \mathbf{a}_{n+1} = \mathbf{a}_{new} \quad (1) \end{aligned}$$

The states $\mathbf{a}_2, \dots, \mathbf{a}_n$ are the intermediate low-level states of the path. The relation exposes only $\mathbf{a}_{old}$ and $\mathbf{a}_{new}$ as transaction boundaries, and all guards and updates are conjoined, causing an attempted execution with an unsatisfied guard to contribute no pair to $\mathcal{R}_T$ and hence to have no effect. Compare this with the same relation for an ordinary sequence of transitions, shown below: the scope of the existential process variables is wider in our transactional relation. The logic-level consequence of reusing the same slice of the system is a strategic sharing of information among the transitions. Without it, that burden falls solely on the state arrays $\mathbf{a}_i$, which are harder to manipulate symbolically because they are functional objects.

$$\begin{aligned} & \exists \mathbf{a}_1, \dots, \mathbf{a}_{n+1}. \mathbf{a}_{old} = \mathbf{a}_1 \wedge \\ & \left(\bigwedge_{1 \leq i \leq n} \exists \mathbf{x}_i. \left(\bigwedge_{j \neq k} x_{i,j} \neq x_{i,k} \right) \wedge \mathcal{G}_{t_i}(\mathbf{a}_i, \mathbf{x}_i) \wedge \mathcal{R}_{t_i}(\mathbf{a}_i, \mathbf{a}_{i+1}, \mathbf{x}_i) \right) \wedge \mathbf{a}_{n+1} = \mathbf{a}_{new} \end{aligned}$$

4.4 Model checking

Our logical semantics translates directly into an iterative algorithm that computes pre-images of cubes under paths using transition-level pre-image functions. During backward search, pre-images of transitions are replaced with pre-images of transaction paths. The transaction-aware algorithm of TxCubicle is shown on the right of Figure 4, with the changes relative to baseline highlighted. We iterate over entire paths instead of transitions (line 7) and choose a slice of the system, represented by σ , that will remain constant over the entire path (lines 8–9). Then, we traverse the path backward (auxiliary function `rev_steps` on line 11), computing the pre-image of the whole path by taking individual step pre-images in reverse execution order (lines 11–12). Here, `pre` is lifted pointwise from cubes to finite sets of cubes. Only predecessors of the complete path return to the search queue: intermediate pre-image cubes are not checked for safety or fixpoint inclusion. Transaction-aware backward search therefore implements transactional safety.

This suffices to deal with rollbacks. In fact, backward search is where rollback semantics are most natural: going backward through a path, the only pre-images that are generated are those that would have satisfied all guards in the first place. The performance of our algorithm is studied in Section 5.4.

Transaction-aware forward exploration executes a complete path before adding its results to the exploration queue. Therefore, interleaving can occur only at transaction boundaries. Successful intermediate states are nevertheless recorded as negative evidence against invariant candidates. In many cases, those traces suffice to reveal the positive invariant satisfied by the system, since they cover the intended control flow in the system. Restricting forward exploration to transactions therefore reduces the size of the search space for invariants. For example, in our German model, the transactions already span all reachable states of the `Cache` array, namely those in which one cell is `Exclusive` and the others `Invalid`, and those in which some are `Shared` but none `Exclusive`. This partition of cache state is a key invariant of the algorithm.

4.5 Choice of safety criteria

Using transactions in a model allows for a more flexible notion of safety, enabling a more iterative modeling process. In baseline Cubicle, safety checks are done after each transition. Often, distributed algorithms allow safety and invariants to be relaxed in the middle of transactions as long as the system is safe and the invariant is preserved before and after the transaction. TxCubicle can be configured to handle either notion of safety, and likewise for invariants. Using the transaction-level notion of safety, we define the negative safety predicate of a TxCubicle model as follows:

$$\begin{aligned} \exists n \in \mathbb{N}. \exists T_1, \dots, T_n \in \text{paths}(\mathcal{T}). \exists \mathbf{a}_0, \dots, \mathbf{a}_n. \\ \mathcal{I}(\mathbf{a}_0) \wedge \left(\bigwedge_{1 \leq i \leq n} \mathcal{R}_{T_i}(\mathbf{a}_{i-1}, \mathbf{a}_i) \right) \wedge \mathcal{U}(\mathbf{a}_n). \end{aligned} \quad (2)$$

Each $\mathcal{R}_{T_i}$ has the semantics of Eq. 1, including an independent process instantiation satisfying $Distinct_{T_i}$. We admit $n = 0$ (the empty execution), so Eq. 2 includes unsafe initial boundary states; it characterizes boundary reachability, not search termination.

Theorem 1. *The algorithm in Figure 4(b) is sound and semi-complete: it terminates with **unsafe** exactly for unsafe models; on a safe model, it terminates with **safe** or diverges.*

Proof (Sketch). First, we prove that TxCubicle models can be compiled into Cubicle models and that Cubicle’s backward reachability algorithm is correct and semi-complete for the safety criteria above. Then, we prove that TxCubicle’s algorithm, when run on an unmodified model, terminates and returns **unsafe** iff the original algorithm does the same on the modified model.

5 Discussion and evaluation

We evaluate TxCubicle as support for progressive modeling and transaction-guided verification. We compare its native algorithms with baseline Cubicle and a compilation-based encoding on the German example and a semi-industrial model.

5.1 Modeling improvements

In TxCubicle, the control flow and safety boundaries of transactions can each be followed or ignored during the backward and forward searches. The transactional level is more optimistic and structured, and can serve as a stepping stone toward proving lower-level, more mechanical properties. Being able to tune the semantics and proof search of a model offers users an opportunity for iterative modeling that more closely resembles how software design is done in practice. Often, systems are first designed around their “happy paths” or “user stories”. Gradually, these are brought together to form information flow and control flow, and the invariants that the system must preserve can be specified. Then, the model of the system becomes more mechanistic and more pessimistic as transactions are broken down to expose intermediate states. TxCubicle supports these model iterations by exposing the right notion of safety and control flow.

5.2 Comparison with baseline

There are two ways to relate our path-based algorithms to the baseline algorithm. The first treats the added iteration over paths as a coarser unit of backward search than the baseline’s single-transition step. This allows sequences and branches among transitions, letting users describe their system’s control flow naturally rather than mechanistically (we explore those mechanical encodings below). Another way is to consider line 11 of Figure 4(b) as the natural counterpart to line 7 of Figure 4(a). From that point of view, the two algorithms differ in

```

1 transition Path_1_start (n)
2 requires {Control = Start} {
3   arg_1 := n;
4   Control := Path_1_step_1
5 }
6
7 transition Path_1_req_shr (n)
8 requires { Curcmd = Empty
9   && Cache[n] = Invalid
10  && Control = Path_1_step_1
11  && arg_1 = n } {
12   Curcmd := Reqs;
13   Curptr := n;
14   Control := Path_1_step_2
15 }
16 transition Path_1_step_2_gnt_shr(n)
17 requires { Curptr = n
18   && Curcmd = Reqs
19   && Exg = False
20   && Control = Path_1_step_2
21   && arg_1 = n
22 } {
23   Curcmd := Empty;
24   Shr[n] := True;
25   Cache[n] := Shared;
26   Control := Start
27 }

```

Fig. 5. Extract of the compiled German model

that the choice of instances on line 8 is now shared over many transitions. Our algorithm forces groups of processes to evolve together over several transitions at a time. Alongside the added modeling structure, our improvements also help reduce the number of system states considered during solving, thereby reducing the symbolic search space.

5.3 Comparison with compilation pipelines

While TxCubicle integrates cleanly into the backward search algorithm, its semantics as a set of transition sequences also allows it to be embedded in Cubicle proper. This gives an alternative semantics for TxCubicle using the usual encoding of state machines in array transition systems. To do so, we create a new enumeration that holds the position of the system in its control-flow graph (which transition of which path is next in the transaction), and some process variables to hold the current slice of the system for that transaction. Each step of each path is copied from the TxCubicle model and instrumented to trigger only in the correct state and to use the path-level process variables. For example, compiling the first path in the graph ($\text{START} \rightarrow \text{req_shr}(n) \rightarrow \text{gnt_shr}(n)$) would produce the snippet shown in Figure 5.

The compiled model adds a preamble for the new state and augments each copied transition with guards and actions for the control location and saved arguments. Compared with our native setup, the SMT solver has to deal with both the added finite state and, more importantly, the `arg_1` variable, which is itself an array index. This is a pain point for SMT solving in the theory of arrays. Our native implementation sidesteps it by handling those path-level process variables structurally rather than logically.

It might seem attractive to instead precompose each path into a single syntactic transition or model-level relation. We nevertheless believe our combined structural-logical approach strikes a balance that avoids some pitfalls of those approaches. First, MCMT provides no theoretical guarantee as to when the sequence of guards

and actions along a transaction path can be syntactically compiled syntactically into a single guard followed by a single action; it guarantees only that pre-images along such sequences remain tractable via model-theoretic techniques. Likewise, treating an entire path as a single logical entity by submitting the path-level relation of Eq. 1 to the SMT solver is undesirable in practice for performance reasons. The interaction between syntax-level cube management and model-level reasoning is key to baseline Cubicle’s performance compared with pure SMT-based techniques. We therefore operate under the assumption that using path-level logical relations would cause performance regressions.

5.4 Performance analysis and evaluation

Native transactions are more convenient than a state-machine-based encoding of control flow and provide concrete performance benefits. To show this, we compared TxCubicle with the baseline backward-search and forward-exploration algorithms, with and without transaction information. We report in Table 1 the wall-clock time, the number of visited and skipped cubes, the number of calls to the SMT solver, and the maximum number of processes (up to logical equivalence) seen in a state. Care is needed when comparing transactional and non-transactional models, as their semantics do not match: only the prescribed paths are allowed in the former, and they must always be fully traversed.

First, we confirm on the German example (Table 1(a)) that enabling TxCubicle’s scheduler without using transactions does not add significant verification overhead. The non-transactional semantics of the baseline model incur more branching and enable earlier stops. The transactional model traverses fewer states (10 cubes versus 14) and requires more solver calls to process all the paths completely (84 versus 77 SMT calls). We also note that when compiling the transactions manually, the model effectively sees more transitions and holds more mutable state than even the baseline. Indeed, the compiled transactions require 4.2 times as many solver calls and visit 5.4 times as many cubes. As the German example shows, using the control-flow information held in transactions during backward search improves performance over using a systematic compilation pipeline that handles control-flow purely logically.

We also evaluated TxCubicle on a preexisting semi-industrial model of the *FLASH* cache coherence protocol [14] (Table 1(b)). This model’s safety proof requires extensive invariant generation. On this model, we use transactions only during forward exploration, where they act as oracles for control flow relevant to invariant search. We marked twenty-five transitions as **triggered** and left them outside any transaction, preventing forward exploration from using them. We also introduced four two-step transactions, some with high branching factors, that encode the beginning of new get/set requests to the cache. These force the forward search to consider all possible responses to those requests before interleaving other requests. This reduces the number of nodes explored to find the relevant invariants by a factor of more than three and cuts the overall wall-clock time by 76%.

Table 1. Benchmark results

(a) <i>German</i> model						
<i>Tx.</i>	<i>Bwd algo.</i>	<i>Time</i>	<i>Cubes</i>	<i>Skipped</i>	<i>SMT calls</i>	<i>Processes</i>
<i>No</i>	<i>baseline</i>	13.4 ms	16	22	91	3
<i>Yes</i>	<i>baseline</i> ^a	38.5 ms	54	86	354	4
<i>No</i>	<i>ours</i>	13.9 ms	14	19	77	3
<i>Yes</i>	<i>ours</i>	20.5 ms	10	39	84	3
^a Transactions are compiled to be used with the baseline algorithm.						
(b) <i>FLASH</i> model						
<i>Tx.</i>	<i>Bwd algo.</i>	<i>Fwd algo.</i>	<i>Time</i>	<i>Nodes (Fwd)</i>	<i>Nodes (Bwd)</i>	
<i>No</i>	<i>baseline</i>	<i>none</i> ^a	> 300 s	<i>N/A</i>	> 7166	
<i>No</i>	<i>baseline</i>	<i>baseline</i>	2.5 s	191649		37
<i>Yes</i>	<i>baseline</i>	<i>ours</i>	0.6 s	56099		37
<i>Yes</i>	<i>ours</i>	<i>ours</i>	0.7 s	56099		31

^aDid not terminate before the 300 s timeout.

6 Conclusion

Parameterized verification is often presented as the algorithmic problem of finding the symbolic pre-images, abstractions, and invariants that decide a safety property. In practice, model structure also matters. Protocol designers rarely start from a flat set of interleavable rules, but from sequential control flows involving requests, responses, invalidations, retries, and stable points. Business processes, object-centric workflows, and relational systems with unbounded mutable data are likewise modeled as control flows over array transition systems [2,9,10]. For finite acyclic transactions, TxCubicle exposes this structure to the model checker and offers configurable precision without adding logical expressive power.

TxCubicle extends the modeling language with explicit transaction annotations while preserving the underlying array transition system discipline. Our native implementation keeps this control-flow information structural and uses it both to reduce the search space and to guide invariant generation. Our approach sharply reduces the number of explored nodes and SMT calls compared with compilation-based approaches, and produces the needed invariants with substantially fewer forward states and a lower overall runtime. This suggests that transactions are not merely a modeling convenience, but also a fruitful interface between the way protocols and workflows are designed and the symbolic procedures used to verify them. Our implementation shows that transaction-level information is beneficial even when maximally reusing code from baseline Cubicle.

Future work includes using intermediate states in fixpoint checks and replacing eager path enumeration with lazy, stepwise graph traversal. The latter could support cyclic control flow while preserving process scope through SMT operations, but its implementation and correctness remain open. It could also serve as the target of a front-end language for declaring atomic multiprocess interactions.

References

1. Arons, T., Pnueli, A., Ruah, S., Xu, Y., Zuck, L.D.: Parameterized verification with automatically computed inductive assertions? In: *Computer Aided Verification*. pp. 221–234. *Lecture Notes in Computer Science*, Springer (2001). https://doi.org/10.1007/3-540-44585-4_19
2. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Formal modeling and SMT-based parameterized verification of data-aware BPMN. In: *Business Process Management. Lecture Notes in Computer Science*, vol. 11675, pp. 157–175. Springer (2019). https://doi.org/10.1007/978-3-030-26619-6_12
3. Clarke, E.M., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: *Verification, Model Checking, and Abstract Interpretation*. pp. 126–141. *Lecture Notes in Computer Science*, Springer (2006). https://doi.org/10.1007/11609773_9
4. Conchon, S., Declerck, D., Zaidi, F.: Parameterized model checking modulo explicit weak memory models. *Electronic Proceedings in Theoretical Computer Science* **271**, 48–63 (2018). <https://doi.org/10.4204/EPTCS.271.4>
5. Conchon, S., Goel, A., Krstic, S., Majumdar, R., Roux, M.: FAR-cubicle: A new reachability algorithm for cubicle. In: *2017 Formal Methods in Computer-Aided Design*. pp. 172–175. IEEE (2017). <https://doi.org/10.23919/FMCAD.2017.8102256>
6. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaidi, F.: Cubicle: A parallel SMT-based model checker for parameterized systems. In: *Computer Aided Verification*. pp. 718–724. *Lecture Notes in Computer Science*, Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_55
7. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaidi, F.: Invariants for finite instances and beyond. In: *2013 Formal Methods in Computer-Aided Design*. pp. 61–68. IEEE (2013). <https://doi.org/10.1109/FMCAD.2013.6679392>
8. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. *Journal of the ACM* **39**(3), 675–735 (1992). <https://doi.org/10.1145/146637.146681>
9. Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Petri net-based object-centric processes with read-only data. *Information Systems* **107**, 102011 (2022). <https://doi.org/10.1016/j.is.2022.102011>
10. Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Safety verification and universal invariants for relational action bases. In: *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*. pp. 3248–3257. *International Joint Conferences on Artificial Intelligence Organization* (2023). <https://doi.org/10.24963/ijcai.2023/362>
11. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT model checking of array-based systems. In: *Automated Reasoning*. pp. 67–82. *Lecture Notes in Computer Science*, Springer (2008). https://doi.org/10.1007/978-3-540-71070-7_6
12. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *Logical Methods in Computer Science* **6**(4) (2010). [https://doi.org/10.2168/LMCS-6\(4:10\)2010](https://doi.org/10.2168/LMCS-6(4:10)2010)
13. Ghilardi, S., Ranise, S.: MCMT: A model checker modulo theories. In: *Automated Reasoning*. pp. 22–29. *Lecture Notes in Computer Science*, Springer (2010). https://doi.org/10.1007/978-3-642-14203-1_3
14. Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., Hennessy, J.: The stanford FLASH multiprocessor. In: *Proceedings of the 21st Annual International Symposium on Computer Architecture*. pp. 302–313. IEEE (1994). <https://doi.org/10.1109/ISCA.1994.288140>

15. Lahiri, S.K., Bryant, R.E.: Indexed predicate discovery for unbounded system verification. In: Computer Aided Verification. pp. 135–147. Lecture Notes in Computer Science, Springer (2004). https://doi.org/10.1007/978-3-540-27813-9_11
16. O’Leary, J., Talupur, M., Tuttle, M.R.: Protocol verification using flows: An industrial experience. In: 2009 Formal Methods in Computer-Aided Design. pp. 172–179. IEEE (2009). <https://doi.org/10.1109/FMCAD.2009.5351126>
17. Park, S., Dill, D.L.: Protocol verification by aggregation of distributed transactions. In: Computer Aided Verification. pp. 300–310. Lecture Notes in Computer Science, Springer (1996). https://doi.org/10.1007/3-540-61474-5_78
18. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 82–97. Lecture Notes in Computer Science, Springer (2001). https://doi.org/10.1007/3-540-45319-9_7
19. Talupur, M., Tuttle, M.R.: Going with the flow: Parameterized verification using message flows. In: 2008 Formal Methods in Computer-Aided Design. pp. 1–8. IEEE (2008). <https://doi.org/10.1109/FMCAD.2008.ECP.14>