Extracting Counterexamples from Transitive-Closure-based Model Checking

Mitchell Kember, Lynn Tran, George Gao, and Nancy A. Day
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
{mkember, tmltran, y238gao, nday}@uwaterloo.ca

***Abstract*—We address the problem of how to extract counterexamples for the transitive-closure-based model checking (TCMC) technique. TCMC is a representation of the CTLFC (CTL with fairness constraints) model checking problem in first-order logic with transitive closure (FOLTC) and has been implemented in the Alloy Analyzer. It is a declarative, symbolic model checking method. As a CTL model checking method, TCMC is defined over transition systems and states (rather than paths) and therefore, returns a transition system with a bug as a counterexample. Our contribution is to isolate a counterexample path/subgraph in a declarative manner by adding constraints that do not depend on the property. Our method does not require extensions to Alloy.**

I. Introduction

Model checking [8] is a technique for debugging and verifying behavioural models of software systems at a variety of levels of description in the development process. In model checking, a temporal logic property is checked of a formal model of a transition system. If the property fails, a counterexample can be returned, which illustrates a path of the model that does not satisfy the property. The counterexample is of critical value to the user in determining the error in the model of the transition system.

In symbolic model checking, the model is represented as a formula in logic. Traditional symbolic model checking approaches create a formula that is reachable in one step from a set of states (pre/post image) and algorithmically, iteratively call a logic solver until all states have been explored (a fixed point). By remembering the set of states at each step in the iteration, a counterexample path can be produced. BDD-based model checking [23] and IC3 [3], [6] are examples of the iterative approach to symbolic model checking.

Bounded model checking (BMC) [2] is a non-iterative approach to symbolic model checking for checking properties in linear temporal logic (LTL). In BMC, one formula in logic is created to represent a finite path of steps in the transition system. A single call to a solver determines the validity of the temporal logic property, rather than repeated calls as in iterative model checking. BMC checks paths of finite length for linear temporal logic (LTL) properties.

We are investigating the model checking of abstract behavioural models described in first-order logic (FOL). The use of FOL allows the modeller to use abstract/uninterpreted datatypes and functions to represent unknown parts of a model precisely; and to describe transitions declaratively as constraints rather than deterministic operations. FOL is well-suited for describing models concisely early in the development process. It is possible to do BMC on FOL models by forming a FOL formula to represent a finite path and using a FOL solver such as a satisfiable module theories (SMT) solver [1] or a finite model finder such as the Alloy Analyzer [18].

Alloy supports first-order logic with transitive closure (FOLTC)[1]. Alloy is a popular modelling language for describing models abstractly. By limiting the problem to finite scopes, problems in FOLTC become decidable and the constraint solving engine of Alloy is very useful for finding bugs automatically. BMC can be done in the Alloy Analyzer without the use of transitive closure.

Vakili and Day [26], [25] presented a non-iterative, symbolic model checking method called transitive-closure-based model checking (TCMC). TCMC declaratively describes the meaning of properties in computational tree logic with fairness constraints (CTLFC) [8] as a set of constraints in FOLTC. For finite models, the transitive closure operator can be unrolled to create one formula to represent the entire model checking problem to be passed to a solver. Even over a reduced scope, TCMC produces real bugs for liveness properties because it checks an entire sub-transition system, not just paths of a certain length. Unlike LTL, which has meaning relative to paths, CTLFC has meaning relative to states. Therefore, if the property does not pass, the solver returns as an instance a faulty transition system. In this paper, we address the problem of how to extract counterexamples from TCMC. The motivation for our work is that users would rather see a path than an entire transition system to debug their models.

In the following sections, we first present an example that shows the entire faulty transition system now returned from TCMC to demonstrate our goal. Then, we explain the background on TCMC. Next, we describe our solution. The contribution of our work is a constraint-based formulation (rather than algorithmic) of the problem of producing a counterexample for TCMC. Our method does not depend on the property but rather constrains the solution to be a path or subgraph of the transition system. Our approach can

[1] Alloy combines operators on relations, including transitive closure, with first-order quantifiers.

© 2019 IEEE. M. Kember, L. Tran, G. Gao and N. Day, "Extracting Counterexamples from Transitive-Closure-Based Model Checking," 2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE), Montreal, QC, Canada, 2019, pp. 47-54, doi: 10.1109/MiSE.2019.00015.

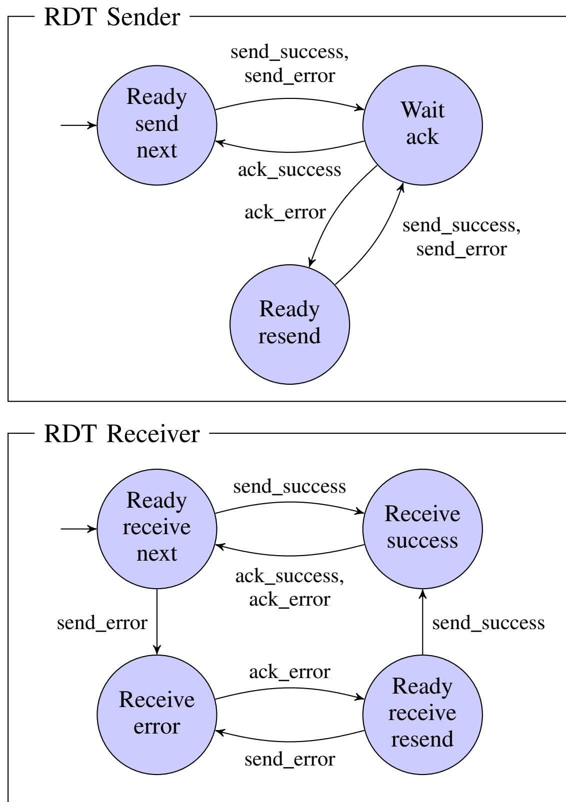


Fig. 1: RDT sender and receiver

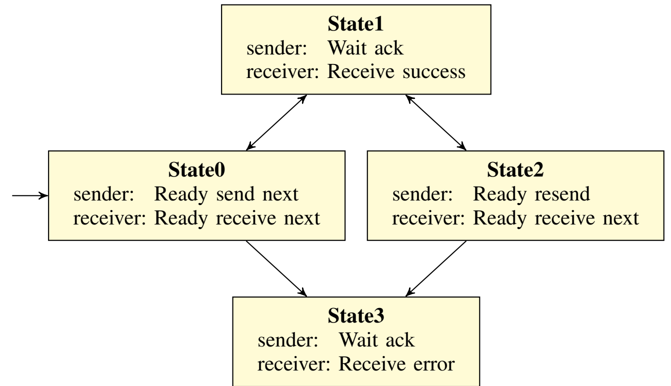


Fig. 2: Transition system counterexample from TCMC

be used in any FOLTC solver. TCMC and our solution are implemented as modules for the Alloy Analyzer, and do not require changes to the implementation of the Alloy Analyzer, making our solution immediately available to users of Alloy. Finally, we describe some case studies, which test the utility and performance of our approach.

II. Example

We use a model of a simple networking protocol as an example to demonstrate the current limitation of TCMC. The model describes a flawed protocol for reliable data transfer (RDT) over an unreliable network. RDT is used by two servers to communicate via alternating data packets and acknowledgment messages. The protocol assumes that any message travelling through the network can be corrupted, in which case detection and recovery are required. To begin, the sender sends a packet to the receiver and awaits a reply. The receiver, upon successful receipt of the packet, replies with a positive acknowledgment. However, in the case where the packet arrives corrupted, the receiver instead responds with a negative acknowledgment and waits for the sender to retransmit. Whenever the sender receives a negative or corrupt acknowledgment message, it resends the previous packet. Fig. 1 shows a concurrent model of a sender and a receiver reacting to four events: *send_success*, *send_error*, *ack_success*, and *ack_error*. Transitions labelled with multiple events means that the transition is taken if either event occurs. *ack_error* can mean either a negative acknowledgment or a corrupt message.

In this model, a sender should not resend a packet when the receiver is expecting the next one. Checking this property using TCMC implemented in Alloy provides the counterexample transition system shown in Fig. 2. A node in this transition system represents a sender-receiver state pair. This output is not the full transition system, but rather a four-element subset of the system containing the counterexample. Within this graph, there is indeed a path from the start state to the problematic state: the path (State0, State1, State2), which occurs when the first two events are *send_success* and *ack_error*. However, the graph also contains states and transitions that are irrelevant to the counterexample. As RDT is rather simple, we can find the counterexample path by manual inspection. However, in more complex systems the path can be hidden inside a very large graph. Our goal is to present the user with a path by removing the irrelevant states and transitions from the transition system.

III. Background

A **transition system** is a five-tuple: $TS = (S, S_0, \sigma, A, l)$, where S is a finite set of states; $S_0 \subseteq S$ is a non-empty set of initial states; $\sigma : S \times S$ is a transition relation; A is a finite set of atomic propositions; and $l : S \to \mathcal{P}(A)$ is the total labelling function.

Computation Tree Logic with fairness constraints (CTLFC) [8] uses a combination of temporal operators and path quantifiers to describe a property of the behaviour of a transition system from a particular state. The path quantifiers A (all paths) and E (some path) are combined with the temporal operators X (next), G (globally), F (eventually), and U (until) to form properties such as $AG\, p$ meaning atomic proposition p is true in all states of the transition system. A primitive set of CTL operators is EU, EG, and EX, from which all the others can be derived. Additionally, the operator $E_c G\, p$ means there exists a fair path on which p is true globally, where a fair path is one where c is true infinitely often. As a branching time logic, the semantics of CTLFC is described over computation trees. A **computation tree** is a tree representing all the (potentially) infinite paths

from an initial state. All LTL properties can be expressed in CTLFC with the addition of variables in the model [9]. Creating a counterexample for a universal temporal formula (contains only As) is the dual of the problem of creating a witness for an existential temporal formula (contains only Es). Clarke *et al.* [10] and Hojati *et al.* [15] describe algorithms for creating witness paths (called linear counterexamples), which generally involve remembering the set of states during the model checking iterations (called stages) and also computing a strongly-connected component within the set of states that satisfy EG. Clarke *et al.* [7] extended this algorithm to handle formulas whose counterexample cannot be described as a single path, but must be stated in terms of a tree. This algorithm recurses through the nested temporal formulas and glues together trees to create a counterexample for the whole formula. Jiang and Ciardo [19] discuss how to extract minimal counterexamples for iterative model checking approaches.

Transitive-closure-based model checking (TCMC) is a declarative, symbolic model checking method for CTLFC properties. Using the (reflexive) transitive closure operator, TCMC describes necessary and sufficient conditions on a finite set of states for them to satisfy a property. The closure operator, a second-order operator, specifies the reachability relation, which is not expressible in FOL. Immerman and Vardi [17] describe an approach (but not an implementation) that uses transitive closure for expressing all of CTL*, which requires the introduction of a new Boolean variable into the transition system for each sub-formula of the property. Vakili and Day [26] present TCMC, which supports only CTLFC, but does not require the introduction of extra Boolean variables. Vakili and Day also implement TCMC in Alloy where the transitive closure operator is automatically unfolded for the finite scope and the result is one formula that represents one model checking query, which is passed to the solver. (In contrast, Hojati *et al.* [16] is an example of an algorithmic approach to using the transitive closure operator for symbolic model checking.)

For a transition relation σ and a CTLFC property φ, TCMC consists of checking $S_0 \subseteq [\varphi]$, where $[\varphi]$ is the set of states that satisfy φ and is defined as follows. σ_X is the (possibly non-total) transition relation σ restricted to domain elements in set X. $\hat{}$ is the transitive closure operator and $*$ is the reflexive transitive closure.

Definition III.1. Transitive-closure-based model checking (TCMC)[2]

1) $[p] = \{s \in S \mid p \in l(s)\}$
2) $[\neg\varphi] = \{s \in S \mid s \notin [\varphi]\}$
3) $[\varphi \vee \psi] = [\varphi] \cup [\psi]$
4) $[EX\varphi] = \{s \in S \mid \exists t \in [\varphi] : \ \sigma(s, t)\}$
5) $[\varphi EU\psi] = \{s \in S \mid \exists t \in [\psi] : \ *(\sigma_{[\varphi]})(s, t)\}$
6) $[EG\varphi] = \{s \in S \mid \exists t \in [\varphi] : *(\sigma_{[\varphi]})(s, t) \wedge \hat{}(\sigma_{[\varphi]})(t, t)\}$
7) $[E_cG\varphi] = \{s \in S \mid \exists t \in [\varphi] : *(\sigma_{[\varphi]})(s, t) \ \wedge \ \hat{}(\sigma_{[\varphi]})(t, t) \ \wedge \ t \in [c]\}$

[2]We use the definition of TCMC found in [13], which is an improved presentation of [26].

In Definition III.1, $[EX\varphi]$ is a set of states that can be reached one step from any state in $[\varphi]$. The definition of $[\varphi EU\psi]$ is a set of states that can reach a state in $[\psi]$ by going through some states in $[\varphi]$ described using the transitive closure of σ. $[EG\varphi]$ is a set of states that can reach some state, t, in $[\varphi]$, and state t must loop back to itself by going through some states in $[\varphi]$ to create an infinite path. Every state in $[E_cG\varphi]$ must reach a state t that satisfies φ and the fairness constraint (c) and that loops back to itself by going through states in $[\varphi]$. As a CTL model checking method, TCMC constraints describe a set of states, not a path (as in LTL model checking). The solver returns as a counterexample/witness a set of states (rather than a path) and a value of σ (the transition relation) in which the property holds. The correctness of TCMC was proven in [25].

Due to the state-space explosion problem, the whole state space usually cannot be fully explored when verifying a property. Farheen [13] describes the meaning of results for **scoped TCMC**. Given a state set of scope *n* in scoped TCMC, where n is less than the size of the entire state space, TCMC checks *all* the full (meaning contains all transitions between the states within the scope), connected subgraphs of size n in the complete transition system from an initial state. From Farheen's methodology, we know:

- For a safety property (which has a finite path counterexample), a counterexample transition system from scoped TCMC contains a *real* bug.
- For a liveness property (counterexample is an infinite path on which the desired proposition is never true), a counterexample transition system from scoped TCMC contains a *real* bug.
- For an existential property (holds for some path), the witness produced is a *real* witness.[3]

Thus, we begin our methodology for extracting counterexamples from a transition system instance that we know has a real bug or witness. We use an implementation of TCMC in Alloy as an Alloy module. No extensions are necessary to the Alloy Analyzer. However, our results are applicable for any implementation of TCMC in FOLTC.

IV. Extracting Counterexample Paths

In this section, we present our constraint-based method for extracting counterexamples in TCMC. The general idea is to add constraints to the model that force the constraint solver to produce an instance as a path or a result less than the whole transition system. In the following subsections, we classify the possible forms of counterexamples, introduce our method for extracting them, describe how to extend the TCMC process with our technique, and discuss limitations in comparison to other model checking techniques.

[3]The witness is guaranteed to contain at least one initial state. It does not guarantee there is a path from every initial state in the complete transition system.

A. *Classification of Counterexamples*

Given a transition system and a universal property it does not satisfy, we wish to produce a counterexample: a specific computation illustrating why the property fails. A counterexample is a **subtree** of the transition system's computation tree starting at its root, including only the paths relevant for refuting the property (it may include finite and infinite paths). There are CTLFC properties whose counterexamples cannot be described in a path but require a branching subtree. For a subtree, there is an associated (finite) **subgraph** of the transition system, containing only the states and transitions used in the subtree. As a set of constraints on a transition system, TCMC produces subgraphs as instances (not subtrees). Our goal is to add constraints that produce a subgraph that isolates the counterexample as clearly as possible. By focusing on the form of the subgraph, we avoid the need to consider the property itself.

Subgraphs can be used to describe counterexamples, but they are less informative than subtrees because they cannot generally specify the order in which transitions are taken. Thus, while every subtree generates a unique subgraph, the reverse is not generally true. For example, consider the transition system in Fig. 3 and the property $AF\,(\neg p \vee AG\,p)$. In words, this asserts that "all paths eventually reach $\neg p$ or a state after which p must always hold." A counterexample should demonstrate that "there is an infinite path along which p always holds yet $\neg p$ is always reachable." This counterexample cannot be a path, but must be a tree. Fig. 4 shows the counterexample subtree (defined recursively because it is infinite) and its associated subgraph. A property with a different counterexample, but the same subgraph, is $(AX\,AX\,p) \vee (AX\,AX\,\neg p)$. The finite tree containing paths S_1, S_2, S_1 and S_1, S_2, S_3 is a counterexample for it and has the same subgraph, but this subtree is not a counterexample for $AF\,(\neg p \vee AG\,p)$. This example shows that a subgraph can describe multiple subtrees and some of these subtrees may not be counterexamples to the property.

First, we characterize the relationship between subgraphs and subtrees. We say a subgraph is **unambiguous** if it is generated by a unique subtree; otherwise, it is **ambiguous**. The significance of unambiguity is that the counterexample subtree can be uniquely determined from the counterexample subgraph. On the other hand, given an ambiguous subgraph that describes a counterexample, from the subgraph we can find subtrees that generate it but we cannot tell valid counterexample subtrees apart from subtrees that happen to use the same states and transitions.

Next, we consider the various forms of subtrees and how they map to subgraphs. An important subcategory of subtrees are **paths**, which have no branches. If a finite path has no repeated states, we call it a **simple path**. Infinite paths must have repeated states, since the transition system is finite. Some infinite paths can be represented by a simple path followed by a state that is a repeat of an earlier state, where a loop begins; such an infinite path called a **lasso**. We use the term **restricted path** to refer to both simple paths and lassos.

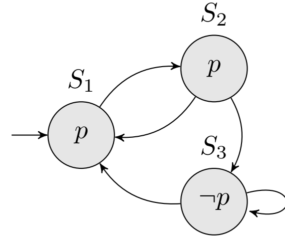
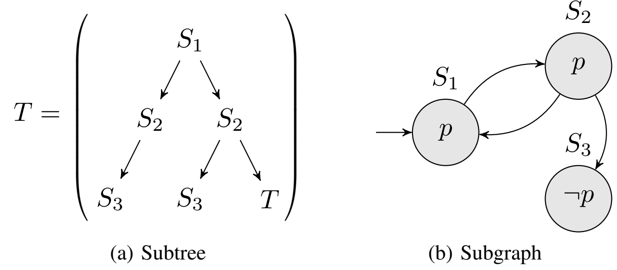


Fig. 3: A three-state transition system



(a) Subtree (b) Subgraph

Fig. 4: Counterexample for $AF\,(\neg p \vee AG\,p)$ on Fig. 3

Not all paths have an unambiguous subgraph. For example, the subgraph in Fig. 4 is ambiguous since it is generated by the finite path S_1, S_2, S_1, S_2, S_3 in addition to the subtree T. As a more extreme example, for some transition systems we can write a property whose only counterexample is a Eulerian path (one that takes every transition once). In this case, the subgraph is the entire transition system, and we cannot infer anything about the counterexample subtree from it.

All simple paths generate unambiguous subgraphs: there is only one possible ordering of states, so the subgraph and subtree are in a one-to-one correspondence. Lassos do not have unambiguous subgraphs, but they have the following similar property: given a subgraph that describes a counterexample, if a lasso generates it, then the lasso is a valid (but perhaps unnecessarily long) counterexample. Based on these observations, we develop two separate counterexample extraction methods: one for restricted paths (Path TCMC) and another for subgraphs (Subgraph TCMC). The former provides more useful debugging information because it generates a path, while the latter only generates a subgraph but works for any property. We choose the boundary between these two methods based whether the subgraph can unambiguously represent a counterexample. In practice, many properties that users check fall under the category of Path TCMC.

B. *Path TCMC*

In this section, we present **Path TCMC**, an extension of TCMC that extracts a restricted path counterexample for a property that does not hold. It can extract both finite paths (simple paths) and infinite paths (lassos). The method is limited to cases where such a path exists; it does not work

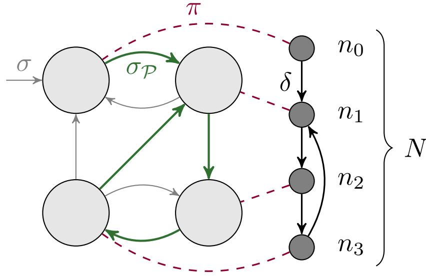


Fig. 5: Example of a lasso represented by $\mathcal{P} = (N, n_0, \delta, \pi)$

when all counterexamples of a property are other types of paths or are branching subtrees.

Given a transition system $TS = (S, S_0, \sigma, P, l)$, we introduce a four-tuple $\mathcal{P} = (N, n_0, \delta, \pi)$ to represent the path, where: N is a finite set of path nodes; $n_0 \in N$ is the initial node; δ is a partial function from N to N representing links between nodes; and π is a total function from N to S. We then define the transition relation of the corresponding subgraph as $\sigma_{\mathcal{P}} = \{(\pi(n), \pi(n')) \mid (n, n') \in \delta\}$, and stipulate three properties to ensure $\mathcal{P}$ represents a path in TS:

- *It respects transitions*: $\sigma_{\mathcal{P}} \subseteq \sigma$.
- *It starts in an initial state*: $\pi(n_0) \in S_0$.
- *It is connected*: $*\delta(n_0, n)$ for all $n \in N$.

$\mathcal{P}$ is a graph that contains only a single finite or infinite path because δ is a function. Via π, its mapping to states in TS, it describes a path through σ, the transition relation of TS. For finite paths, δ is defined for all except one path node, which is the last node. For infinite paths, δ is total. Fig. 5 gives an example of how $\mathcal{P}$ can represent a lasso where N is the set $\{n_0, n_1, n_2, n_3\}$. Note that constraining σ_P (a subset of σ) rather than directly constraining σ, is necessary because σ might be defined concretely or have constraints that force it to always include certain states and transitions.

To perform Path TCMC with a property φ, we apply the temporal operators from Definition III.1 to the transition system $\sigma_{\mathcal{P}}$ instead of σ. We then ask an FOLTC solver, such as the Alloy Analyzer, for an instance that satisfies the TCMC constraints plus the above constraints. If a counterexample path exists, it will return an instance of TS and $\mathcal{P}$. At this point, we know $\sigma_{\mathcal{P}}$ forms a valid counterexample subgraph. We showed in Section IV-A that these subgraphs are either unambiguous or identify an infinite path that is guaranteed to be a valid counterexample. Therefore, we conclude that the restricted path given by $\{\pi(\delta^i(n_0))\}_{i \geq 0}$ is a valid counterexample for φ, where δ^i denotes i repeated applications of δ.

If it is desired to obtain a *minimal* counterexample path, we can iteratively reduce the scope of N. As a final step, we can check for a non-infinite counterexample path by adding the constraint that there exists a path node with no δ-successor (meaning the path is finite). We must do this because the solver could return an infinite path when a finite one would suffice; for example, the link from n_3 back to n_1 in Fig. 5 might be unnecessary, but including it does not change $|N|$.

C. Subgraph TCMC

In this section we present **Subgraph TCMC**, an extension of TCMC that extracts a minimal counterexample subgraph for a property that does not hold. Unlike Path TCMC, it is applicable to all properties, even those whose counterexamples are not paths. However, its results can be more difficult to interpret because it gives the states and transitions used in a problematic scenario, not the subtree that defines the scenario itself.

Given a transition system $TS = (S, S_0, \sigma, P, l)$, we introduce $\sigma_G \subseteq \sigma$ to define a subgraph. Then, similar to Path TCMC, we apply the temporal operators from Definition III.1 to σ_G instead of σ. We iteratively reduce the scope of σ_G given to the FOLTC solver to get a minimal counterexample subgraph. The result is a subgraph containing only the states and transitions necessary to refute φ.

If the solver does not support setting scopes on the size of relations (Alloy does not), then we can simulate it by defining $G = (T, src, dest)$, where T is a set of transition objects and src and $dest$ are total functions from T to S describing the source and destination states of each transition. We then define $\sigma_G = \{(src(t), dest(t)) \mid t \in T\}$, and stipulate three properties to ensure G behaves correctly:

- *It respects transitions*: $\sigma_G \subseteq \sigma$.
- *It has no duplicates*: for all $t, t' \in T$, if $src(t) = src(t')$ and $dest(t) = dest(t')$, then $t = t'$.
- *It is connected*: there is a state $s \in S$ such that $*\sigma_G(s, s')$ for all s' in the images of src and $dest$.

The no-duplicate property is needed to ensure that an exact scope on T results in an exact number of transitions in σ_G. The connectedness property is not strictly needed, since a disconnected subgraph would not be minimal, but it helps to make the non-minimal counterexamples more understandable.

D. Process

Figure 6 outlines our process to extract counterexamples from a transition system that does not satisfy a property. The process assumes that the user has just received a failing instance containing *m* transitions after using TCMC to check a model with state scope *n*. At this point, we know a bug exists in the system, but it is unclear whether the counterexample can be represented by a restricted path.

As a first step, we check the model using Path TCMC; if the counterexample can fit within a restricted path, an instance will be produced. Note that Path TCMC introduces the notion of path nodes, denoted N in Section IV-B, and scoped TCMC imposes a limit on its cardinality. We recommend using scope to *n* to account for the worst case scenario where the path spans every state in the instance.

After Path TCMC returns an instance, users who desire smaller counterexamples can iteratively reduce the scope of path nodes, and re-run Path TCMC. Failure to return an instance indicates that there are no counterexamples involving fewer path nodes. As noted in Section IV-B, users may also add the constraint that there exists a path node with no δ-successor to avoid unnecessarily infinite counterexample paths.

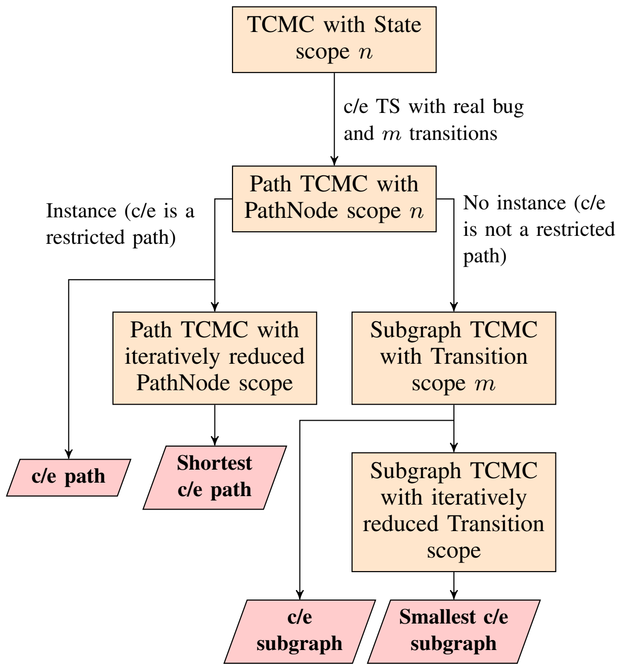


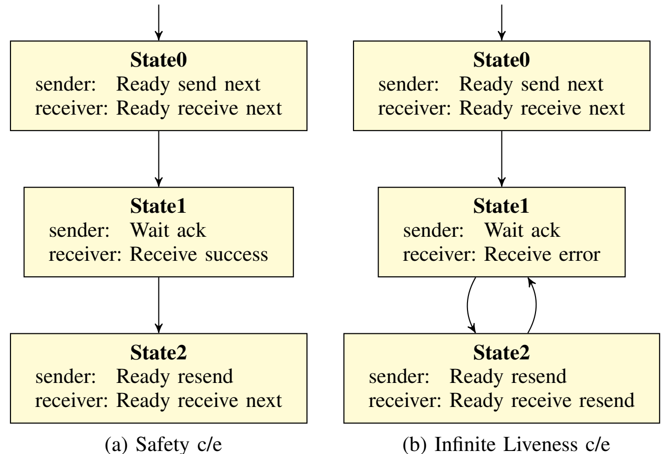
Fig. 6: Counterexample extraction process

If in the first step Path TCMC fails to generate an instance, then we conclude that the counterexample cannot fit within a restricted path. Subgraph TCMC should now be used for extraction, with a scope imposed on the number of relations as described in Section IV-C. We recommend using m as the initial scope, to account for the worst case that the counterexample contains an Eulerian path. Again, users may now iteratively reduce the scope and re-check until the returned subgraph is small enough to understand.

Path TCMC and subgraph TCMC are implemented in additional modules for the Alloy Analyzer.

E. Comparison to Other Techniques

In this section, we compare our method and its usefulness with the popular model checker NuSMV 2.6.0 [5]. When a restricted path counterexample exists, Path TCMC and NuSMV produce similar results: both give a sequence of states and, in the case of a lasso, indicate where the loop begins. For a non-restricted path, NuSMV produces a trace with states in order, whereas with our method we must resort to Subgraph TCMC and lose explicit ordering. However, in these cases the NuSMV trace can have many occurrences of "Loop starts here," which is difficult to interpret (we cannot tell when each loop is taken) and indicates a failure to identify a finite counterexample. For a branching subtree counterexample, as in the example in Fig. 4, Subgraph TCMC shows us all the states and transitions necessary to refute the property but does not give any information about the ordering of transitions as a subtree would. NuSMV, on the other hand, produces an incomplete portion of the subtree: it gives the repeating sequence $S_1, S_2, S_1, S_2, \ldots$, which is the right-hand edge of the full subtree. In general, when refuting a property of the form $\varphi_1 \vee \varphi_2$, NuSMV produces a path refuting only the left disjunct even though both need to be false.



(a) Safety c/e (b) Infinite Liveness c/e

Fig. 7: RDT Alloy counterexample output

V. Case Studies

In this section, we demonstrate our method and investigate its performance using the Alloy Analyzer. Revisiting our RDT example from Section II, we check that there should never be a system state where the receiver is expecting the next packet while the sender is preparing a resend. This safety property is formalized by:

$$AG\,\neg(\text{sender} = \text{“Ready resend”} \wedge \\ \text{receiver} = \text{“Ready receive next”})$$

Our Path TCMC constraints successfully reduced the output to the finite counterexample path shown in Figure 7a.

A naïve RDT liveness property that we can check is that if the receiver receives a corrupt packet, it will eventually receive the corrected one:

$$AG\,(\text{receiver} = \text{“Receive error”} \Rightarrow \\ AF\,\text{receiver} = \text{“Receive success”})$$

The infinite path counterexample that immediately comes to mind is an infinite loop of negative acknowledgments and corrupt packets, in which the correct packet is never received. Indeed, this is the exact path returned by Alloy in Figure 7b, which is a lasso.

We applied our method to some existing small Alloy models, created previous to our work, to show that our solution does not require any changes to the model and to investigate its performance compared to TCMC itself. Our models come from Alloy transition systems generated using Dash [24], and transition systems modeled directly in Alloy [13].

To produce a counterexample from these models, we designed properties that should not be valid in the model. Our properties range from safety to liveness and some are properties whose counterexamples cannot be expressed as paths (called "subgraph properties"). Table I has one row

TABLE I: Performance of TCMC, Path TCMC and Subgraph TCMC in Alloy (minutes/seconds) (“-” means not relevant)

Property	Scope	TCMC	Path TCMC	Subgraph TCMC
Musical Chairs Liveness	11	3.4s	4.0s	-
Musical Chairs Liveness	12	1m08.7s	2.3s	-
Traffic Light Safety	18	3m08.0s	5m30.4s	-
Traffic Light Liveness	18	16.8s	5m37.8s	-
Mutex Safety	10	55.9s	1m33.6s	-
Mutex Safety	14	2m23.3s	17.7s	-
Musical Chairs Subgraph	12	30.7s	1m29.1s	1m51.4s
Traffic Light Subgraph	12	6.1s	2m50.0s	1m12.4s

for each model-property pair being checked. Our experiments were run on an Intel® Core™ i7-4720HQ CPU @ 2.6 GHz running Windows 10 Home.

The TCMC column provides timings for model checking using the base TCMC Alloy module. The Path TCMC and Subgraph TCMC columns show timings of the same process, but with the base TCMC module replaced with our own modules that implement Path and Subgraph TCMC respectively. Each cell in the table is a complete re-run of the Alloy Analyzer's execution, and represents the average time elapsed until the first counterexample instance was found. These executions were run until the elapsed time converged to within 5% of the running average time. Finally, following the process from Figure 6, we ran the Subgraph Module only when the Path Module failed to return an instance (the last two rows of the table).

In most cases, our augmented TCMC modules ran slower than the base TCMC module. However, in cases such as Musical Chairs Liveness scoped to 12 States or Mutex Safety scoped to 14 States, the opposite was observed. Our modules introduce path node and transition objects, increasing the search space. However, they also add constraints between existing objects, reducing the search space. Thus, it is difficult to predict constraint solving times.

VI. Related Work

In this section, we focus on related work on model checking support for abstract modelling languages. BMC [2] unrolls a transition relation for a fixed number of steps to check if an LTL property holds for all paths of this length. Liveness properties can be checked by requiring a loop at the end of the path. BMC can be implemented in Alloy without using transitive closure. A common method for unrolling the transition relation a fixed number of steps is to use the ordering module in Alloy [18], however it disallows repeated states on the path, which means it cannot be used to produce counterexamples for liveness. Cunha [11] extended the ordering module to describe paths with loops. As BMC-based approaches, these methods of model checking are for LTL properties.

Electrum [22] and DynAlloy [14] are extensions to Alloy to model transition systems. Electrum supports BMC and also translates to NuXmv [4], which can verify CTL properties. DynAlloy does not support temporal logic properties but rather pre/post condition reasoning.

Tools for languages such as TLA+ [27], B [21], [20], and ASMs [12] provide model checking via explicit state model checking, BMC, or translations to existing tools.

Overall, Path TCMC and Subgraph TCMC support the generation of counterexamples for a rich set of properties (CTLFC) by the introduction of constraints without any extensions to tool support.

VII. Conclusion

We have presented a constraint-based method for extracting counterexamples from the results of transitive-closure-based model checking for all CTLFC properties, even those whose counterexample is not a path. We characterize how subgraphs represent counterexamples. By focusing on subgraphs, our constraints do not depend on the property and therefore avoid recursing through the nested temporal operators. Since our technique relies on TCMC, which has been proven correct, our method is guaranteed to not produce spurious counterexamples. To produce a path counterexample, our method is purely declarative. Iteration is only used in our method to produce the smallest subgraph or path. A significant advantage of the declarative approach is its simplicity. Our result allows users to focus their attention on the source of the error in a model by reviewing a minimal set of information. Our method is immediately accessible to Alloy users without any extensions to the Alloy Analyzer. Our subgraph representation of counterexamples is useful for constraint-based methods for CTL model checking.

References

[1] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, 2009.

[2] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. In *Advances in Computers*, volume 58, pages 117 – 148. Elsevier, 2003.

[3] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes In Computer Science*, pages 70–87. Springer, 2011.

[4] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *Computer-Aided Verification*, volume 8559 of *Lecture Notes In Computer Science*, pages 334–342. Springer, 2014.

[5] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, and Marco Pistore. NuSMV 2: An opensource tool for symbolic model checking. In *Computer-Aided Verification*, volume 2404 of *Lecture Notes In Computer Science*, pages 360–364. Springer, 2002.

[6] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In *Computer-Aided Verification*, volume 7358 of *Lecture Notes In Computer Science*, pages 277–293. Springer, 2012.

[7] E. Clarke, S. Jha, Yuan Lu, and H. Veith. Tree-like counterexamples in model checking. In *Logic in Computer Science*, pages 19–29. IEEE, 2002.

[8] Edmund Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[9] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[10] EM Clarke, O Grumberg, K McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *ACM/IEEE Design Automation Conference*, pages 427–432. IEEE Computer Society, 1995.

[11] Alcino Cunha. Bounded Model Checking of Temporal Formulas with Alloy. In *International Conference on Abstract State Machines, Alloy, B, VDM, and Z (ABZ)*, volume 8477 of *Lecture Notes In Computer Science*, pages 303–308. Springer, 2014.

[12] Giuseppe Del Castillo and Kirsten Winter. Model Checking Support for the ASM High-Level Language. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes In Computer Science*, pages 331–346. Springer, 2000.

[13] Sabria Farheen. *Improvements to Transitive-Closure-based Model Checking in Alloy*. MMath thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2018.

[14] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. DynAlloy: Upgrading Alloy with Actions. In *International Conference on Software Engineering*, pages 442–451. ACM, 2005.

[15] Ramin Hojati, Robert K. Brayton, and Robert P. Kurshan. BDD-based debugging of designs using language containment and fair CTL. In *Computer-Aided Verification*, pages 41–58. Springer, 1993.

[16] Ramin Hojati, Herve Touati, Robert P. Kurshan, and Robert K. Brayton. Efficient ω-regular language containment. In *Computer-Aided Verification*, pages 396–409. Springer, 1993.

[17] Neil Immerman and Moshe Vardi. Model Checking and Transitive-Closure Logic. In *Computer-Aided Verification*, volume 1254 of *Lecture Notes In Computer Science*, pages 291–302. Springer, 1997.

[18] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[19] Ciardo G. Jiang C. Generation of minimum tree-like witnesses for existential CTL. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10805 of *Lecture Notes In Computer Science*, pages 328–343. Springer, 2018.

[20] Sebastian Krings and Michael Leuschel. Proof assisted bounded and unbounded symbolic model checking of software and system models. *Science of Computer Programming*, 158:41–63, 2017.

[21] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In *FME*, volume 2805 of *Lecture Notes In Computer Science*, pages 855–874. Springer, 2003.

[22] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations. In *Foundations of Software Engineering (FSE)*, pages 373–383. ACM, 2016.

[23] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[24] Jose Serna and Nancy A. Day. Dash. http://129.97.7.33:8080/dash/editor.html. Accessed: 2019-01-27.

[25] Amirhossein Vakili. *Temporal Logic Model Checking as Automated Theorem Proving*. PhD thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2016.

[26] Amirhossein Vakili and Nancy A. Day. Temporal logic model checking in Alloy. In *International Conference on Abstract State Machines, Alloy, B, VDM, and Z (ABZ)*, volume 7316 of *Lecture Notes In Computer Science*, pages 150–163. Springer, 2012.

[27] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer Verlag; New York, 1999.