Noname manuscript No.
(will be inserted by the editor)

Transitive-Closure-based Model Checking (TCMC) in Alloy

Sabria Farheen · Nancy A. Day · Amirhossein Vakili · Ali Abbassi

Received: date / Accepted: date

Abstract We present transitive-closure-based model checking (TCMC): a symbolic representation of the semantics of computational tree logic with fairness constraints (CTLFC) for finite models in first-order logic with transitive closure (FOLTC). TCMC is an expression of the complete model checking problem for CTLFC as a set of constraints in FOLTC without induction, iteration, or invariants. We implement TCMC in the Alloy Analyzer, showing how a transition system can be expressed declaratively and concisely in the Alloy language. Since the total state space is rarely representable due to the state-space explosion problem, we present scoped TCMC where the property is checked for state spaces of a size smaller than the total state space. We address the problem of spurious instances and carefully describe the meaning of results from scoped TCMC with respect to the complete model checking problem. Using case studies, we demonstrate scoped TCMC, and compare it with bounded model checking (BMC), highlighting how TCMC can check infinite paths.

1 Introduction

The process of model-driven engineering (Selic, 2007) promises many benefits from the use of models early in the development process; in general, the earlier that quality models are created, the fewer errors there will be to discover later in the process. A modelling language used early in the design process must be able to express abstract concepts because of the lack of details available at this point in the project. However, if we wish to provide analysis support for these models to increase their quality and utility, we must be able to express the models precisely. Languages such as Alloy (Jackson, 2002), B (Abrial,

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
E-mail: {sfarheen,nday, avakili, aabbassi}@uwaterloo.ca

1996), Z (International Organisation for Standardization, 2000), TLA+ (Yu et al, 1999), and ASMs (Börger, 2005) have many features to express abstract concepts (e.g. sets, relations, and functions) without sacrificing precision. Abstract behavioural models are usually declarative, meaning that they describe a transition system using constraints rather than assignments to variables, in addition to providing more abstract datatypes.

We are interested in the problem of analyzing temporal properties of declarative models of transition systems. In this article, we will use as a running example the game of musical chairs. This game is conveniently and concisely modelled as sequences of transitions that modify a function mapping chairs to the player who occupies the chair. In a declarative model of this game, we can specify that in a step we want all the chairs to be occupied without detailing all the possible combinations of players occupying chairs. An example of a temporal property that we want to verify is that eventually there is a single winner to the game.

There has been a variety of work on verifying temporal properties of declarative behavioural models. In TLA+ (Yu et al, 1999) (with the TLC model checker), a user creates and checks behavioural models for a subset of LTL properties using explicit-state model checking. ProB (Leuschel and Butler, 2008) is a tool for analyzing finite B machines, in particular, simulation and model checking against linear temporal logic (LTL) specifications using explicit-state model checking. Iterative (meaning involve multiple runs of the solver) symbolic model checking algorithms (such as IC3 (Bradley, 2011)) for checking B machines are implemented in (Krings and Leuschel, 2018). None of these approaches use non-iterative symbolic model checking algorithms. A non-iterative symbolic model checking algorithm is one where one formula is constructed and evaluated per model checking query, rather than producing multiple SAT/SMT problems per query.

Del Castillo and Winter provided model checking support for a transition system specified as an Abstract State Machine (ASM) (Börger, 2005) via the translation of a class of ASMs to SMV by restricting the range of functions to finite sets (Del Castillo and Winter, 2000). Chang and Jackson added finite relations and functions to a traditional state-based specification of a transition system (i.e. the SMV language (McMillan, 1992)), and developed a BDD-based model checker that analyzed these models for computational tree logic (CTL) specifications (Chang and Jackson, 2006). Translation-based approaches usually unfold user-level abstractions and make understanding models and counterexamples difficult.

Within the popular Alloy Analyzer toolset, it is fairly straightforward to specify a transition relation and do bounded model checking (BMC) (Biere et al, 1999): create a formula that describes a path for multiple steps to check bounded duration temporal properties (Jackson, 2006). Electrum (Macedo et al, 2016) and DynAlloy (Frias et al, 2005) are extensions of Alloy to model transition systems. Electrun does BMC for LTL properties and DynAlloy checks dynamic properties. Neither of these approaches work without extensions to Alloy or allow us to check a full set of temporal properties for the complete (unbounded) model checking problem.

We seek a non-iterative symbolic model checking method for a full set of temporal properties on a declarative model without translation. If the state-space explosion problem makes it impossible to represent the entire state space for analysis, we would like to avoid spurious instances and have a clear description of what the results from a smaller scope mean for the complete state space.

Describing the traditional representation of the semantics of a temporal logic with respect to a single transition system and state within first-order logic (FOL) is not possible because of the need for quantification over paths (a second-order operator). Thus, using constraint-based first-order solvers for model checking has remained elusive. Immerman and Vardi (Immerman and Vardi, 1997) encoded the semantics of CTL and CTL* in first-order logic with transitive closure (FOLTC). Their semantics has the important property that the use of transitive closure replaces the need for quantification over the paths. Our first contribution (Section 3) is an encoding of CTL with fairness constraints (CTLFC) in FOLTC that is linear in the size of the model, which we call **transitive-closure-based model checking (TCMC)**. Immerman and Vardi's encoding required an exponential increase in the size of the model with respect to the size of the temporal logic formula. TCMC is an expression of the complete (unbounded) model checking problem for a transition system with a finite-state space for CTLFC as a set of constraints in FOLTC without induction, iteration, or invariants. Since the constraints of a declarative model can be satisfied by multiple transition system instances, TCMC can check that either all transition systems that satisfy the constraints satisfy the property (**universal model checking**) or that some instance satisfies the property (**existential model checking**). Novel to TCMC, is that a counterexample is an instance of a transition system with a bug rather than a single counterexample path. Our second contribution (Section 4) is to show that TCMC can be implemented in the Alloy Analyzer, making it possible to do complete model checking of declarative models of transition systems described in Alloy without translation. The model checking problem is turned into a non-iterative constraint solving problem. These first two of our contributions were originally presented in (Vakili and Day, 2012; Vakili, 2016). Here, we give an improved presentation of these results.

Novel to this article, we tackle some of the practical issues in using the Alloy Analyzer for TCMC with results found in the first author's thesis (Farheen, 2018). First in Section 5, we discuss style guidelines for modelling transition systems in Alloy showing an illustrative example. These guidelines do not involve any extensions to Alloy and are relevant for the use of any model checking method in Alloy (not just TCMC). Second, since the total state space is rarely representable due to the state-space explosion problem, we present **scoped TCMC** where the property is checked for transition systems of a certain size that satisfy the constraints of the model (Section 6). Third, we address the problem of spurious instances of transition systems by introducing

significance axioms (Section 6), which require the instance of the model to be of a large enough size to be interesting to the user. Our significance axioms provide a measure independent of computing resource limitations that a significant part of the state space has been verified. Stating these axioms is possible in a model that follows our style guidelines. Since the significance axioms are requirements of transition system instances (rather than path lengths), they are of use in the TCMC methodology. Fourth in Section 7, we present a methodology that carefully describes the meaning of results from scoped TCMC with respect to the complete model checking problem (meaning over the entire state space), highlighting distinctions for properties with respect to finite and infinite paths. Finally, we provide a comparison between TCMC and BMC.

In Section 2, we provide brief background material on CTLFC model checking and the Alloy language. Sections 3 (TCMC), 6 (Significance Axioms), and 7 (TCMC Methodology) are relevant to any verification effort of CTLFC properties (not just in Alloy). The discussion on significant scopes matters for any method where it is not possible to search the entire state space. Our technique has been implemented in the Alloy language and its toolset, which is a popular and well-used verification environment, thus our work has wide applicability. Sections 4 (TCMC in Alloy) and 5 (Modelling a Transition System in Alloy) are Alloy-specific. Section 8 discusses TCMC performance results in the Alloy Analyzer, a comparison of our methodology to BMC, and the use of fairness constraints in TCMC, through case studies. We conclude with related work in Section 9.

2 Background

In this section, we provide a brief overview on temporal logic model checking and Alloy.

2.1 Temporal Logic Model Checking

Temporal logic model checking is a decision procedure for checking whether a transition system satisfies a temporal logic specification (Clarke et al, 1999). A transition system is a finite directed graph with a labelling function that associates a set of propositional variables to each vertex. A vertex represents a state of a system, and the propositional variables that it is labelled with represent the values of the variables in that particular state. An edge between two vertexes represents a transition from one state to another.

Definition 1 Transition System: The transition system TS is a five tuple, $TS = (S, S_0, \sigma, P, l)$, where: S is a finite set of states; S_0, the set of initial states, is a non-empty subset of S; σ, the transition relation, is a binary relation over S; P is a finite set of atomic propositions; and l, the labelling function, is a total function from S to the power set of P.

A computation path starting at s where $s \in S$ is a sequence of states, $s_0 \to s_1 \to \ldots$ such that $s_0 = s$ and $\forall i \geq 0 : \ \sigma(s_i, s_{i+1})$. If the transition relation is a total binary relation then there is at least one infinite computation path starting at each state.

A specification is a set of temporal logic formulas. A temporal logic, such as CTL or CTLFC (Clarke et al, 1999), has logical connectives for specifying properties over the computation paths of a transition system. Equation 1 is the grammar for a complete fragment of CTL:

$$\varphi \ ::= \ p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid \varphi EU \varphi \ , \text{ where } p \in P \tag{1}$$

The satisfiability relation for CTL, $\models$, is used to give meaning to formulas. The notation $TS, s \models \varphi$ denotes that the state s of the transition system TS satisfies the property φ and $TS, s \not\models \varphi$ is used when $TS, s \models \varphi$ does not hold. The relation $\models$ is defined by structural induction on φ:

Definition 2 Semantics of CTL: For a transition system, TS, with a total transition relation σ, the semantics of CTL formulas is as follows.

$TS, s \models p$	iff	$p \in l(s)$
$TS, s \models \neg\varphi$	iff	$TS, s \not\models \varphi$
$TS, s \models \varphi \vee \psi$	iff	$TS, s \models \varphi$ or $TS, s \models \psi$
$TS, s \models EX\varphi$	iff	$\exists s' \in S : \sigma(s, s') \wedge TS, s' \models \varphi$
$TS, s \models EG\varphi$	iff	there exists a path, $s_0 \to s_1 \to \ldots$, where $s = s_0$, and for all i's $TS, s_i \models \varphi$.
$TS, s \models \varphi EU \psi$	iff	there exist a j and a path, $s_0 \to s_1 \to \ldots$, where $s = s_0$, $TS, s_j \models \psi$ and for all i less than j $TS, s_i \models \varphi$.

The transition system TS satisfies the CTL formula φ, denoted by $TS \models \varphi$, if and only if for all $s_0 \in S_0$ we have $TS, s_0 \models \varphi$.

The syntax of a complete fragment of CTLFC is the same as Equation 1 with the addition of one connective, E_cG. In this connective, c is a fairness constraint formula, which is used to define a *fair* computation path. The computation path $s_0 \to s_1 \to \ldots$ is fair with respect to c iff:

$$\{i \mid TS, s_i \models c\} \text{ is infinite.}$$

The semantics of **CTLFC** is the same as Definition 2 along with the semantics of E_cG:

$TS, s \models E_cG\varphi$	iff	there exists a fair computation path with respect to c, $s_0 \to s_1 \to \ldots$, where $s = s_0$, and for all i's $TS, s_i \models \varphi$.

Multiple fairness constraints can be converted to an equivalent property with a single fairness constraint using the method described in (Vakili, 2016), which is based on Vardi and Wolper's work (Vardi and Wolper, 1994). Therefore, we describe our method for a single fairness constraint.

If X is a subset of S, then σ_X denotes the transition relation σ when its domain is restricted to X:

$$\sigma_X(s, s') \text{ iff } \sigma(s, s') \wedge s \in X$$

In this article, ˆ denotes the transitive closure operator; for example, $\hat{}\sigma_X$ is the transitive closure of the relation σ_X. The reflexive transitive closure operator is $*$. The bounding operator has higher precedence than the transitive closure operators: i.e. $\hat{}\sigma_X$ is $\hat{}(\sigma_X)$.

2.2 Alloy

Alloy is a lightweight declarative relational modelling language (Jackson, 2002). The logic that Alloy provides for modelling is essentially first-order logic with the transitive closure operator (FOLTC). An Alloy model consists of a set of declarations, which specify the sets, relations, and functions in a model, and a set of constraints, which are logical formulas. In general, first-order logic is undecidable; as a result, automatic consistency checking of Alloy models is not possible. The Alloy Analyzer, the main analysis tool for Alloy models, provides finite scope analysis: a user is required to choose a finite size for the sets in the model (called the scope) and then after expanding the transitive closure operator for the scope, the Alloy Analyzer translates the model to a propositional CNF formula, which is handed to a SAT solver for consistency checking. By fixing the sizes of the sets in an Alloy model, the Alloy Analyzer evaluates a model for consistency using the `run` command and validity using the `check` command.

3 Transitive-Closure-based Model Checking (TCMC)

Immerman and Vardi show how CTL and CTL* can be encoded in FOLTC for finite models (Immerman and Vardi, 1997). The transitive closure operator is defined only for a finite set. Their encoding of CTL* requires the introduction of Boolean variables into the model for every sub-formula, and as a result, the number of states of a transition system increases exponentially with respect to the size of the formula. They do not provide any implementation of their idea.

In this section, we present our translation of CTLFC to FOLTC with a similar approach to that of Immerman and Vardi. We chose CTLFC for three reasons: 1) unlike CTL*, the encoding of CTLFC in FOLTC does not increase the size of a transition system, 2) it is more expressive than CTL, and 3) LTL model checking can be reduced to CTLFC model checking[1] (Clarke et al, 1997). We call our approach transitive-closure-based model checking (TCMC).

The general idea for TCMC is to use the (reflexive) transitive closure operator to specify the necessary and sufficient conditions for the set of states that

[1] This translation increases the size of a transition system.

satisfy a property. The closure operator is used to specify the reachability relation, which is not expressible in FOL. Similar to traditional representations of CTL model checking, we define an operator, $[\cdot]$, that takes a formula as input and outputs a symbolic representation of the set of states that satisfy the input formula. In TCMC, this operator in defined using transitive closure. The recursive definition for $[\cdot]$ is given in Definition 3. The key difference from the work of Immerman and Vardi is that each formula can be defined directly; support for all of CTL* would require the introduction of a new Boolean variable into the transition system for each sub-formula of the property.

Definition 3 TCMC Let $TS = (S, S_0, \sigma, P, l)$ be a transition system and c be a fairness constraint. The operator $[\cdot]$ takes a CTLFC formula, and produces a subset of S:

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

$[EX\varphi]$ is the set of states that can be reached in one step from states in $[\varphi]$. $[\varphi EU\psi]$ is the set of states that can reach a state in $[\psi]$ via the transitive closure of σ restricted to states in $[\varphi]$. $[EG\varphi]$ are states that can reach some state, t, via the transitive closure of σ restricted to states in $[\varphi]$ and t must loop back to itself via a path of states in the set $[\varphi]$. The definition of $[E_cG\varphi]$ is based on the model checking algorithm of E_cG that finds the strongly connected components (SCCs) in a transition system. The state t in the definition of $[E_cG\varphi]$ is a state that belongs to an SCC and satisfies the fairness constraint, c. This state t must be in a loop (it returns to itself in the transitive closure of σ) of states in the set $[\varphi]$ and is reachable from s via a path of states in the set $[\varphi]$.

Theorem 1 *Let $TS = (S, S_0, \sigma, P, l)$ be a transition system, φ a CTLFC formula, and $[\cdot]$ the operator defined in Definition 3. We have:*

$$[\varphi] = \{s \in S |\ TS, s \models \varphi\}$$

Theorem 1 is proven by structural induction on φ. The proof is straightforward for the first six cases. The details of the proof of this theorem are in (Vakili, 2016). The following corollary of Theorem 1 defines the use of TCMC for model checking a transition system:

Corollary 1 *Let $TS = (S, S_0, \sigma, P, l)$ be a transition system, φ a CTLFC formula, and $[\cdot]$ the operator defined in Definition 3. We have:*

$$TS \models \varphi \textit{ iff } S_0 \subseteq [\varphi]$$

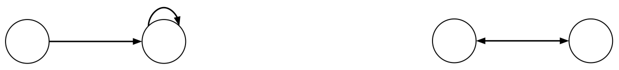

Fig. 1: Multiple instances of a transition system for the constraint: every state must reach a state that is reachable from itself.

If the declarative model of a transition system is not fully defined, there can be multiple instances that satisfy its constraints. For example, the declarative specification "every state must reach a state that is reachable from itself" specifies more than one transition system as shown in Figure 1.

Corollary 1 can be used in two ways because there are potentially multiple instances of σ. **Universal TCMC** is traditional model checking, which checks whether the property is satisfied on all paths starting from all initial states in *all* TS instances of the model and can be accomplished by verifying $S_0 \subseteq [\varphi]$ for all instances of the model. **Existential TCMC** checks if *some* TS instance of the model satisfies the property from all initial states. In this case, we are checking if the model definition is consistent with the property $S_0 \subseteq [\varphi]$.

4 TCMC in Alloy

In this section, we describe the implementation of TCMC in the Alloy language. We create the module `ctlfc` (part of which is shown in Figure 2), which takes the transition system's set of states as a parameter (Line 1). The `TS` (Lines 3-7) declares the sets and relations that are needed to describe a transition system, where `S0` refers to the initial states, `sigma` refers to the transition relation, and `FC` refers to the set of fair states if a fairness constraint is present. These are accessed using the functions on Lines 9–11.

TCMC (Definition 3) is implemented as Alloy functions as shown in Figure 2 Lines 18–29. It uses two helper functions, `domainRes` and `id`, implemented and explained in Lines 13–16. `domainRes[R,X]` is the subset of `R` with its domain restricted to `X`; `id[X]` is the identity relation over `X`. In defining the temporal operators, we take advantage of the Alloy join function, "`.`". For example the `.S` on Line 24 extracts the domain from the relation produced in the rest of the expression. In our complete `ctlfc` module, we also include the universal path quantifiers, `AX, AG, AU, ACG`, defined in terms of the existential temporal operators. Our `ctlfc` module is available on-line[2].

An example template for developing a model to use with TCMC is shown in Figure 3. We import the CTLFC `module` (Line 1). In the `modelDefinition`, on Line 6, we equate the `initialState` function from the module with the initial state constraints of our model. Similarly, we set up the `nextState` relation, and the fairness constraint (`fc`), if any. Then we use `ctlfc_mc` (Lines 10-13) to perform model checking tasks. Our template shows the use of the `ag` and

[2] `https://cs.uwaterloo.ca/~nday/artifacts/`

```
module ctlfc[S]

private one sig TS{
  S0: some S,     // initial states
  sigma: S -> S, // next-state relation
  FC: set S       // fair states
}
----------------------- MODEL ----------------------------------
fun initialState: S {TS.S0}       // initial state constraints
fun nextState: S -> S {TS.sigma}  // transition relation
fun fc: S {TS.FC}                 // fairness constraints
----------------------- HELPER FUNCTIONS _-----------------------
// domainRes[R,X]={(x, y) | (x,y) in R and x in X}
private fun domainRes[R: S -> S, X: S]: S -> S {X <: R}
// id[X]={(x, x) | x in X}
private fun id[X:S]: S->S {domainRes[iden,X]}
----------------- TEMPORAL LOGIC OPERATORS ----------------------
fun not_[phi: S]: S {S - phi}
fun or_[phi, si: S]: S {phi + si}
fun ex[phi: S]: S {TS.sigma.phi}
fun eu[phi, si: S]: S {(*(domainRes[TS.sigma, phi])).si}
fun eg[phi: S]: S {
  let R= domainRes[TS.sigma,phi]|
       *R.((^R & id[S]).S)
}
fun ecg[phi:S]:S {
  let R= domainRes[TS.sigma,phi]|
       *R.((^R & id[S]).S & TS.FC)
}
-------------------- MODEL CHECKING -----------------------------
// used for model checking in user's model file
pred ctlfc_mc[phi: S] {TS.S0 in phi}
```

Fig. 2: Part of CTLFC module in Alloy

```
 1 open ctlfc[State] as ctlfc
 2
 3 sig State { ... }
 4
 5 fact modelDefinition {
 6   all s:State | s in initialState iff ...
 7   all s,s':State | s->s' in nextState iff ...
 8   all s:State | s in fc iff ...
 9 }
10 // universal TCMC
11 check {ctlfc_mc[ag[{s:State| <universal_property>}]]} for exactly <scope>
12 // existential TCMC
13 run {ctlfc_mc[ef[{s:State| <existential_property>}]]} for exactly <scope>
```

Fig. 3: Template for use of TCMC in Alloy

`ef` temporal logic properties, but others can be used. The scope chosen can be for the sets that are components of the state or the `State` set itself.

To implement universal TCMC, we use `ctlfc_mc` with `check`, as shown in Figure 3, Line 11. If the property is satisfied, then the Alloy Analyzer will not be able to find a counterexample. If the property is violated, we get a counterexample – an inspectable transition system that is an instance of our model containing a path that violates the checked property. Unlike other model checking methods, TCMC in Alloy returns an instance of a transition system with a bug rather than a single counterexample path.

For existential TCMC, we use `ctlfc_mc` with `run`, as shown in Figure 3, Line 13. If the model constraints are consistent with the temporal logic property, the Analyzer shows a transition system that is a valid instance of our model. Otherwise, no instance is found.

5 Modelling a Transition System in Alloy

There are many ways that a transition system (TS) can be modelled in FOL. To some extent the style of modelling chosen is based on user preference, however, we have developed some guidelines for Alloy that we find give structure to the model, which we present via an example in this section. These guidelines do not involve any extensions to Alloy. In most uses of symbolic model checking, the user defines a unique transition relation, so our guidelines are focused on defining a declarative model with a single transition system instance. Our example in this section also illustrates the modelling convenience afforded by the abstraction of FOL as compared to writing the same model in the SMV language.

We use the game of musical chairs to illustrate an Alloy model of a transition system. Our model was inspired by Nissanke's model of musical chairs (Nissanke, 1999). As illustrated in Figure 4, each round of the game moves through the modes Start, Walking, Sitting and End. The number of rounds will depend on the number of players; we wish to write a flexible model description that can be used for any number of players, and choose the number of players by setting a finite scope only when we start analyzing the model.

Our behavioural model for Musical Chairs in Alloy consists of three parts: 1) the declaration of the state space, 2) the initial state constraints, and 3) constraints describing the transitions. We combine the constraints describing the transitions to create the transition relation in a standard way.

The state-space definition, as shown in Figure 5, consists of the primitive sets `Chair` and `Player`, and the four possible modes. The `State` set encapsulates the current set of players, chairs, mode, and chair occupancy by players, `occupied`, which is a relation from players to chairs. The use of uninterpreted sets, such as `Chair` and `Player`, plus the use of the relation `occupied` are examples of the abstractions possible in declarative models, which make models concise, but precise.

The encapsulation provided by the `State` set is convenient, but in Alloy such encapsulation is not a record, rather `State` is a distinct set, and the fields are mappings from a `State` element to a set of players, etc. Two `State` elements with the same attribute values are treated as two distinct elements by default. To match our intuition that states with the same attributes are equivalent, we introduce an equality predicate, shown in Lines 11-17 in Figure 5, to force `State` elements with the same attributes to be the same element.

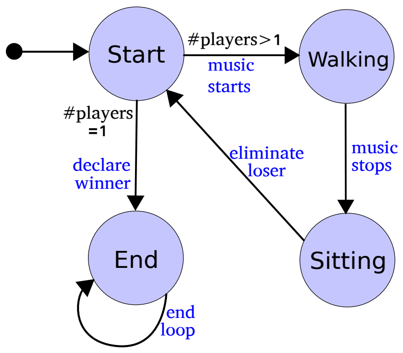

Fig. 4: Musical Chairs Overview

```
sig Chair, Player {}
abstract sig Mode {}
one sig start, walking, sitting, end extends Mode {}
sig State {
  players: set Player,
  chairs: set Chair,
  occupied: set Chair -> set Player,
  mode : set Mode
}
// two states with the same attributes are equivalent
pred equality [s,s':State] {
  (s.players=s'.players and
   s.chairs=s'.chairs and
   s.occupied=s'.occupied and
   s.mode=s'.mode)
     implies s = s'
}
```

Fig. 5: Musical Chairs state space

In Alloy, every element is modelled as a set. Therefore, even though every chair can be associated with at most one player, Alloy treats `occupied` as a relation. If we use the Alloy keyword `lone` (constraining every `Chair` to be associated with at most one `Player`) in the declaration of `occupied`, we are inserting a constraint that may or may not be maintained by the transitions of the transition system. There is the potential to create an inconsistency. We recommend using sets/relations for all attributes of the state space and making the constraint that `occupied` is functional an invariant that is checked by model checking.

The initial state constraints for Musical Chairs, shown in Figure 6, set up the initial `mode` and constrain the number of players in the game.

The Musical Chairs model has five operations as shown in Figure 4: `music_starts`, `music_stops`, `eliminate_loser`, `declare_winner`, and `end_loop`. Any pair of states that satisfies at least one of the operations is a transition in the model. Figure 7 shows the predicates that define the

```
1 pred init [s:State] {
2   s.mode = start
3   #s.players > 1
4   #s.players = (#s.chairs).plus[1]
5 }
```

Fig. 6: Initial state constraints

```
 1 pred pre_eliminate_loser [s: State] {
 2   s.mode = sitting
 3 }
 4 pred post_eliminate_loser [s, s': State] {
 5   s'.mode = start
 6   // loser is the player in the game not in the range of occupied
 7   s'.players = Chair.(s.occupied)
 8   #s'.chairs = (#s.chairs).minus[1]
 9 }
10 pred eliminate_loser [s, s': State] {
11   pre_eliminate_loser[s]
12   post_eliminate_loser[s,s']
13 }
14 ...
15 fact modelDefinition{
16   all s:State | s in initialState iff init[s]
17   all s,s':State | s->s' in nextState iff
18     (music_starts[s,s'] or
19      music_stops[s,s'] or
20      eliminate_loser[s,s'] or
21      declare_winner[s,s'] or
22      end_loop[s,s'])
23   all s, s': State | equality[s,s']
24 }
```

Fig. 7: DisjMethod for Eliminate Loser Operation and Musical Chairs Model Definition

`eliminate_loser` operation in Alloy. For the sake of modularity in the model description, we separate the operation description into separate predicates for the pre- and post-conditions. The pre-condition is a constraint on a single state and the post-condition is a constraint on the previous and the next states. Line 7 removes from the game the player who is not in the range of the `occupied` relation. Line 8 eliminates a chair declaratively, that is, any chair could be the one eliminated. The statement of the operation itself on Lines 10–13 combines the pre- and post-conditions with conjunction.

Lines 15–24 of Figure 7 show the model definition `fact`, which matches the template of Figure 3 and begins to make use of the `ctlfc` module. It equates the `initialState` and `nextState` functions from the `ctlfc` module to the model-specific constraints. A state can be an initial state if and only if it satisfies the constraints listed in the `init` fact, and a pair of states can be in the `nextState` relation if and only if it satisfies the constraints in one of the operations. Each operation is the conjunction of its pre- and post-conditions, and the `nextState` relation is a disjunction of the definitions of each operation. We

```
pred eliminate_loser [s, s': State] {
  pre_eliminate_loser[s] implies
  post_eliminate_loser[s,s']
}
...
fact modelDefinition{
  all s,s':State | s->s' in nextState iff
    (music_starts[s,s'] and
     music_stops[s,s'] and
     eliminate_loser[s,s'] and
     declare_winner[s,s'] and
     end_loop[s,s'])
    ...   }
```

Fig. 8: ConjMethod Method for Defining `nextState` Relation

call this form of model the **disjunctive modelling method (DisjMethod)** for transition relations. The model definition `fact` also enforces the `equality` predicate described previously for all elements of `State`. Because of the equality predicate, where states with the same attributes must be equal, a model written in this manner defines a unique transition system. There are likely multiple TS transitions between states that satisfy the constraints of a single operation.

An alternative, common method for modelling the transition relation is to define each operation as an implication (pre-condition implies post-condition) and conjunct the definitions of all the operations (similar to Dijkstra's guarded commands (Dijkstra, 1975)). We call this the **conjunctive modelling method (ConjMethod)**. An example of this modelling method for musical chairs is shown in Figure 8.

For Musical Chairs, these two modelling methods yield equivalent transition relations, but this is not the case for all models. The two methods produce equivalent transition relations when the pre-conditions are mutually exclusive and complete (some pre-condition is satisfied in every state). Otherwise, the transition relations resulting from these two methods can differ, as illustrated in Figures 9 and 10, where the transition relation is defined as: $\sigma(s, s') \Leftrightarrow (pre_1(s) \wedge post_1(s, s')) \vee (pre_2(s) \wedge post_2(s, s')))$ for the DisjMethod, and as: $\sigma(s, s') \Leftrightarrow (pre_1(s) \Rightarrow post_1(s, s')) \wedge (pre_2(s) \Rightarrow post_2(s, s')))$ for the ConjMethod.

If a state satisfies multiple user-defined operations' pre-conditions, that is, the pre-conditions are not mutually exclusive, then the transition relation from the DisjMethod can include more transitions than the ConjMethod. Figure 9 illustrates this case; the figure only shows transitions that *start* from $S1$. For the ConjMethod, all operations from a state that satisfies their pre-conditions ($S1$) must have their post-conditions satisfied in the next state ($S4$). This requires the next state to satisfy the post-conditions of multiple operations at the same time, thus there are fewer transitions. But for the DisjMethod, only one of the possible post-conditions from a state that satisfies their pre-

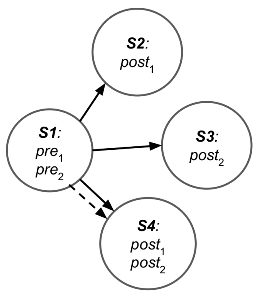

Fig. 9: Overlap in pre-conditions. Shows only transitions starting from $S1$.
Solid lines: DisjMethod transitions.
Dashed lines: ConjMethod transitions.

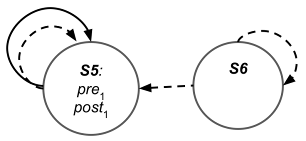

Fig. 10: Incomplete pre-conditions. Shows all transitions between $S5$ and $S6$.
Solid lines: DisjMethod transitions.
Dashed lines: ConjMethod transitions.

conditions ($S1$) needs to hold in the next state ($S2, S3, S4$). So there is a higher number of transitions included in the transition relation for the DisjMethod.

The opposite happens when the pre-conditions of the operations are incomplete, that is, they do not cover all states. Figure 10 illustrates this case; the figure shows *all* transitions occurring between the two states. From a state that does not satisfy any pre-condition ($S6$), transitions to all other states ($S5, S6$) are included in the transition relation for the ConjMethod, because the antecedent of the implications in all the operations is false. So there are more transitions included in the transition relation for the ConjMethod than the DisjMethod in this scenario, although none of these extra transitions are likely ones the user is expecting. While the modelling style is a matter of user preference, we prefer the DisjMethod because it is more modular and additive in nature than the ConjMethod, and we believe it is more likely to produce a transition relation that the modeller is expecting.

To demonstrate the value of the declarative nature of describing behavioural models in Alloy, we compare to a description in NuSMV (Cimatti et al, 2002) of Musical Chairs. In NuSMV, a transition can be described as a constraint, but it lacks abstract data structures. Figure 11 shows the `eliminate_loser` operation described in NuSMV. Lines 1-7 declares some of the elements of the state (players, chairs, and occupied). Since NuSMV does not have sets and relations as native constructs, an array of Booleans is used to represent sets; `occupied` is an array of integers, where the indices represent the chairs and the array values represent the players. A `0` player value is used to designate that a chair is empty. To describe the elimination of a player, there is a case for each player (Lines 15-20), thus the model is for a fixed scope. Additional lines (22-23) are needed to keep track of previously eliminated chairs, which also need to be extended if the scope of chairs is increased. Modules in NuSMV might make this description less verbose, but it is clear that the abstractions

```
VAR
  -- boolean represents whether player is still in the game
  players : array 1..numPlayers of boolean;
  -- boolean represents whether chair is still in the game
  chairs : array 1..numChairs of boolean;
  -- mapping of chairs to players
  occupied : array 1..numChairs of 0..numPlayers;
  ...
TRANS
  case
  ...
    mode = sitting :
    next(mode) = start &
    -- eliminate player if player doesn't occupy any chairs
    ((occupied[1]!=1 & occupied[2]!=1) ?
      !next(players[1]) : next(players[1])=players[1]) &
    ((occupied[1]!=2 & occupied[2]!=2) ?
      !next(players[2]) : next(players[2])=players[2]) &
    ((occupied[1]!=3 & occupied[2]!=3) ?
      !next(players[3]) : next(players[3])=players[3]) &
    -- leave chair outside game if already outside
    ((!chairs[1]) -> next(chairs[1])=FALSE) &
    ((!chairs[2]) -> next(chairs[2])=FALSE) &
    -- eliminate 1 chair for next round
    count(chairs[1],chairs[2]) =
      next(count(chairs[1],chairs[2])) + 1 &
    ...
```

Fig. 11: Eliminate Loser Operation in NuSMV

provided by Alloy are substantially better for writing declarative models that do not depend on a fixed scope.

Our style guidelines are useful for describing a transition system in Alloy in a structured manner, which may be analyzed via TCMC or BMC.

6 Scope, Spurious Instances, and Significance Axioms

To use Alloy's finite model finding capabilities for analysis, we must decide on scopes for all sets. If we set the scope for the basic sets (`Players` and `Chairs` for Musical Chairs), the size of all other sets can be determined, generating a total state space. If we fix the scope of the `State` set to the size of the total state space and run the model, assuming the initial state set is not empty, we would get as an instance the **complete transition system**. However, the total state space is usually too large to be represented in Alloy, especially when the model includes relations. In the Musical Chairs example, `occupied` is a relation between `Chairs` and `Players`. If we have 4 chairs and 5 players, the number of all possible `occupied` relations is 2^{4*5}, and this is for only one element of the state.

One solution is to limit the number of states to the number of reachable states, using a generator axiom that uses the transitive closure operator, however, this is also usually too big. Following Jackson's small scope hypothesis (Jackson, 2006), we try smaller scopes for the `State` set than the entire

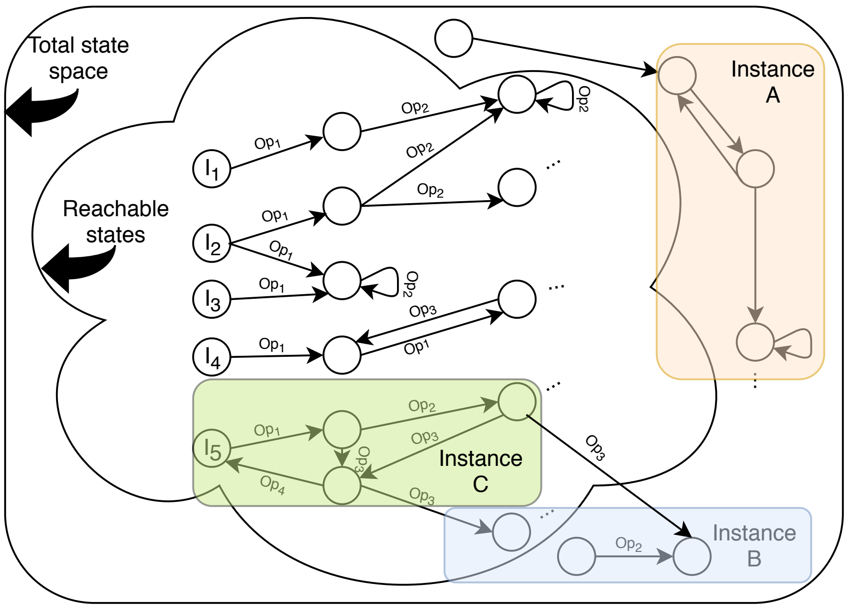

Fig. 12: Complete transition system and examples of its full subgraphs, with and without significance axioms. In the figure, the states labelled with I_n are initial states and the user-defined operations are Op_1, Op_2, Op_3, and Op_4

state space with the goal of finding bugs. We call this method **scoped TCMC**. For a state set of scope n in scoped TCMC, if our model describes a unique transition relation, we inspect *all full subgraphs*[3] of size n in the complete transition system.

This set of full subgraphs consists of all subsets of size n of the state space of the complete transition system, which introduces the spurious instance problem. Spurious instances are instances that satisfy the model but contain disconnected states. Additionally, we also consider an instance that does not include enough of the user-defined operations to be spurious because it is not interesting for the user. Seeing spurious instances does not help us inspect the correctness of the model. Figure 12 illustrates some spurious instances of a hypothetical model. Instances A and B are instances of the model with the scope of exactly 3 states. They are each spurious because some of the states included are not reachable from an initial state or they are disconnected from each other, or both.

For example, in the Musical Chairs model, if we ask Alloy for a transition system satisfying the constraints for scopes where the `Player` set has 3

[3] A full subgraph of a graph is a subset of the nodes with all edges between these nodes that are found in the original graph.

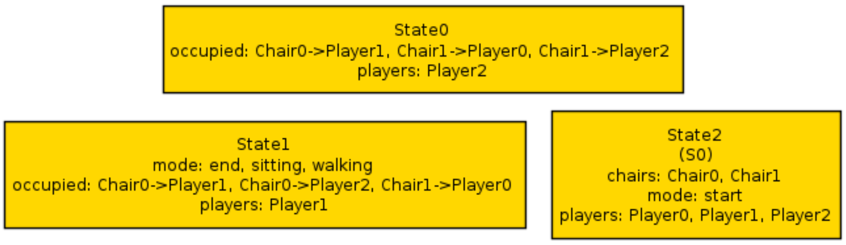
State0
occupied: Chair0->Player1, Chair1->Player0, Chair1->Player2
players: Player2
State1
mode: end, sitting, walking
occupied: Chair0->Player1, Chair0->Player2, Chair1->Player0
players: Player1
State2
(S0)
chairs: Chair0, Chair1
mode: start
players: Player0, Player1, Player2

Fig. 13: A spurious instance returned by Alloy for the Musical Chairs example

elements, the `Chair` set has 2 elements, and the `State` set has 3 elements, it can return an instance such as that shown in Figure 13. This instance is a full subgraph satisfying all constraints of the model, but it has an empty transition relation because none of the pairs of states satisfy any operation. The Alloy Analyzer treats the transition relation as a set of pairs of states and a relation that satisfies the constraints as an instance. It is not useful for a verification run to consider this instance of the model.

We propose a set of axioms, which we call **significance axioms**, that helps us find a small, yet big enough to be interesting, scope that excludes spurious instances from the model. These axioms work whether the transition relation is uniquely defined or not and are relevant to any kind of model checking that cannot inspect the entire reachable state space. These axioms limit the satisfying instances by excluding non-interesting parts. Our significance axioms are:

1. *Reachability Axiom*: All states produced must be reachable from an initial state. This axiom also ensures that an initial state is included, and all transitions in the instance are reachable. Equation 2 represents this axiom, where s and s_i are states, σ is the transition relation, and S_0 is the set of initial states (recall that $*$ is the reflexive transitive closure operator):
$$\forall s \cdot \exists s_i \cdot *\sigma(s_i, s) \wedge s_i \in S_0 \tag{2}$$
2. *Operations Axiom*: At least one transition that satisfies each operation must be included. Equation 3 represents this axiom, where s and s' are states, op is an operation, and $op(s, s')$ is a predicate where (s, s') satisfies the operation op.
$$\forall op \cdot \exists s, s' \cdot op(s, s') \tag{3}$$

Separately from a model checking run, we can find the minimum scope that satisfies the significance axioms by iteratively increasing the size of the state space until a TS instance is returned. We call this scope the **significant scope**. Note that the reachability axiom does not require the inclusion of the entire reachable state space, just that the states included in the instance are all reachable. The uniqueness of the transition relation remains unchanged after adding the axioms. For the abstract example of Figure 12, a scope of

```
pred reachabilityAxiom {
  all s:State | s in initialState.*nextState
}
pred operationsAxiom {
  some s,s':State | music_starts[s,s']
  some s,s':State | music_stops[s,s']
  some s,s':State | eliminate_loser[s,s']
  some s,s':State | declare_winner[s,s']
  some s,s':State | end_loop[s,s']
}
pred significanceAxioms {
  reachabilityAxiom
  operationsAxiom
}
run significanceAxioms for exactly 3 Player,
  exactly 2 Chair, exactly 8 State
```

Fig. 14: Musical Chairs: Significance Axioms. The `initialStateAxiom` and the `totalityAxiom` are not specific to the Musical Chair example.

4 states (Instance C) is needed to satisfy both of these significance axioms. Every state in Instance C is reachable and the instance contains a transition for each user-defined operations. We call an instance that satisfies the significance axioms a **significant instance**. The significance axioms for Musical Chairs are shown in Figure 14. In the Musical Chairs example, the significance axioms ensure we have an instance in which some player wins, but the transition system is not required to include paths for every player to win. One can view our significance axioms as an example of Jackson's generator axioms that are specific for transition systems (Jackson, 2006).

7 TCMC Methodology

Model checking at a small scope evaluates if properties hold for transition systems of that size, but moreover, we can draw some conclusions about whether the properties hold for the complete transition system. In this section, we propose a scoped TCMC methodology. We assume that the complete transition system is uniquely defined.

7.1 Types of Properties

Before introducing our proposed TCMC methodology, we establish some categories for classifying properties. These categories are shown in Figure 15 and based on the negation normal form of the property (negations are only applied to atomic propositions). In our property examples, p and q are atomic propositions. The shaded leaves of the diagram cover all possible CTLFC properties.

The first distinction made is between universal and existential properties. **Universal properties** are CTLFC properties with only universal quantifiers,

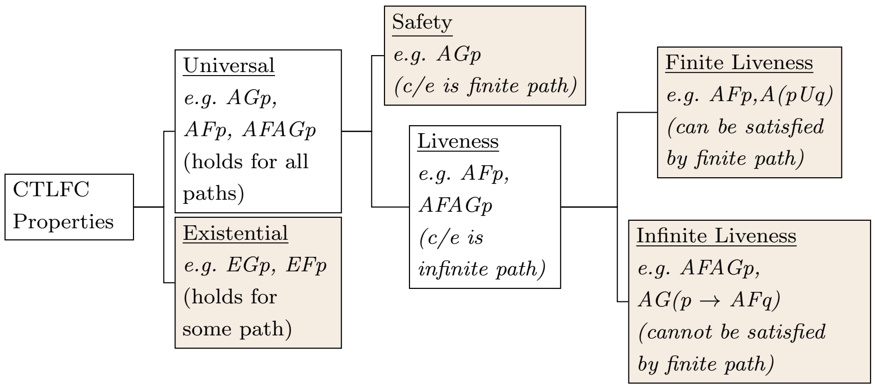

Fig. 15: CTLFC property categories

As, and no existential quantifier, E, in them. These properties are also referred to as ACTL properties (Grumberg and Long, 1991). If the property does not hold, a counterexample, which is a path where the property is not satisfied, can be produced. AGp, AFp and $AFAGp$ are all examples of universal properties. **Existential properties** are CTLFC properties that contain one or more existential quantifier, E, in them. If the property does not hold, no counterexample path can be produced. EGp and EFp are examples of such properties.

Following traditional definitions, universal properties are categorized into safety and liveness properties. **Safety properties** are properties that have finite paths as counterexamples. **Liveness properties** are those that have infinite paths as counterexamples.

Liveness properties are further categorized based on whether they can be *satisfied* by a finite path or not. **Finite liveness properties** are those that *can* be satisfied by finite paths. A property of the form AFp is a finite liveness property. Both finite and infinite paths can satisfy these properties. **Infinite liveness properties** are those that *cannot* be satisfied by finite paths. An example of such a property is one of the form $AFAGp$. Any universal property with a fairness constraint is categorized as an infinite liveness property. Only infinite paths can satisfy these properties.

The rest of this section describes TCMC model checking methodologies and how to interpret results for the complete transition system for these different types of properties. In our figures, we use the word *real* to signify if a pass or fail holds for the complete transition system of the model.

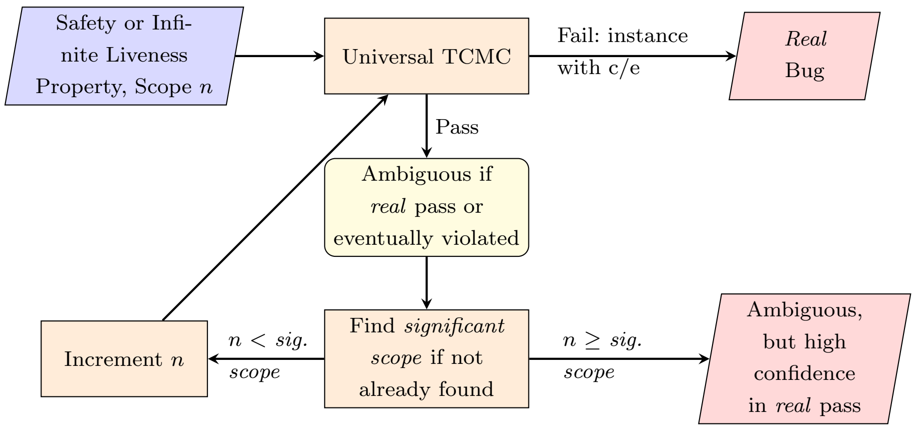
Safety or Infinite Liveness Property, Scope n
Universal TCMC
Fail: instance with c/e
Real Bug
Pass
Ambiguous if *real* pass or eventually violated
Find *significant scope* if not already found
$n < sig.$ *scope*
Increment n
$n \geq sig.$ *scope*
Ambiguous, but high confidence in *real* pass

Fig. 16: TCMC methodology for Safety and Infinite Liveness Properties

7.2 Safety Properties

The process outlined in Figure 16[4] is used to perform TCMC of safety properties. We run universal TCMC as described in Section 4. If the check fails, we get a TS instance with a finite path with a bug; this is a real bug in the complete transition system of the model.

If it passes, we can conclude that it passes in all transition systems of the specified scope, however, for the complete transition system, it is unknown if the pass holds or if a violating state would be encountered at a larger scope. At this stage, we recommend testing the model up to the significant scope, which is the minimum scope required to satisfy our significance axioms, as described in Section 6. We iteratively increment the scope of our `check` and rerun universal TCMC until this significant scope is reached or a failure occurs. We increment iteratively instead of directly checking at the significant scope so that we take advantage of better model checking performance at lower scopes. The process of iteratively increasing the scope is standard in verification practice, however, with our identification of what constitutes a significant scope we can reach a point in the process of having confidence in our pass results because we have checked some significant instances without just exhausting computational resources.

Figure 17 shows an example of checking a safety property in our Musical Chairs model. Here we consider a game starting with 3 players and 2 chairs. We check that the number of players is always one more than the number of chairs, using the `ag` function from the `ctlfc` module. We start the model checking process at a low `State` scope of 2 to detect initial bugs since a lower scope yields better performance. When we get a pass result, we iteratively increment the `State` scope until we reach 8, which is the significant scope for the Musical Chairs model of 3 players and 2 chairs. A pass at this scope

[4] Infinite Liveness, also described in this figure, is explained in a later subsection.

```
assert safety {
  ctl_mc[ag[{s: State| #s.players = (#s.chairs).plus[1] }]]
}
check safety for exactly 3 Player, exactly 2 Chair, exactly 8 State
```

Fig. 17: Checking a safety property of Musical Chairs

gives us considerable confidence that the property is satisfied in the complete transition system.

7.3 Finite Liveness Properties

Although transition systems are often thought of as having only infinite paths generated from a total transition relation, when we perform scoped TCMC in Alloy, the transition systems checked contain a limited number of states, and thus may contain finite paths (i.e. states that have no successor). Finite liveness properties are those that are violated only by infinite paths, but can be satisfied by finite paths. These properties can be checked using scoped TCMC in Alloy using the methodology illustrated in Figure 18.

When checking finite liveness properties, universal TCMC inherently only considers and checks infinite paths[5]. Therefore, if the check fails (while considering only infinite paths), the culprit path in the counterexample instance is an infinite path, guaranteeing that a real bug has been uncovered in the complete transition system of the model.

If the check passes, it is ambiguous whether the property holds for the complete transition system or not, since paths that are finite at the specified scope have not been checked. However, since finite liveness properties can be satisfied by finite paths, it is useful to consider finite paths also. At the given scope, if all paths, finite and infinite, satisfy a finite liveness property, then the property is satisfied for the complete transition system. We check if the given property holds on the finite paths of the transition system by adding **dead-loop** transitions, i.e. a loop at every dead-end state, which is a reachable state with no successor. The template for dead-loop transitions between two states, s and s', is:

$$(\neg(\exists s'' \cdot ops(s, s'')) \wedge (s = s'))$$

where *ops* is a predicate satisfied by any pair of states that are an operation. Adding the dead-loop transitions forces all finite paths in an instance to be infinite by adding a transition from any reachable state without a successor back to itself, which enables TCMC to check finite paths when checking for finite liveness properties. These added transitions make all paths infinite and allow TCMC to distinguish between a real pass and an ambiguous pass. A

[5] The use of id[X] in *EG* (from which *AF* and *AU* are derived) in the TCMC implementation in Figure 2 requires there to be a looping path from a state back to itself to make an infinite path.

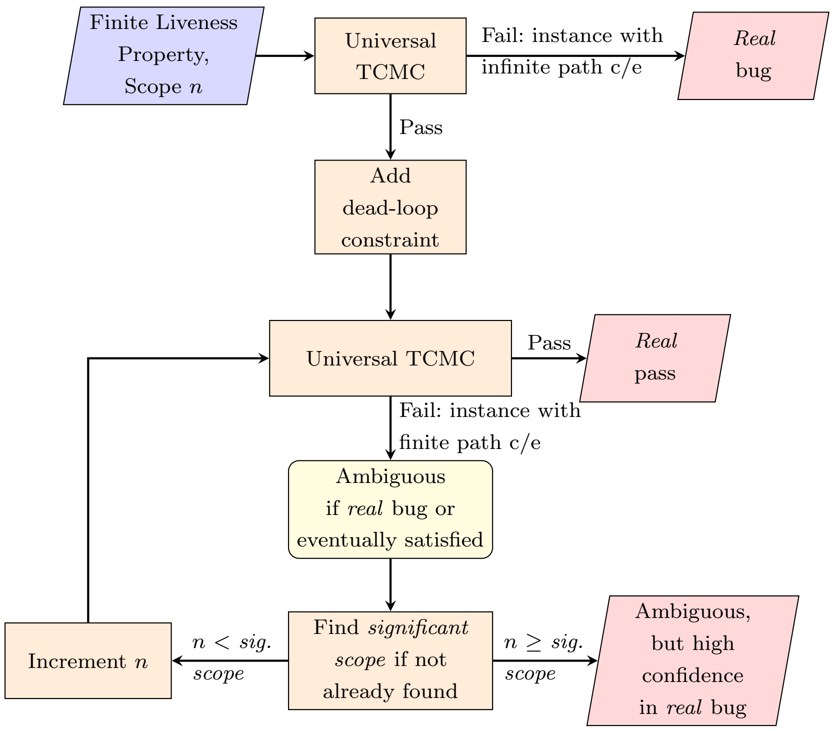
Finite Liveness Property, Scope n
Universal TCMC
Fail: instance with infinite path c/e
Real bug
Pass
Add dead-loop constraint
Universal TCMC
Pass
Real pass
Fail: instance with finite path c/e
Ambiguous if *real* bug or eventually satisfied
Find *significant scope* if not already found
$n <$ *sig. scope*
Increment n
$n \geq$ *sig. scope*
Ambiguous, but high confidence in *real* bug

Fig. 18: TCMC methodology for Finite Liveness Properties

pass result after adding dead-loop transitions means that all paths originating from all initial states reach satisfying states within the limited scope and we can deduce that the property passes in the complete transition system as well, and we can stop our model checking process.

If the check fails, it means that there is a violating finite path in the given scope. However, it is unknown whether the path represents a real bug in the complete transition system or if the finite path can eventually lead to a satisfying state, which makes the fail result ambiguous. To add some assurance to this result, as with safety properties, we model check up to the significant scope. A failure at the significant scope results in higher confidence that the finite liveness property is not satisfied in the complete transition system.

Figure 19 shows an example of checking a finite liveness property in our Musical Chairs model. Here, we check that the game always reaches a `State` with a `sitting` mode, ensuring the game's progress, using the `af` function from the `ctlfc` module. When we start model checking at a scope of 2 `States`, the check passes although vacuously since no infinite paths exist for a scope of 2 for this model. Then we add dead-loop transitions to consider finite paths as well, but the check still fails for scope size 2. We increase the scope to increase confidence since 2 is less than the significant scope. When we increase the

```
assert finiteLiveness {
  ctl_mc[ af [{ s: State| s.mode=sitting }]]
}
check finiteLiveness for exactly 3 Player , exactly 2 Chair,
    exactly 3 State

// ops[s1 ,s2] is a disjunction of the model's operations
pred dead_loop [s,s': State] {
  (no s_n:State | ops[s,s_n]) and s=s'
}

fact modelDefinition {
  all s,s': State | s->s' in nextState iff
    (ops[s,s'] or dead_loop[s,s'])
  ...
}
```

Fig. 19: Checking a finite liveness property of Musical Chairs

`State` scope to 3 (which is not yet the significant scope), we find that the property holds, which is a real pass for the complete transition system.

7.4 Infinite Liveness Properties

An infinite liveness property can only be satisfied and violated by infinite paths, therefore, we only need to consider infinite paths during scoped TCMC. Our proposed method for using TCMC to check infinite liveness properties is outlined in Figure 16 (since it is similar to safety properties).

If TCMC for an infinite liveness property fails, the counterexample produced represents a real bug in the complete transition system. TCMC inherently only considers infinite paths for these properties, meaning that only an instance with a culprit infinite path, thus, representing a real bug, can be produced as a counterexample.

If TCMC passes for such a property, then it is ambiguous whether the result represents a real pass in the complete transition system or a false positive. Longer paths may exist that have not been checked that violate the property. However, as before, model checking up to the significant scope gives us greater confidence in our pass result. There is no point in adding dead-loop transitions to check finite paths in this case, because, unlike finite liveness properties, infinite liveness properties cannot be satisfied by finite paths.

Figure 20 shows an example of checking an infinite liveness property in our Musical Chairs model. We use the `af` and `ag` functions from the `ctlfc` module to check that we always eventually reach a point where the number of players is one and always remains at one at all further states on that path. We start the model checking at a `State` scope of 4. We find that the check passes (although, from our knowledge about the model, we know that this pass occurs vacuously since no paths at this scope are infinite). We repeat the check until we reach a scope of 8, which is the significant scope. At this point, we are relatively confident of our pass result.

```
pred infiniteLiveness {
  // #players eventually always reaches and remains at 1
  ctl_mc[af[ag[{s: State| infiniteLiveness[{s: State | #s.players=1}]]]
}
check infiniteLiveness
  for exactly 3 Player, exactly 2 Chair, exactly 8 State
```

Fig. 20: Checking an infinite liveness property of Musical Chairs

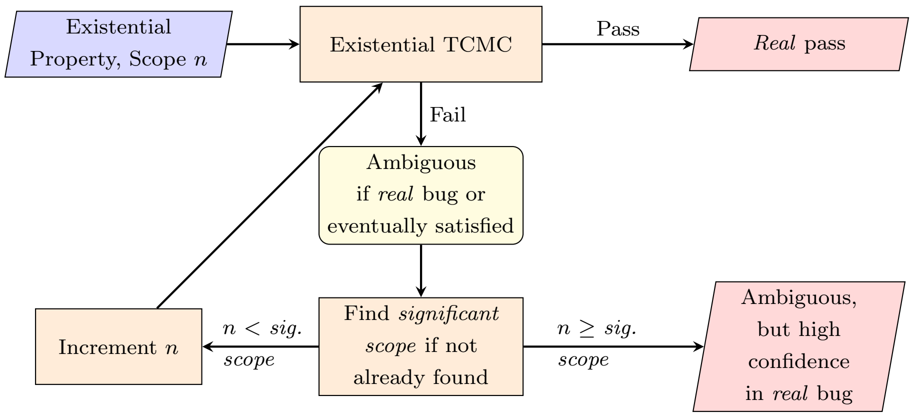
Existential Property, Scope n
Existential TCMC
Pass
Real pass
Fail
Ambiguous if *real* bug or eventually satisfied
Find *significant scope* if not already found
$n <$ *sig. scope*
Increment n
$n \geq$ *sig. scope*
Ambiguous, but high confidence in *real* bug

Fig. 21: TCMC methodology for Existential Properties

7.5 Existential Properties

To check existential properties (including existential properties with fairness constraints) such as EFp or EGp, in TCMC, we use existential TCMC. Checking an existential property using universal TCMC would check if there is *some* path in *all* TS instances of the model that satisfies the property. This check is too strong since to satisfy an existential property, there only needs to be *some* path in *some* TS instance of the model that satisfies the property, which is what we accomplish with existential model checking[6].

Our methodology for checking existential properties is shown in Figure 21. If an existential TCMC `run` returns a satisfying TS instance, then the property passes for the complete transition system of the model because a path (finite or infinite) exists in some TS instance that satisfies the property. If the `run` does not return an instance, it is unknown whether the property fails for the complete transition system, or there exists paths outside the specified scope where the property is satisfied. However, as before, model checking up to the significant scope gives us greater confidence in our pass result.

[6] Existential TCMC requires the satisfying TS instance to have some path from *all* initial states of the TS instance, however, unless the model requires there to be multiple initial states, usually there is a TS instance with only one initial state meaning there is some path from some initial state.

```
one sig Alice extends Player{}
pred existential  {
  ctl_mc[ef[{s: State | s.mode=end and s.players=Alice}]]
}
run existential
  for exactly 3 Player, exactly 2 Chair, exactly 8 State
```

Fig. 22: Checking an existential property of Musical Chairs

Figure 22 shows an example of checking an existential property in our Musical Chairs model. In this example, we assert that there is a player named Alice in the game, and there exists an instance where she eventually wins the game. When we start our model checking process at a low `State` scope of 2, our property fails (since an `end` state has not been reached). We increment the scope but get failures until we reach 8. At this point, we find the property is satisfied, which means it is satisfied for the complete transition system.

8 Case Studies

We developed three case studies in addition to our Musical Chairs example to evaluate TCMC:

– Feature Interaction in a Telephone System (Vakili, 2016)
– Traffic Light Controller (McMillan, 1992)
– Elevator System (Plath and Ryan, 2001; Macedo et al, 2016)

All these models satisfy the properties we checked. The complete Alloy models are available on-line[7]. For TCMC of our models, we used the Alloy Analyzer 4.2 with the MiniSat SAT-solver (Eén and Sörensson, 2004). The experiments were run on an Intel(R) Xeon(R) CPU E3-1240 v5 @ 3.50GHz x 8 machine running Linux version 4.4.0-92-generic with up to 64GB of user-space memory.

The rest of this section discusses the utility of TCMC by examining the feasibility and performance of TCMC at standard Alloy model sizes, a comparison of TCMC to BMC, and the use of fairness constraints in TCMC.

8.1 Scalability

Table 1 shows performance results for four case studies across a range of the types of properties. The scope size (SS) denotes the sum of scopes of all sets. If the execution did not complete within 1 hour, the run was terminated.

With respect to scalability, we found that temporal specifications can be analyzed up to the scopes that non-temporal specifications are often analyzed in Alloy. Thus, our method is immediately valuable to those currently using Alloy for modelling and analysis relying on Jackson's small scope hypothesis.

[7] `https://cs.uwaterloo.ca/~nday/artifacts/`

Musical Chairs. NS: 8, NR: 4				
SS	Safety	Existential	Finite Liveness	Infinite Liveness
8	0.041 s	0.011 s	0.015 s	0.132 s
10	1.037 s	0.076 s	0.025 s	0.379 s
13	8.547 s	0.377 s	0.050 s	4.726 s
15	11 m 51 s	0.488 s	0.096 s	6 m 29 s
18	>1 hour	4.386 s	0.134 s	>1 hour

(a)

Elevator System. NS: 3, NR: 4			
SS	Safety	Finite Liveness	Infinite Liveness
12	0.626 s	1.815 s	2.197 s
13	1.934 s	16.111 s	18.676 s
14	22.621 s	1 m 24 s	4 m 4 s
15	3 m 11 s	9 m 38 s	>1 hour

(b)

Feature Interaction. NS:5, NR:6		Traffic Light Controller. NS:18, NR:5	
SS	Safety	SS	Safety
9	2.54 s	16	0.711 s
10	18.40 s	17	3.815 s
11	9 m 25 s	18	11 m 55 s
12	> 1 hour	19	> 1 hour

(c)

Table 1: Performance Results of Case Studies. NS: Number of Signatures, NR: Number of Relations, SS: Scope Size, m: minutes, s: seconds

The models checked in Alloy are not as large as those that can be checked using a model checker such as NuSMV (Cimatti et al, 2002), however, the declarative and relational aspects of Alloy have significant advantages for creating concise, abstract behavioural models. We have added to Alloy the ability to check complex temporal logic specifications directly on small scopes of these models, and a methodology to make useful conclusions about larger scopes as well.

8.2 Comparison to BMC

Bounded model checking (BMC) (Biere et al, 1999) uses symbolic model checking to verify temporal (generally LTL) properties along paths up to a certain length. It is different from scoped TCMC in that BMC limits the path length whereas scoped TCMC limits the number of states in the transition system. In scoped universal TCMC of scope n, we check all TS instances of size n of all transition systems that satisfies the model's constraints. In BMC, all paths of a certain length of all transition systems that satisfies the model's constraints are checked.

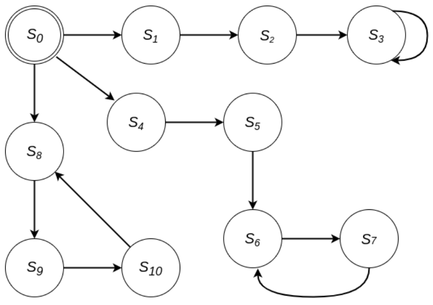

Fig. 23: Example Transition System

Using the example transition system shown in Figure 23, where S_0 is the initial state, we can compare the two approaches. For a bound of 3, BMC looks at the following paths:

- $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$
- $S_0 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6$
- $S_0 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10}$

For a state scope of 4 (which is comparable to a BMC bound of 3 since one more state than the path length may be used in BMC), scoped TCMC considers transition system instances (and paths) such as the following:

- Instance 1: S_0, S_1, S_2, S_3
  - $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_3 \rightarrow$ ... (infinite path)
- Instance 2: S_0, S_4, S_5, S_6
  - $S_0 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6$
- Instance 3: S_0, S_8, S_9, S_{10}
  - $S_0 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow$ ... (infinite path)
- Instance 4: S_0, S_1, S_2, S_4
  - $S_0 \rightarrow S_1 \rightarrow S_2$
  - $S_0 \rightarrow S_4$
- etc. All instances with 4 states.

In TCMC, paths are not limited to the scope size, and can be infinite. In BMC, all paths are finite and of the length specified.

In Alloy, we can perform BMC by utilizing Jackson's `ordering` module (Jackson, 2006). The `ordering` module does not allow repeated states in a path, therefore, it is impossible to represent infinite paths. To compare TCMC in Alloy with BMC in Alloy, we implemented our Musical Chairs model using the `ordering` module, and model checked a safety property and a finite liveness property using BMC. BMC is much faster than TCMC because TCMC checks more instances (and paths) than BMC. However, aside from performance, TCMC has several advantages over BMC in Alloy:

- The counterexamples produced by TCMC for liveness are real bugs in the complete transition system. In BMC, it is not possible to limit the search to

infinite paths, and therefore spurious counterexamples, i.e. instances with violating finite paths that would satisfy the liveness property if extended, are possible. It is possible to represent infinite paths for BMC in Alloy using the method proposed in (Cunha, 2014) (requires extra constraints to represent loops in paths and to consider only infinite paths), which would prevent spurious counterexamples.
- BMC can only check LTL properties, which quantify over paths, which means that it cannot check CTLFC's existential properties. TCMC checks CTLFC and quantifies over transition system instances. Existential TCMC allows us to check existential properties in Alloy.
- Checking up to the significant scope in TCMC provides a measure of confidence in the result independent of computing resources. The significant scope is a measure of transition system size, rather than path length.
- When a property does not hold, universal TCMC returns an instance that is a transition system. A transition system instance provides more inspectable information than a path; a violating (likely small) instance may include multiple paths that violate the property uncovering multiple bugs.

Table 2 is a summary of the comparison between scoped TCMC and BMC with the ordering module with respect to what we can conclude regarding the entire reachable state space.

Table 2: Deducing Complete Model Checking Results: Scoped TCMC vs. BMC

		Scoped TCMC		BMC using `ordering`	
Property		Pass	Fail	Pass	Fail
Safety		Ambiguous	Real Bug	Ambiguous	Real Bug
Finite Liveness	w/o dead-loop	Ambiguous	Real Bug	Real Pass	Ambiguous
	w/ dead-loop	Real Pass	Ambiguous		
Infinite Liveness		Ambiguous	Real Bug	*Cannot Express*	
Existential		Real Pass	Ambiguous	*Cannot Express*	

8.3 Fairness Constraints

Our Traffic Light Controller case study shows an example of the use of fairness constraints in TCMC. It also shows an application of the method described in (Vakili, 2016) to convert multiple fairness constraints to one.

Our model has three fairness constraints that ensure all directions at the three-way traffic light intersection (North, South and East) receive adequate green light time. The fair states satisfying each of these three constraints are described by the functions implemented in Lines 3–5 in Figure 24. The `fact fairness { ... }` in Lines 7–14 dictates the update of a counter attribute in `State` whenever a new type of fair state is encountered, and the counter

```
 1 open ctlfc[State]
 2 ...
 3 fun N_fair[]: State {State - (sensors.N_Sense & goes.N_Go)}
 4 fun S_fair[]: State {State - (sensors.S_Sense & goes.S_Go)}
 5 fun E_fair[]: State {State - (sensors.E_Sense & goes.E_Go)}
 6 // combines 3 fcs into 1 fc by checking that all 3 fcs occur infinitely
      often using a counter
 7 fact fairness {
 8   all s,s':State | s->s' in nextState implies (
 9     (s in N_fair[] and s.counter=f0) implies s'.counter=f1 else
10     (s in S_fair[] and s.counter=f1) implies s'.counter=f2 else
11     (s in E_fair[] and s.counter=f2) implies s'.counter=f3 else
12     s.counter=f3 implies s'.counter=f0 else
13     s'.counter=s.counter)
14 }
15 pred fair[s:State] {
16   s.counter = f3
17 }
18 ...
19 fact modelDefinition{
20   ...
21   all s:State | s in fc iff fair[s]
22 }
```

Fig. 24: Traffic Lights Control fairness constraints

is reset when all three types of fair states have occurred. The predicate `fair` (Lines 15–17) is true whenever a member from each of the three fair state sets has been encountered. We equate the set of accepted fair states in the `ctlfc` module, `fc`, to those satisfying `fair` (Line 21). Therefore, when model checking, the `ctlfc` module ensures that the `fair` predicate holds infinitely often in checked instances, thus, satisfying all three fairness constraints of the model.

9 Related Work

The `ordering` module of Alloy can be used for simple bounded model checking (BMC) (Biere et al, 1999). Cunha (Cunha, 2014) uses the `ordering` module for bounded model checking of LTL properties. Our approach supports more sophisticated temporal properties and provides some advantages over BMC as discussed in Section 8.2.

A declarative relational modelling language for transition systems has been proposed by Chang and Jackson (Chang and Jackson, 2006). They augment the traditional languages of model checkers with sets, relations and declarative constructs to specify a transition system. Their technique is not capable of model checking a declarative model with multiple instances of a transition system, and suffers from the state-space explosion problem.

B (Abrial, 1996) is a modelling language that has many similarities to Alloy. Models developed in B are called B *machines*, and the variables used to define the state space can be sets and relations. ProB (Leuschel and Butler, 2008) is a tool for analyzing finite B machines, in particular, model checking and automatic refinement checking of B machines. ProB provides LTL model checking support. LTL properties are checked by explicit-state search. Since each single state in a B machine represents some sets and relations, computing the set of the next states of a single state is computationally very costly. Several implementations of symbolic model checking algorithms (BMC, k-induction, IC3) for B machines are provided in (Krings and Leuschel, 2018), however, they cannot check all CTLFC properties, are iterative (meaning involve multiple runs of the solver), and suffer from solver performance constraints (as does TCMC).

The Abstract State Machine (ASM) method (Börger, 2005) is for high-level system design and analysis. The ASM method is used to specify an infinite transition system. Analysis techniques for the ASM method include theorem proving (Schellhorn and Ahrendt, 1997; Dold, 1998), and model checking (Del Castillo and Winter, 2000), which consists of translating an ASM to SMV by fixing the size of the scopes in the ASM.

TLA+ (Yu et al, 1999) (with the TLC model-checker) checks behavioural models for temporal properties. TLC supports unbounded model checking of a subset of LTL formulas using explicit-state model checking. TCMC is a symbolic approach to model checking.

Electrum (Macedo et al, 2016) is an extension of Alloy that incorporates features from both Alloy and TLA+. It supports finite state model checking of LTL properties. Electrum's model checking mechanism requires modelling over a time dimension, which adds complexity to the model checking problem. DynAlloy (Frias et al, 2005), along with the DynAlloy Analyzer (Regis et al, 2017), is a set of extensions to Alloy for describing and analyzing dynamic properties of systems using actions. DynAlloy does not support model checking of temporal properties, such as CTLFC.

10 Conclusion

We have presented transitive-closure-based model checking (TCMC): a method for encoding every CTLFC formula in first-order logic plus transitive closure. Compared to Immerman and Vardi (Immerman and Vardi, 1997), our encoding does not increase the size of the model, and the translation algorithm is linear with respect to the size of the CTLFC formula. We have used TCMC to model check transition systems in Alloy by using the constraint solver of the Alloy Analyzer up to similar scopes as are used to check non-temporal properties. We introduced style guidelines for modelling transition systems natively in Alloy (i.e. without any extensions to Alloy). We tackled the problem of spurious instances of transition systems through significance axioms, which give us a measure of whether we are checking instances that are large enough to be interesting. We describe a methodology for scoped TCMC, which uses the significance axioms and describes what the scoped results mean for the complete transition system.

We are working on ways to add common modelling abstractions, such as state hierarchy, to declarative models of transition systems (Serna et al, 2017). In the future, we plan to explore the use of TCMC for declarative models that define more than one transition system. We are also exploring methods to extract paths or other useful information from the (usually small) TS instances returned by TCMC (Kember et al, 2019). We also want to compare our approach to model checking using Alloy* (Milicevic et al, 2015) (which has second-order quantification) and investigate methods for improving the scalability of TCMC.

11 Acknowledgments

We thank Amin Bandali, George Gao, Eunsuk Kang, Mitchell Kember, Joseph Poremba, Khadija Tariq, and Lynn Tran for their help in discussions regarding Alloy. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

Abrial JR (1996) The B Book: Assigning Programs to Meanings. Cambridge University Press

Biere A, Cimatti A, Clarke E, Zhu Y (1999) Symbolic model checking without BDDs. In: Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, pp 193–207

Börger E (2005) The ASM Method for System Design and Analysis. A Tutorial Introduction. In: Frontiers of Combining Systems, Lecture Notes In Computer Science, vol 3717, Springer, pp 264–283

Bradley AR (2011) SAT-based model checking without unrolling. In: International Conference on Verification, Model Checking, and Abstract Interpretation, Springer, Lecture Notes In Computer Science, vol 6538, pp 70–87

Chang FSH, Jackson D (2006) Symbolic Model Checking of Declarative Relational Models. In: International Conference on Software Engineering, ACM, pp 312–320

Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Computer Aided Verification, Lecture Notes In Computer Science, vol 2404, Springer, pp 241–268

Clarke E, Grumberg O, Peled DA (1999) Model Checking. MIT Press

Clarke EM, Grumberg O, Hamaguchi K (1997) Another Look at LTL Model Checking. Formal Methods in System Design 10:47–71

Cunha A (2014) Bounded model checking of temporal formulas with Alloy. In: International Conference on Abstract State Machines, Alloy, B, VDM, and Z, Springer Berlin Heidelberg, pp 303–308

Del Castillo G, Winter K (2000) Model Checking Support for the ASM High-Level Language. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes In Computer Science, vol 1785, Springer, pp 331–346

Dijkstra EW (1975) Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8):453–457

Dold A (1998) A Formal Representation of Abstract State Machines Using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm

Eén N, Sörensson N (2004) An Extensible SAT-solver. In: Theory and Applications of Satisfiability Testing, Lecture Notes In Computer Science, vol 2919, Springer, pp 333–336

Farheen S (2018) Improvements to transitive-closure-based model checking in Alloy. MMath thesis, University of Waterloo, David R. Cheriton School of Computer Science

Frias MF, Galeotti JP, López Pombo CG, Aguirre NM (2005) DynAlloy: Upgrading Alloy with Actions. In: International Conference on Software Engineering, ACM, pp 442–451

Grumberg O, Long DE (1991) Model checking and modular verification. In: Proccedings of 2nd International Conference on Concurrency Theory, Springer, Lecture Notes In Computer Science, vol 527, pp 250–265

Immerman N, Vardi M (1997) Model Checking and Transitive-Closure Logic. In: Computer-Aided Verification, Lecture Notes In Computer Science, vol 1254, Springer, pp 291–302

International Organisation for Standardization (2000) Information Technology Z Formal Specification Notation Syntax, Type System and Semantics

Jackson D (2002) Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology 11(2):256–290

Jackson D (2006) Software Abstractions - Logic, Language, and Analysis. MIT Press

Kember M, Tran L, Gao G, Day NA (2019) Extracting counterexamples from transitive-closure-based model checking. In: Workshop on Modelling in Software Engineering (MISE) @ International Conference on Software Engineering (ICSE), ACM, pp 47–54

Krings S, Leuschel M (2018) Proof assisted bounded and unbounded symbolic model checking of software and system models. Science of Computer Programming 15:41–63

Leuschel M, Butler M (2008) ProB : an automated analysis toolset for the B method. International Journal on Software Tools for Technology Transfer 10:185–203

Macedo N, Brunel J, Chemouil D, Cunha A, Kuperberg D (2016) Lightweight specification and analysis of dynamic systems with rich configurations. In: Foundations of Software Engineering, ACM, pp 373–383

McMillan K (1992) Symbolic model checking: An approach to the state explosion problem. PhD thesis, Pittsburgh, PA, USA

Milicevic A, Near JP, Kang E, Jackson D (2015) Alloy*: A general-purpose higher-order relational constraint solver. In: International Conference on Software Engineering, IEEE, vol 1, pp 609–619

Nissanke N (1999) Formal Specification: Techniques and Applications, 1st edn. Springer-Verlag

Plath M, Ryan M (2001) Feature integration using a feature construct. Science of Computer Programming 41(1):53–84

Regis G, Cornejo C, Gutiérrez Brida S, Politano M, Raverta F, Ponzio P, Aguirre N, Galeotti JP, Frias M (2017) DynAlloy analyzer: A tool for the specification and analysis of alloy models with dynamic behaviour. In: Foundations of Software Engineering, ACM, pp 969–973

Schellhorn G, Ahrendt W (1997) Reasoning about Abstract State Machines: The WAM Case Study. Journal of Universal Computer Science 3(4):377–413

Selic B (2007) From Model-Driven Development to Model-Driven Engineering. In: Euromicro Conference on Real-Time Systems, IEEE Computer Society

Serna J, Day NA, Farheen S (2017) DASH: A new language for declarative behavioural requirements with control state hierarchy. In: International Workshop on Model-Driven Requirements Engineering (MoDRE) @ IEEE International Requirements Engineering Conference (RE), pp 64–68

Vakili A (2016) Temporal logic model checking as automated theorem proving. PhD thesis, University of Waterloo, David R. Cheriton School of Computer Science

Vakili A, Day NA (2012) Temporal Model Checking in Alloy. In: International Conference on Abstract State Machines, Alloy, B, VDM, and Z, Springer, Lecture Notes In Computer Science, vol 7316, pp 150–163

Vardi MY, Wolper P (1994) Reasoning about infinite computations. Information and Computation 115:1–37

Yu Y, Manolios P, Lamport L (1999) Model checking TLA+ specifications. In: IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Springer Verlag, pp 54–66