Temporal Logic Model Checking in Alloy

Amirhossein Vakili and Nancy A. Day

Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
{avakili,nday}@uwaterloo.ca

Abstract. The declarative and relational aspects of Alloy make it a desirable language to use for high-level modeling of transition systems. However, currently, these models must be translated to another tool to carry out full temporal logic model checking. In this article, we show how a symbolic representation of the semantics of computational tree logic with fairness constraints (CTLFC) can be written in first-order logic with the transitive closure operator, and therefore described in Alloy. Using this encoding, the question of whether a declarative model of a transition system satisfies a temporal logic formula can be solved using the Alloy Analyzer directly. Also, since a declarative description of a model may actually represent a family of transition systems, we define two distinct model checking questions on this family (existential and universal model checking) and show how these properties can be evaluated in the Alloy Analyzer.

1 Introduction

The process of model-driven engineering [1] promises many benefits for 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 handle the lack of details available at this point in the project. Therefore, it must be able to express concepts that are abstract. 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 [2], B [3], Z [4], and ASMs [5] have many features to express abstract concepts (e.g., sets, relations, and functions) without sacrificing precision. Abstract models are usually declarative, meaning they are described as a set of constraints and do not necessarily have an operational semantics.

We are interested in the problem of analyzing temporal properties of declarative models. Chang and Jackson added finite relations and functions to a traditional state-based specification of a transition system, and developed a BDD-based model checker that analyzed these models against computational tree logic (CTL) specifications [6]. Del Castillo and Winter provided model checking support for a transition system specified as an Abstract State Machine (ASM) [5],

via the translation of a class of ASMs to SMV by restricting the range of functions to finite sets [7]. ProB [8] is a tool for analyzing finite B machines, in particular, simulation and model checking against linear temporal logic (LTL) specifications. Within Alloy, it is fairly straightforward to specify a transition relation and then iterate it to check bounded duration temporal properties [9]. None of these approaches allow us to check a full set of temporal properties against a fully declarative model of a transition system.

Describing the traditional representation of the semantics of a temporal logic with respect to a single transition system and state in first order logic 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 encoded the semantics of CTL and CTL* in first order logic with transitive closure FO(TC) [10]. Their semantics has the important property that the use of transitive closure replaces the need for quantification over the paths. Our first contribution in this paper is to show that a variant of Immerman and Vardi's encoding can be used to encode CTL with fairness constraints (CTLFC) in the Alloy language. We use this symbolic encoding to create a CTLFC model checker for finite scope declarative models of transition systems directly in the Alloy Analyzer. The model checking problem is turned into a constraint solving problem. Compared to Immerman and Vardi, our encoding is linear in the size of the model, whereas in theirs the encoding requires an exponential increase in the size of the model with respect to the size of the temporal logic formula. We validate the simplicity and utility of our approach through several examples of model checking temporal logic properties of declarative models in the Alloy Analyzer.

All related work described earlier on model checking declarative models has focused on a specification of a single transition relation (possibly with non-determinism) that uses declaratively constrained relations and functions to describe the system's behavior. In our work, the transition systems are specified in a fully declarative language, therefore, it may be the case that the model describes a family of transition relations. For example, the declarative specification "every state must reach a state that is reachable from itself" specifies more than one transition system even with 2 states: It is therefore possible to

consider multiple questions about how a family of transition relations satisfies a temporal property. In this paper, we consider two questions: 1) Universal model checking: Do all the transition relations in the family defined by the declarative model satisfy the temporal property? 2) Existential model checking: Is there a transition relation in the family defined by the declarative model that satisfies the temporal property?

These questions are important in different scenarios; e.g., the first question is relevant for black-box verification: verifying a system by using the specifications of its subsystems rather than their implementation details. In this case, a user is interested in checking whether the system satisfies certain properties no matter how the subsystems are implemented. The second question is relevant when details to be added in the future will constrain the transition relation of interest. In this case, a user needs to know whether the abstract model can be extended into a more detailed model that satisfies the property.

Our second contribution in this paper is to show that these two distinct questions can be described as consistency problems in the Alloy Analyzer for finite scope declarative models. We show several examples that demonstrate the relevance of these two questions for abstract modeling.

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 [11]. A transition system is a finite directed graph with a labeling 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 labeled 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 total binary relation over S; P is a finite set of atomic propositions; l, the labeling 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, $s_0 \rightarrow s_1 \rightarrow \dots$ such that $s_0 = s$ and $\forall i \geq 0 : \ \sigma(s_i, s_{i+1})$.

A specification is a set of temporal logic formulas. A temporal logic, such as CTL or CTLFC [11], has logical connectives for specifying properties over the computation paths of a transition system. Equation 1 represents 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:***

$$
\begin{array}{ll}
TS, s \models p & \textit{iff } p \in l(s) \\
TS, s \models \neg\varphi & \textit{iff } TS, s \not\models \varphi \\
TS, s \models \varphi \vee \psi & \textit{iff } TS, s \models \varphi \textit{ or } TS, s \models \psi \\
TS, s \models EX\varphi & \textit{iff } \exists s' \in \sigma(s) : TS, s' \models \varphi \\
TS, s \models EG\varphi & \textit{iff there exists a path starting at } s,\ s_0 \rightarrow s_1 \rightarrow \ldots,\ \textit{such} \\
& \quad \textit{that for all } i\textit{'s } TS, s_i \models \varphi. \\
TS, s \models \varphi EU \psi & \textit{iff there exist a } j \textit{ and a path, } s_0 \rightarrow s_1 \rightarrow \ldots,\ \textit{starting} \\
& \quad \textit{at } s \textit{ such that } TS, s_j \models \psi \textit{ and for all } i \textit{ less than } j \\
& \quad TS, s_i \models \varphi.
\end{array}
$$

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, $E_C G$. In this connective, C is a finite set of formulas, fairness constraints, which is used to define a *fair* computation path. The computation path $s_0 \rightarrow s_1 \rightarrow \ldots$ is fair with respect to $C = \{\psi_1, ..., \psi_n\}$ iff:

$$\forall \psi \in C : \ \{i \mid TS, s_i \models \psi\} \text{ is infinite.}$$

The semantics of CTLFC is same as Definition 2 along with the semantics of $E_C G$:

$$
\begin{array}{ll}
TS, s \models E_C G\varphi & \textit{iff there exists a fair computation path starting at } s, \\
& \quad s_0 \rightarrow s_1 \rightarrow \ldots,\ \textit{such that for all } i\textit{'s } TS, s_i \models \varphi.
\end{array}
$$

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

$$\sigma_X(s_1, s_2) \ \text{ iff } \ \sigma(s_1, s_2) \ \wedge \ s_1 \in X$$

In this article, ˆ denotes the transitive closure operator; for example, $\hat{}\sigma_X$ is the transitive closure of the relation σ_X. Notice that $\hat{}\sigma_X$ is $\hat{}(\sigma_X)$ and not $(\hat{}\sigma)_X$; in other words, the bounding operator has higher precedence over the transitive closure operator. Similarly, $*$ denotes the reflexive transitive closure operator.

2.2 Alloy

Alloy is a lightweight declarative relational modeling language that has static type checking [2]. The logic that Alloy provides for modeling is first-order logic with the transitive closure operator. 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 fix the size of the sets in the model to constant numbers and then, the Alloy Analyzer translates the model to a propositional

CNF formula, which is then 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. Figure 1 is a simple Alloy model of a transition system with transition relation `sigma`, and its only valid instance with four states is presented in Figure 2, where the vertexes represent the states, edges represent the transitions, and the labeling function is indicated by labeling the vertexes. In Figure 1, Lines 3-4, `S` is a set, `sigma` and `l` are functions that map each element of `S` to a subset of `S`, and to a subset of `P` respectively. Lines 1-2 are definitions of three sets, `P,p,q`, where `p` and `q` are singleton subsets of `P`. The keyword `abstract` is used to specify that every element of `P` belongs to one of its subsets. The result of this declaration is `P={p,q}`. The `fact` block is used to specify the constraints that need to be satisfied by the entities in this model.

```
1 abstract sig P {}
2 one sig p,q extends P {}
3 sig S { sigma: some S,
4  l: set P}
5 one sig S0 extends S {}
6 fact{ all s1,s2:S |
7  s1->s2 in sigma iff
8  (s1.l !in s2.l or s1=s2)
9  S0.l = P}
```

Fig. 1. A simple transition system in Alloy

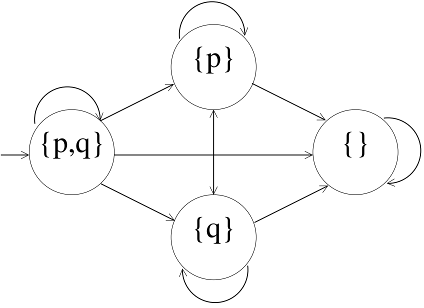

Fig. 2. A valid instance of Figure 1

3 Translating CTLFC to FO(TC)

Immerman and Vardi show how CTL and CTL* can be encoded in FO(TC) [10]. 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 hypothesize that a symbolic model checking algorithm based on their encoding may be faster than previous approaches [11], however, they do not provide any implementation of their idea.

In this section, we present our translation of CTLFC to FO(TC) with a similar approach to that of Immerman and Vardi. We chose CTLFC for three reasons: 1) unlike CTL*, the encoding of CTLFC in FO(TC) does not increase the size of a transition system, 2) it is more expressive than CTL, 3) LTL model checking can be reduced to CTLFC model checking[1] [12].

---

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

Our general idea for temporal logic model checking in Alloy is to use the (reflexive) transitive closure operator to specify the necessary and sufficient conditions for the set of states that satisfy a property. The closure operator is used to specify the reachability relation, which is not expressible in first-order logic. We define an operator, [ ], that takes a formula as input and outputs a symbolic representation of the set of states that satisfy the input formula. This operator is defined recursively in Definition 3. The key difference from the work of Immerman and Vardi is that each formula can be defined directly; support for CTL* would require the introduction of a new Boolean variable into the transition system for each sub-formula of the property.

Definition 3. ***Translation Operator*** *Let $TS = (S, S_0, \sigma, P, l)$ be a transition system and $C = \{\psi_1, \psi_2 \ldots, \psi_n\}$ a set of fairness constraints. The operator [ ] 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. $[E_C G\varphi] = \{s \in S|\ \exists t \in [\varphi], \exists u_1 \in [\psi_1], \exists u_2 \in [\psi_2], \ldots, \exists u_n \in [\psi_n]:$
   $*(\sigma_{[\varphi]})(s,t) \wedge \hat{}(\sigma_{[\varphi]})(t,t) \wedge$
   $*(\sigma_{[\varphi]})(t,u_1) \wedge *(\sigma_{[\varphi]})(u_1,u_2) \wedge \cdots \wedge *(\sigma_{[\varphi]})(u_{n-1},u_n) \wedge *(\sigma_{[\varphi]})(u_n,t)\}$

Theorem 1. *Let $TS = (S, S_0, \sigma, P, l)$ be a transition system, C a set of fairness constraints, φ a CTLFC formula, and [.] 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 definition of $[E_C G\varphi]$ is based on the model checking algorithm of $E_C G$ that finds the strongly connected components (SCCs) in a transition system. The state t in the definition of $[E_C G\varphi]$ is a state that belongs to a SCC that includes a state satisfying each fairness constraint ψ_i. Due to space restrictions, the details of the proof of this theorem are available on-line [2]. A simple yet useful corollary of Theorem 1 is the following:

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

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

[2] `http://www.cs.uwaterloo.ca/~avakili/projects/`

4 Model Checking in Alloy

In this section, we write Definition 3 using Alloy's syntax to create an operator that takes a CTLFC formula, a transition system, a set of fairness constraints and produces the set of states that satisfy the CTLFC formula. The operator, $SET(\varphi, TS, C)$, takes a CTLFC formula, φ, a transition system, TS, and a set of fairness constraints, C as input, and produces the subset of states that satisfies the CTLFC formula. The algorithm implemented by this operator visits each sub-formula only once, and as result, it is linear with respect to the size of the CTLFC formula and the fairness constraints. This algorithm is presented in Figure 4, and it uses three helper functions, `bound`, `id`, and `loop`. Each one is described using the equivalence symbol, $\equiv$, and their corresponding Alloy code is given in Figure 3. Intuitively, `bound[R,X]` is a subset of `R` when its domain is restricted to `X`; `id[X]` is the identity relation over `X`; `loop[R]` is a subset of states that are reachable from themselves through `R`.

`bound`$[R,X]\equiv$	`id`$[X]\equiv$	`loop`$[R]\equiv$
$\{(x,y)\in R \mid x\in X\}$	$\{(x,x)\mid x\in X\}$	$\{s\mid (s,s)\in \hat{} R\}$

```
fun bound[R:S->S,X:S]      fun id[X:S]               fun loop[R: S->S]
:S->S{ X <: R}             :S->S{bound[iden,X]}      :S{S.(^R & iden)}
```

Fig. 3. Helper functions

$\underline{SET(\varphi, TS, C)}$:
case φ of
1) p -> $l.p$
2) $\neg\varphi$ -> S - $SET(\varphi,TS,C)$
3) $\varphi \vee \psi$ -> $SET(\varphi,TS,C)$ + $SET(\psi,TS,C)$
4) $EX\varphi$ -> $\sigma.SET(\varphi,TS,C)$
5) $\varphi EU \psi$ -> `(*bound[`σ`,`$SET(\varphi,TS,C)$`])`.$SET(\psi,TS,C)$
6) $EG\varphi$ -> `let R=bound[`σ`,`$SET(\varphi,TS,C)$`] | (*R).loop[R]`
7) $E_C G\varphi$ -> `let R=bound[`σ`,`$SET(\varphi,TS,C)$`],`
`ids1=id[`$SET(\psi_1,TS,C)$`], ..., idsn=id[`$SET(\psi_n,TS,C)$`] |`
`(*R).(loop[R]&loop[(*R).ids1.(*R).ids2.(*R). ... .(*R).idsn.(*R)])`

Fig. 4. Translation algorithm where $C = \{\psi_1, \ldots, \psi_n\}$

According to Corollary 1, in order to check if $TS \models \varphi$ with respect to the fairness constraints C in Alloy, we add the constraint in Equation 2 to the model as an `assertion` and `check` its validity.

$$S_0 \texttt{ in } SET(\varphi, TS, C) \tag{2}$$

Example 1. In order to check whether the Alloy model of Figure 1 satisfies $E_C Gp$, where $C = \{q, \neg q\}$, the Alloy code of Figure 5 is added to the model and the Alloy Analyzer is used to check for the validity of the assertion. In Figure 1, the definition of operator SET has been expanded to create the assertion.

```
1 assert CTLFC_MC_1{
2  let R=bound[sigma,l.p], ids1=id[l.q], ids2=id[S-l.q]|
3   S0 in (*R).(loop[R]&loop[(*R).ids1.(*R).ids2.(*R)])}
4 check CTLFC_MC_1 for exactly 4 S
```

Fig. 5. Model checking $E_C Gp$ where $C = \{q, \neg q\}$

```
1 assert CTLFC_MC_2{let R=bound[sigma,l.p]| S0 in (*R).(l.q)}
2 check CTLFC_MC_2 for exactly 4 S
```

Fig. 6. Model checking $pEUq$

Since this property is not satisfied, the Alloy Analyzer outputs `Counterexample found. Assertion is invalid`; similarly, for model checking of the Alloy model of Figure 1 against $pEUq$, the Alloy code of Figure 6 is used, and the Alloy Analyzer outputs `No counterexample found. Assertion may be valid.`

To make model checking in Alloy easy and accessible, we wrote parameterized Alloy modules so that users can import the definitions of the temporal logic operators. The parameter of these modules is the set of states. We have two modules, `ctl` for model checking CTL, and `ctlfc` for model checking CTLFC. Since the number of fairness constraints is not fixed, a user needs to change some parts of the `ctlfc` module, which can be done easily. The universal path quantifiers, `AX, AG, AU, ACG`, have been defined in terms of the existential operators. The following example uses the `ctlfc` module. These module are available on-line[3].

Example 2. Figure 7 is an Alloy model of a binary counter. The `State` space of this transition system is defined by 3 `BIN`ary variables, `input`, `d1`, and `d2`. Lines 3-5 define the set `BIN={ZERO,ONE}` and the negation of a bit `function`, `comp`. This model has two fairness constraints, $C = \{d_1, \neg d_2\}$, which is modeled in Line 7. Lines 8-9 state that if the variables of two states are equal, then those states are equal. Line 10 defines the `initial State`s of the system, and Lines 11-14 define the transition relation, `nextState`. Since we are interested in CTLFC model checking, the `ctlfc` module is imported having the set of `State`s as its parameter, Line 2. Suppose, we want to check that whenever $d1$ is zero, it will eventually become one. By using the temporal connectives of CTLFC from the `ctlfc` module, the property can be written as in Lines 15-16, and the Alloy Analyzer concludes the model checking problem, `CTLFC_MC` is valid.

5 Model Checking Classes of Transition Systems

A satisfiable Alloy model may have more than one valid instance. This is common when constraints that are used to model a system are not strong enough to

[3] `http://www.cs.uwaterloo.ca/~avakili/projects/`

```
1 module binary_counter
2 open temporal_logics/ctlfc[State]
3 abstract sig BIN{}
4 one sig ZERO, ONE extends BIN {}
5 fun comp[b:one BIN]:one BIN{ b=ZERO implies ONE else ZERO }
6 sig State{ input, d1, d2: BIN }
7 fact { fc1=d1.ONE and fc2=d2.ZERO
8   all s,s':State|s.input=s'.input and s.d1=s'.d1 and
9     s.d2=s'.d2 implies s=s'
10   initialState = (d1.ZERO & d2.ZERO)
11   all s,s':State| s' in nextState[s] iff
12     s.input=ZERO implies (s'.d1=s.d1 and s'.d2=s.d2)
13     else(s'.d1=comp[s.d1] and
14     (s.d1=ZERO implies s'.d2=s.d2 else s'.d2=comp[s.d2]))}
15 assert
16   MC{CTLFC_MC[ACG[implies_ctlfc[d1.ZERO,ACF[d1.ONE]]]]}
17 check MC
```

Fig. 7. Model checking $A_C G(\neg d_1 = 0 \rightarrow A_C F\ d_1 = 1)$, where $C = \{d_1, \neg d_2\}$, for a binary counter.

uniquely identify the transition system; for example, Figure 8 is an Alloy model of a transition system that has more than one valid instance, namely those in Figures 9-10. Each valid interpretation represents a different transition system; as a result, the Alloy model represents a *class* of transition systems rather than a *single* system.

This observation is formalized as follows: the model $\mathfrak{D}$ represents a class of transition systems, $CTS(\mathfrak{D})$:

$$CTS(\mathfrak{D}) = \{TS \mid TS \text{ is a transition system satisfying the constraints in } \mathfrak{D}\} \tag{3}$$

In Equation 3, where the model $\mathfrak{D}$ is considered as a set of transition systems, two questions can be studied: 1) *do all transition systems* in $CTS(\mathfrak{D})$ satisfy the specifications? 2) *is there a transition system* in $CTS(\mathfrak{D})$ that satisfies the specifications? We define two model checking problems for a class of transition systems that correspond to these questions:

Definition 4. ***Universal Model Checking:*** *The universal model checking of the declarative model $\mathfrak{D}$ and the temporal property φ is defined as checking whether all valid instances of $\mathfrak{D}$ satisfy φ:*

$$\mathfrak{D} \textit{ universally satisfies } \varphi \textit{ iff } \quad \forall\, TS \in CTS(\mathfrak{D}) : TS \models \varphi$$

We use $\mathfrak{D} \models_\forall \varphi$ to denote that the declarative model $\mathfrak{D}$ universally satisfies φ.

Definition 5. ***Existential Model Checking:*** *The existential model checking of the declarative model $\mathfrak{D}$ and the temporal property φ is defined as checking*

```
1 sig S { sigma: some S,  l: set P}
2 abstract sig P {}
3 one sig p,q extends P {}
4 one sig S0 extends S {}
5 fact{ all s1,s2:S | s1.l !in s2.l implies s1->s2 in sigma
6  S0.l = P}
```

Fig. 8. A simple transition system in Alloy

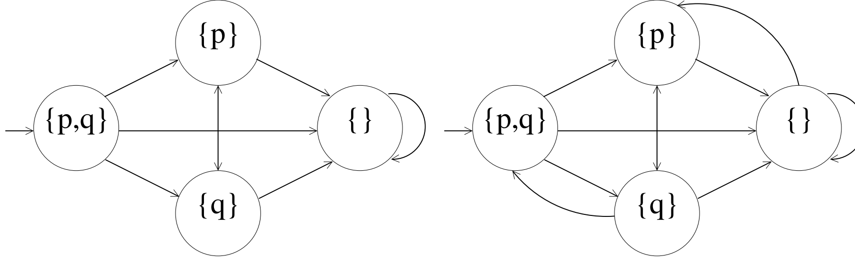

Fig. 9. A valid instance of Figure 8 that does not satisfy EGp

Fig. 10. A valid instance of Figure 8 that satisfies $pEUq$

whether there exists a valid instance of $\mathfrak{D}$ *that satisfies* φ*:*

$$\mathfrak{D} \textit{ existentially satisfies } \varphi \textit{ iff } \quad \exists\, TS \in CTS(\mathfrak{D}) : TS \models \varphi$$

We use $\mathfrak{D} \models_{\exists} \varphi$ *to denote that the declarative model* $\mathfrak{D}$ *existentially satisfies* φ*.*

The model finding capability of the Alloy Analyzer can be exploited to solve the universal and existential model checking. The model checking approach described in Section 4 solves universal model checking: since we add the constraint of Equation 2 as an assertion, the Alloy Analyzer checks whether all valid instances of the model, which in this case are transition systems, satisfy the assertion, which is the CTLFC property. If the model $\mathfrak{D}$ universally satisfies φ ($\mathfrak{D} \models_{\forall} \varphi$), the Alloy Analyzer outputs `valid`; otherwise, a valid interpretation of $\mathfrak{D}$ such as TS ($TS \in CTS(\mathfrak{D})$) that does not satisfy the constraint of Equation 2 ($TS \not\models \varphi$) is given as a counterexample.

For existential model checking of model $\mathfrak{D}$ against the CTLFC formula φ, the constraint of Equation 2 is added to the model as a `pred`icate and the Alloy Analyzer is used to check for the consistency of the predicate with the model. If the predicate is consistent with the model, the Alloy Analyzer outputs a valid interpretation of $\mathfrak{D}$ such as TS ($TS \in CTS(\mathfrak{D})$) that satisfies the constraint of Equation 2 ($TS \models \varphi$); otherwise, the output of the Alloy Analyzer is `inconsistent predicate`, which means $\mathfrak{D} \not\models_{\exists} \varphi$.

Example 3. In order to check whether the class of transition systems defined by the Alloy model of Figure 8 universally satisfies EGp, the Alloy code of Figure 11 is added to the model and the Alloy Analyzer is used to check for the

validity of the assertion. Since this property is not satisfied, the Alloy Analyzer outputs the instance of Figure 9 as a counterexample; similarly, for existential model checking of the Alloy model of Figure 8 against $pEUq$, the Alloy code of Figure 12 is used, and the Alloy Analyzer outputs the transition system of Figure 10 as a valid instance.

```
1 assert MC1{
2   let R=bound[sigma,l.p]|
3     S0 in (*R).(loop[R])}
4 check MC1 for exactly 4 S
```

Fig. 11. Universal model checking of EGp

```
1 pred MC2[]{
2   let R=bound[sigma,l.p]|
3     S0 in (*R).(l.q)}
4 run MC2 for exactly 4 S
```

Fig. 12. Existential model checking of $pEUq$

6 Experimental Validation

We completed several examples to show that our method makes it possible to check CTLFC temporal logic specifications of declarative models in the Alloy Analyzer, thereby validating the simplicity and utility of our approach. We used four examples from different domains: 1) the semantics of untyped lambda calculus [13], 2) the address book from Jackson [9], 3) feature interaction (FI) between call-waiting and call-forwarding, 4) model checking a traffic light controller [14]. These models satisfy their temporal specifications. Our parameterized Alloy modules for CTL and CTLFC hide the details of model checking in Alloy for a user, so that temporal specifications can be added to models smoothly. These models are available on-line. We used the Alloy Analyzer 4.2 along with the MiniSat SAT-solver [15]. The experiments were run on an Intel Core 2 Due 2.40 GHz machine running Ubuntu 10.04 with up to 3G of user-space memory.

Table 1 presents data on the types of properties, type of model checking (universal/existential), scope size, number of signatures, number of relations, and the Alloy Analyzer time to check the property. With respect to scalability, we found that temporal specifications can be analyzed up to the size of the scopes that non-temporal specifications are often analyzed in Alloy. Thus, our method is immediately valuable to those who use Alloy for modelling and analysis now relying on the *Small Scope Hypothesis [9]*. These models are not as large as those that can be checked using a model checker such as SMV [14], however, the declarative and relational aspects of Alloy have significant advantages for creating abstract, concise models, and we now provide the ability to check temporal logic specifications directly on small scopes of these models.

Furthermore, the untyped λ-calculus example shows the value of the existential model checking question. We used existential model checking to generate a λ-term that does not have a normal form, $(\lambda x.xx)(\lambda x.xx)$, and a term

Table 1. Experimental results. MC: Model Checking, NS: Number of Signatures, NR: Number of Relations, SS: Scope Size, min: minute, sec: seconds

Untyped λ-calculus		Address Book		Feature Interaction		Traffic Light Controller	
NS:6, NR:10		NS:5, NR:3		NS:6, NR:10		NS:13, NR:4	
SS	Time	SS	Time	SS	Time	SS	Time
7	8.22 sec	14	1 min 14 sec	10	14.28 sec	7	4.71 sec
		15	2 min 57 sec	11	2 min 7.6 sec	8	36.81 sec
		16	9 min 15 sec	12	20 min 51 sec	9	12 min 42 sec
		17	13 min 43 sec	13	> 1 hour	10	> 1 hour
Safety, Liveness		Safety		Safety		Safety with fairness	
Existential MC		Universal MC		Universal MC		Universal MC	

that has a normal form but not necessarily every reduction path terminates, $(\lambda x.(\lambda x.xx))((\lambda x.xx)(\lambda x.xx))$. Since, a result was found for the scope 7, there was no need to do existential model checking for higher scopes. As this example suggests, one way of using existential model checking is to generate interesting instances. In general, existential model checking can help a user to have a better understanding about a declarative model of a transition system by checking the *existence* of specific instances; in other words, existential model checking can be considered as an approach for "simulating" a declarative transition system.

7 Related Work

The `ordering` module of Alloy can be used for bounded model checking of safety properties. This approach does not support model checking liveness properties or even safety with fairness constraints. Our approach, which is available as `ctlfc` and `ctl` modules in Alloy, supports much more sophisticated temporal properties.

A declarative relational modeling language for transition systems has been proposed by Chang and Jackson [6]. They augment the traditional languages of model checkers by sets and relations and declarative constructs to specify a transition system. Their technique is not capable of model checking a class of models, and suffers from the state-space explosion problem.

B [3] is a modeling language that has many similarities with Alloy. Models developed in B are called B *machines*, and the variables used to define the state space can be sets and relations. ProB [8] is a tool for analyzing finite B machines, in particular, model checking and automatic refinement checking of B machines. ProB provides LTL model checking. LTL properties are checked by explicit state-space 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. ProB also does not provide model checking for a class of transition systems.

The Abstract State Machine (ASM) method [5] 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 [16, 17], and model checking [7], which consists of translating an ASM to SMV by fixing the size of the scopes in the ASM.

DynAlloy is an extension to Alloy for describing the dynamic properties of systems by using actions [18]. It provides partial correctness analysis of DynAlloy models by using the Alloy Analyzer. Our work is concerned with transition systems and temporal properties.

Modal transition systems (MTSs) are generalized transition systems that are mostly used for verification of complex systems by combining over- and under-approximation for abstraction [19]. In an MTS, a user needs to specify the "must" transitions, those that are part of a system, and the "may" transitions, the ones that may become part of the transition system. Determining "must" transitions requires some analysis of the specification, and discovering how the system must work. Our approach does not need such an analysis and the systems can be completely declarative.

8 Conclusion

We have shown that every CTLFC formula can be encoded in first-order logic plus transitive closure using a similar approach to Immerman and Vardi [10]. 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 this translation to model check transition systems in Alloy by using the constraint solver of the Alloy Analyzer to similar scopes as are used to check non-temporal properties.

When an Alloy model of a transition system has more than one valid instance, it represents a class of transition systems. We have defined two model checking problems concerning a class of transition systems: 1) universal model checking (Definition 4) 2) existential model checking (Definition 5); further, we have used our encoding of CTLFC in Alloy, and the capability of the Alloy Analyzer in valid instance finding to solve the model checking problems that we have defined. The scalability of our approach is dominated by the SAT-solver's capability in solving constraints.

The declarative aspects of Alloy make it a very suitable language for modeling structural aspects of product. We are interested in provided more language support for specifying declarative models of transition systems to help with the readability of these models.

The witness (or counterexample) that is produced for existential (universal) model checking is a transition system; by adding labels for each sub-formula of the specification to states, a user can see why a witness (or counter-example) satisfies (does not satisfy) the specification. Developing a post-processor that takes a transition system generates SMV style counterexamples, which are computation paths, will make our approach more accessible to a non-expert user.

Even though, we do not restrict the length of computation paths in our approach as is done in bounded model checking [20], bounding the signatures of an Alloy model results in bounding the states. Bounding the signatures differently may result in discovering different errors. The relationship between the system and how its signatures are bounded can be studied to make this approach more effective for declarative models.

References

1. Selic, B.: From Model-Driven Development to Model-Driven Engineering. In: ECRTS, IEEE Computer Society (2007)
2. Jackson, D.: Alloy: a lightweight object modelling notation. ACM TOSEM **11**(2) (2002) 256–290
3. Abrial, J.R.: The B Book: Assigning Programs to Meanings. Cambridge University Press (August 1996)
4. International Organisation for Standardization: Information Technology Z Formal Specification Notation Syntax, Type System and Semantics. (2000)
5. Börger, E.: The ASM Method for System Design and Analysis. A Tutorial Introduction. In: Frontiers of Combining Systems. Volume 3717 of LNCS. Springer (2005) 264–283
6. Chang, F.S.H., Jackson, D.: Symbolic Model Checking of Declarative Relational Models. In: ICSE '06. (May 2006) 312–320
7. Del Castillo, G., Winter, K.: Model Checking Support for the ASM High-Level Language. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 1785 of LNCS. Springer (2000) 331–346
8. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: FME 2003: Formal Methods. Volume 2805 of LNCS. Springer (2003) 855–874
9. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press (2006)
10. Immerman, N., Vardi, M.: Model Checking and Transitive-Closure Logic. In: CAV. Volume 1254 of LNCS. Springer (1997) 291–302
11. Clarke, E., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
12. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another Look at LTL Model Checking. Formal Methods in System Design **10** (1997) 47–71
13. J.R. Hindley, J.P. Seldin: An Introduction to Combinators and the λ-calculus. 2 edn. Cambridge University Press (2008)
14. McMillan, K.L.: The SMV system (November 06 1992)
15. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Theory and Applications of Satisfiability Testing. Volume 2919 of LNCS. Springer (2004) 333–336
16. Schellhorn, G., Ahrendt, W.: Reasoning about Abstract State Machines: The WAM Case Study. Journal of Universal Computer Science **3**(4) (1997) 377–413
17. Dold, A.: A Formal Representation of Abstract State Machines Using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm (July 1998)
18. Frias, M.F., Galeotti, J.P., López Pombo, C.G., Aguirre, N.M.: DynAlloy: Upgrading Alloy with Actions. In: Proceedings of ICSE'05, ACM (2005) 442–451
19. Huth, M., Jagadeesan, R., Schmidt, D.: Modal transition systems: A foundation for three-valued program analysis. In: Programming Languages and Systems. Volume 2028 of LNCS. Springer (2001) 155–169
20. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: TACAS. Volume 1579 of LNCS. Springer (1999) 193–207