Using Model Checking to Analyze Static Properties of Declarative Models: Extended Version*

Amirhossein Vakili and Nancy A. Day
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
{avakili,nday}@cs.uwaterloo.ca

August, 2011

Technical Report: CS-2011-22

Abstract

We show how static properties of declarative models can be efficiently analyzed in a symbolic model checker; in particular, we use Cadence SMV to analyze Alloy models by translating Alloy to SMV. The computational paths of the SMV models represent interpretations of the Alloy models. The produced SMV model satisfies its LTL specifications if and only if the original Alloy model is inconsistent with respect to its finite scopes; counterexamples produced by the model checker are valid instances of the Alloy model. Our experiments show that the translation of many frequently used constructs of Alloy to SMV results in optimized models such that their analysis in SMV is much faster than in the Alloy Analyzer. Model checking is faster than SAT solving for static problems when an interpretation can be eliminated by early decisions in the model checking search.

1 Introduction

In model-driven engineering, modeling is the first step in creating a computer-based system. Models guide developers throughout the production of a system. Errors in end products that are due to incorrect models are very costly to repair. Therefore, it is particularly important that a model is correct, meaning that it satisfies both general (e.g., consistency and completeness) and particular specifications. Designers and developers can use various analysis techniques to verify and test different types of properties of models.

Properties of models that are independent of time are called *static* properties and time-dependent ones are called *dynamic* or *temporal* properties. For example, "a student has at most two supervisors" is a static property whereas "whenever an interrupt occurs it must be answered at some point" is a dynamic one.

Alloy is a light-weight modeling language that uses predicate and relational calculus to specify declarative models and properties [2, 3]. It has an analyzer, the Alloy Analyzer, which supports finite scope

*A short version of this paper is published in the proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), November 2011 [1].

analysis. By limiting the sizes of entities in the model to finite numbers, the Alloy Analyzer translates the model into a CNF formula, which is then checked for satisfiability by a SAT-solver: the CNF formula is satisfiable if and only if the original Alloy model is consistent with respect to its finite scope. Alloy's simplicity, precision, and analyzer have made it one the most popular formal modeling languages for expressing and analyzing structures and static properties of models (e.g., [4, 5, 6, 7, 8]).

As the sizes of the scope in an Alloy model increase, the Alloy Analyzer is often unable to complete the analysis. The cause of failure is either that the conversion of the model to a CNF formula fails or that its SAT-solver cannot check the satisfiability of the produced CNF formula.

We propose a new approach to analyze Alloy models: we create an equivalent transition system in which a finite satisfying interpretation is iteratively created over time as a computational path of the transition system. Constraints on Alloy models are encoded as linear temporal logic (LTL) formulas [9] over computational paths of the transition system. Then, we use a BDD-based symbolic model checker [10] as a decision procedure. The transition system satisfies its LTL specifications if and only if the original Alloy model is consistent with respect to its finite scope; counterexamples produced by the model checker are directly mapped to valid instances of the Alloy model. We present a translation algorithm from Alloy models to the SMV language [11] and use the Cadence SMV model checker [11, 12], but our approach could be used with any LTL model checker.

In our approach, because a satisfying interpretation is found step-by-step as a computational path of the transition system, non-satisfying interpretations are also discarded step-by-step. Therefore, in an appropriately structured transition system, all interpretations that share a common part can be eliminated from consideration together via a form of partial evaluation of the model. Through experiments, we have found that for many frequently used Alloy constructs, it is possible to construct a transition system and LTL properties for which model checking is a faster and more efficient method for analyzing larger scope than the Alloy Analyzer.

To the best of our knowledge, using counterexamples discovered via model checking as satisfying instances of a static property is a completely new approach. Model checking has been used for analyzing and specifying high-level models [13, 14, 15, 16, 17]: however, none of these approaches analyze static properties of declarative models. Chang and Jackson's work, [13], on model checking a declarative relational language does not support static properties.

The next section is an overview of Alloy and SMV. In Section 3, we present the general translation algorithm. Section 4 discusses techniques for optimizing the translation of some frequently used Alloy constructs that lead to better performance from our model checking approach. In Section 5, we present our experimental results, and Section 6 discusses related work. Finally, Section 7 concludes the paper and includes future work.

2 Background

In this section, we briefly cover some features of Alloy and SMV needed to understand our approach.

2.1 Overview of Alloy

Alloy provides a combination of predicate and relational calculus to express constraints on models [3]. The Alloy book [3] provides a complete overview of Alloy's syntax and semantics.

Because Alloy has a transitive closure operator, its logic is more expressive than first-order logic. An Alloy model consists of two parts: (1) a set of declarations that specifies the entities of the model, (2) constraints on these entities. The entities are represented as sets, relations, and functions.

Example 1. Figure 1 is a simple Alloy model for an abstract memory. The keyword `sig` is used to declare sets, relations and functions in a model. The model uses sets `Data` and `Addr` to represent the set

```
1 sig Data {}
2 sig Addr {content : lone Data}
```

Figure 1: Example 1: Simple Alloy model for memory

```
1 sig Data {}
2 one sig d extends Data {}
3 sig Addr {content,content' : lone Data}
4 one sig a extends Addr {}
5 fact{content' = content ++ a->d}
6
7 pred show[]{}
8 run show for exactly 3 Addr,
9    exactly 4 Data
```

Figure 2: Example 2: Simple Alloy model of memory write

of data and addresses in the model respectively. In Figure 1, `content` is a function from `Addr` to `Data` representing the content of each memory address. The keyword `lone` in the definition of `content` specifies that `content` maps each element of `Addr` to *at most one* element of `Data`; in other words, `content` is a partial function.

The general form of the declaration of a relation of type $A_1 \times A_2 \times \ldots \times A_n$ is:

`sig A1 {rel : set A2 -> .. -> set An}`

The general form of the declaration of a function of type $A \rightarrow B$ in Alloy is:

`sig A {f : [lone] B}`

The keyword `lone` is used to specify that the function `f` is a partial function from `A` to `B`; if `lone` is not used then `f` is a total function.

Example 2. To model the operation of writing into memory, we add another partial function, `content'` to the Alloy model of Figure 1 to represent the state of the memory after writing the data `d` to address `a`. Figure 2 is the resulting Alloy model.

In general, `one sig s extends S {}` is used to specify that `s` is a singleton subset of `S`; in other words, `s` is a member of `S`. This construct is mainly used to specify a particular element of a set so that it can be used to express certain properties. If the keyword `one` is omitted, `s` would be a subset of `S` that can have any cardinality with respect to the size of `S`. The keyword `in` can also be used to specify subset relations between sets instead of `extends`; the difference is that subsets defined by using `extends` are mutually disjoint.

The `fact` block is used to express constraints on the model that need to be satisfied by all valid instances. Figure 3 lists the core syntax of Alloy for specifying constraints. Expressions, *expr* in Figure 3, are interpreted as sets in Alloy. In Example 2, the `fact` block is used to specify that `content'` is equal to `content` overridden by `a->d`.

In the presentation of our algorithm, we show the translation steps for Alloy models without some of the syntactic sugar available in the Alloy language (e.g., quantifiers `no`,`one`, ...). Since the elimination of those constructs does not reduce the expressive power of Alloy, our method is complete for all Alloy constructs. We describe our translation algorithm for single file Alloy models with one fact block.

formula	::=	expr **in** expr	subset
	::=	**not** formula | formula **and** formula	negation, conjunction
	::=	**all** var : expr **|** formula	universal quantification
expr	::=	var | expr binop expr | uniop expr	
binop	::=	+ | & | - | . | ++ | ->	union, intersection, difference, join, override, Cartesian product
uniop	::=	~ | ^	transpose, transitive closure

Figure 3: Alloy's core syntax for specifying constraints

```
 1  module main(I){
 2    input I : boolean;
 3    c0,c1 : boolean;
 4    init(c0) := 0;
 5    init(c1) := 0;
 6    if (I)
 7       next(c0) := ~c0;    --  ~ is not
 8    else
 9       next(c0) := c0;
10    if (I & c0)
11       next(c1) := ~c1;
12    else
13       next(c1) := c1;
14 }
```

Figure 4: Example 3: simple SMV model of a 2 digit counter

In order to use the Alloy Analyzer, some commands need to be added to the model. These commands specify the size of each set in the model. Since the size of each set is a finite number, the number of interpretations for certain scopes is also finite; therefore, the problem of checking the consistency of a model with respect to its finite scope is decidable. In Example 2, according to the specified scope in the `run` command, lines 8 and 9, the Alloy Analyzer searches for a valid instance of the model when the sizes of `Addr` and `Data` are 3 and 4, respectively.

2.2 Overview of SMV

An SMV model consists of a definition of a transition system and a set of temporal properties. The transition system is defined using variables, and conditional statements for the transition relation. The manual of SMV covers its syntax in detail [12].

Example 3. Figure 4 is an SMV model of a simple binary counter with three boolean variables: two for the two digits, `c0,c1` and one for the input value, `I` (declared on lines 2 and 3). The control variables `c0,c1` are initialized to 0. The conditional statements define the values of the variables in the following step using `next` statements.

LTL formulas are used to specify properties over infinite paths of computation; for example, `G p` is an LTL formula specifying that `p` is *globally* true: in other words, `p` holds in every state of a computation path. Another LTL connective that we use in our translation is `F`, *eventually*. `F p` is satisfied by a computation path if and only if at least one of the states along the computation path satisfies `p`.

A model checker tries to find a computation path that violates the formula; if such a path exists, the model checker states that the formula is *false* and returns a path that violates the property as a witness

to the user. This witness is called a *counterexample*, and it is used to understand the bugs in the system. If all computations paths satisfy the LTL property, the model checker return *true*.

Example 4. To check whether there exists a computation path in which `c1` is infinitely often true and `c0` is infinitely often false, we add the following LTL property to Example 3,

```
AS : assert ~((G F c1) & (G F ~c0));
```

This property is false and the model checker produces a counterexample that satisfies the negation of `AS`.

3 Translation Process

In this section, we describe our algorithm to translate Alloy models into SMV models. The general idea is to create an SMV model and specification such that the original Alloy model is *inconsistent* with respect to its finite scope, if and only if the SMV model *satisfies* its LTL specifications. If the Alloy model is consistent then the model checker produces a counterexample that represents a consistent instance of the Alloy model. The translation is done in such a way that *any* counterexample produced by the model checker is a valid instance of the original Alloy model.

To convert the problem of finding a valid instance of an Alloy model into a model checking problem, the two parts of an Alloy model, declarations and constraints, are translated to a transition system and a set of LTL formulas, respectively. The intuition behind the translation is that a transition system can be viewed as the definition of a set of infinite computation paths. A finite prefix of each of these paths can be considered as an interpretation of the original Alloy model; in other words, each state in this prefix represents a part of the interpretation of that path. Assumed LTL properties limit the computational paths considered to those that satisfy the Alloy model's constraints.

Algorithm 1 is the pseudo-code describing our translation algorithm. In the following subsections, we describe the different stages of the translation. The process can be further optimized for specific Alloy constraints, which is described in Section 4. Throughout this section, we will use the simple Alloy model of Figure 5 to illustrate our approach. It is translated into the SMV model of Figure 6. An example of a computation path of the model, which represents an interpretation of the model is shown in Figure 7.

Algorithm 1 ***Translation Algorithm:***

1. *Find the finite size of each set (called its `SetSize()`) declared in the model.*
2. *Compute the function size (`FuncSize()`) and relation size (`RelSize()`) for each declared function and relation in the model.*
3. *Let the interpretation size of the model (called `IntSize`) be the maximum value of `FuncSize()` and `RelSize()` for all functions and relations in the model.*
4. *Add the following SMV code for the definition of `TIME`:*

```
TIME: 0..IntSize;
init(TIME) := 0;
if(TIME<IntSize)
   next(TIME):= TIME+1;
else
   next(TIME):= TIME;
```

5. *For each relation declaration,*
   sig A1 {rel : set A2,.., set An}, add the following SMV code:

```
output rel: boolean;
init(rel) := 0;
if(TIME<RelSize(rel))
   next(rel):= 0..1;
else
   next(rel):= 0;
```

6. *For each function declaration,*
   sig A {func : [lone] B} add the following SMV code:

```
output func: 0..SetSize(B);
init(rel) := 0;
if(TIME<FuncSize(func))
   next(func):= LB..SetSize(B);
else
   next(func):= 0;
```

   If `func` *is a partial function, then* `LB` *is* `0`*; otherwise,* `LB` *is* `1`*.*

7. *For each constraint φ in the* `fact` *block do:*

   (a) *Create ψ as the expansion of φ's quantifiers by instantiating the variables with all the possible values from their sets.*

   (b) *Translate ψ to an LTL property and add the formula to the list of assumed properties called* `LAP`*.*

8. *Add the following assertion and property:*

```
Model: assert ~F(TIME=IntSize);
using LAP prove Model;
assume LAP;
```

3.1 Translating Declarations

The first step in translation is to design a transition system that defines the interpretations. An interpretation needs to specify the content of the functions and relations of an Alloy model. The contents of each relation and function in an interpretation are discovered in parallel with each other within a computation path of the transition system.

First, we calculate `IntSize`, the size of an interpretation of an Alloy model translated using our approach (steps 1–3 of Algorithm 1). We restrict the interesting part of a computation path to a prefix that is this size. The `IntSize` is the maximum size of any entity in the Alloy model. The size of a relation or function, called its `RelSize()` or `FuncSize()` respectively, is determined from the sizes of the sets of the Alloy model. The size of each set in Alloy, called its `SetSize()`, is found from the scope

```
1  sig B {}
2  sig C {func : lone A}
3  sig A {rel: set B -> set C}
4  fact{
5     some x:A,y:B,z:C| (x->y->z) in R
6  }
7  pred show[]{}
8  run show for exactly 2 A,
9      exactly 1 B, exactly 2 C
```

Figure 5: Example Alloy model

```
1  TIME: 0..4;
2  init(TIME):=0;
3  if (TIME < 4)
4     next(TIME):=TIME+1;
5  else
6     next(TIME):=TIME;
7
8  output func: 0..2;
9  init(func):=0;
10 if (TIME < 2)
11    next(func):= 0..2;
12 else
13    next(func):=0;
14
15 output rel: boolean;
16 init(rel):=0;
17 if (TIME < 4)
18    next(rel):= 0..1;
19 else
20    next(rel):=0;
21
22 p1: assert
23           F((TIME=1) & rel) |
24           F((TIME=2) & rel) |
25           F((TIME=3) & rel) |
26           F((TIME=4) & rel);
27
28 P: assert ~ F (TIME = 4);
29 using p1 prove P;
30 assume p1;
```

Figure 6: Translated SMV model of Figure 5

Computation Steps	0	1	2	3	4	5	6	...
TIME	0	1	2	3	4	4	4	...
rel	0	1	0	0	1	0	0	...
func	0	2	1	0	0	0	0	...

Figure 7: Example of a computation path of SMV model of Figure 6

Computation Steps	0	1	2	3	4	5	6	...
TIME	0	1	2	3	4	4	4	...
$A \times B \times C$	NA	(a_1, b_1, c_1)	(a_1, b_1, c_2)	(a_2, b_1, c_1)	(a_2, b_1, c_2)	NA	NA	...
rel(a,b,c)=	NA	true	false	false	true	NA	NA	...
C	NA	c_1	c_2	NA	NA	NA	NA	...
func[c]=	NA	a_2	a_1	NA	NA	NA	NA	...

Figure 8: Interpretation encoded by the computation path of Figure 7. (NA: Not Applicable)

command in the Alloy model. For all Alloy models, in SMV, we use `TIME` as an enumeration variable that ranges from `0` to `IntSize`. As shown on lines 1–6 of Figure 6, at each step, `TIME` is incremented by 1 until it reaches `IntSize`. Producing these definitions in SMV is step 4 of Algorithm 1. For a model with an interpretation size of 4, steps 1–4 inclusive on the computation path contain the elements of the interpretation. We have found that initializing everything to 0 results in better performance in model checking so step 0 on the computation path is not considered part of the interpretation of the Alloy model. Figure 7 is a computation path of SMV model of Figure 6, and Figure 8 represents the interpretation encoded by this computation path.

Relations: We model each relation as a Boolean variable and reserve one step in the computation for each possible tuple that could be in the relation (step 5 of Algorithm 1). At a step, if the value is `1` (true) this means the tuple is in the relation in the interpretation and vice versa. This method is an encoding of the characteristic predicate for the relation with some ordering of the tuples. The `RelSize` of a relation is thus the size of the Cartesian product of its component sets. The variable is nondeterministically assigned a value at each step, therefore, there is one computation path of the transition system for each interpretation of the Alloy model. For the general form of a relation declaration in Alloy:

```
sig A1 {rel :  set A2 -> ..  -> set An}
```

with the size of each set `Ai` being `Si` for $1 \leq \texttt{i} \leq n$, we calculate the size of the relation as:

```
RelSize(rel) = S1 * ..  * Sn
```

which is the maximum number of tuples that can be in `rel`. A computation path in which the value of `rel` is `1` when `TIME` is i is interpreted as an interpretation specifying that the i^{th} tuple of the Cartesian product of `A1` to `An` is in the relation `rel`.

For the model of Figure 5, `rel` is a relation of type $A \times B \times C$. The finite scope command in the Alloy model limits the sizes of sets `A, B`, and `C` to be 2, 1, and 2 respectively. Thus, the maximum number of tuples in the relation is 4 ($2 \times 1 \times 2$). We use steps 1 to 4 of the computation path to represent an interpretation of `rel`. Figure 7 contains values for `rel` from steps 1 to 4 inclusive. (We don't care about the values of `rel` after this step or in the initial step.) If set $A = \{a_1, a_2\}$, $B = \{b_1\}$, and $C = \{c_1, c_2\}$, then using an ordering of the tuples, such as the one in Figure 8, the interpretation of Figure 5 has `rel` containing the following tuples:

$$rel = \{(a_1, b_1, c_1), (a_2, b_1, c_2)\} \tag{1}$$

Functions: We model a function as a variable that takes on the possible values in the range of the function and we reserve one step in the computation for each element of its domain (step 6 of Algorithm 1). The `FuncSize` of a function is the size of its domain. The variable is nondeterministically assigned a value from the range at each step, therefore, there is one computation path of the transition system for each

interpretation of the function of the Alloy model. If the function is a partial one, we include the additional value 0 in the range, which represents the mapping from a domain element to undefined.

For the general form of a declaration of a function in Alloy:

$$\texttt{sig A \{func: [lone] B\}},$$

with `n` and `m` as the sizes of the sets `A` and `B`, respectively, we calculate the size of the function as:

$$\texttt{FuncSize(func)} = \texttt{n}$$

which is the size of the function's domain. A computation path in which the value of `func` is the positive integer `y` when `TIME` is `x`, is considered as an interpretation in which `func[x]=y`. If `func=0` when `TIME=x` then the value of function `func` for input `x` is interpreted as undefined.

For the model of Figure 5, `func` is a function of type $C \rightarrow A$. The finite scope command in the Alloy model limits the sizes of sets `A`, and `C` to be 2. We use 2 steps (the size of the domain) of the computation path to represent an interpretation of `func`. In Figure 7, an example computation path is shown, which contains values for `func` for steps 1 to 2. (We don't care about the values of `func` after this step or in the initial step.) If set $C = \{c_1, c_2\}$, and $A = \{a_1, a_2\}$, then using a sequential ordering of the elements of the sets, the interpretation of Figure 7 has `func` containing the mappings $\{c_1 \mapsto a_2, c_2 \mapsto a_1\}$.

For both functions and relations, the value of a function (relation) for the i^{th} input (tuple) is found when the value of `TIME` is i.

The keyword `extends` is syntactic sugar that can be written by using `in` and adding constraints to make subsets mutually disjoint. In general, the declaration `sig X in Y {}` is considered as a definition for a unary relation `X` of type `Y` and its translation is the same as other relations.

3.2 Translating Constraints

Translating declarations results in an SMV model that defines the possible finite interpretations of the Alloy model. The next step is to add constraints to this SMV model so that the invalid interpretations are filtered out. In the result, a counterexample in which `TIME` is equal to `IntSize` at some point in the computation path is a valid instance of the Alloy model.

In general, the constraints of an Alloy model are translated to a set LTL properties that are *assumed* on the model (step 7 of Algorithm 1). Assuming a property on an SMV model instructs the model checker to consider just the computation paths that satisfy the assumed property; if the model is inconsistent then there is no computation path that satisfies the properties.

Because the scope of an Alloy model is finite, quantifiers can be expanded into a finite number of constraints on the relations and functions. The expansion of quantifiers results in a set of propositional formulas that are easily expressible in LTL. Optimizations of the general process presented in this section are possible for some Alloy constraints by encoding them as part of the transition relation instead of in LTL to result in better performance. These constraints are discussed in Section 4.

Working from the outside of a formula in, we expand each quantifier and instantiate the formula with all the possible values for each quantified variable.

For a set `A` of 3 elements, the constraint, `all x:A | f[x] != a` is equivalent to the following:

```
f[1] != a && f[2] != a && f[3] != a
```

For functions, in our SMV model, the argument to the function, i, corresponds to the value of SMV variable representing that function at `TIME=i`. The translation of the above constraint into an LTL formula for the SMV model is as follows:

```
F((TIME=1)&(f~= a)) & F((TIME=2)&(f~= a)) & F((TIME=3)&(f~= a))
```

Tuple	Number
$1->1$	1
$1->2$	2
$2->1$	3
$2->2$	4

Figure 9: Sample ordering on $A \times A$

If the definition of a constraint involves relations, then the expansion of quantifiers is done according to the chosen ordering of the tuples.

It is fairly straightforward to understand how the above expansion works for most of Alloy's operators. In the following, we describe the expansion for the Cartesian product (`->`), join (`.`), and transitive closure (`^`) operators.

Cartesian Product: For a set `A` of 2 elements, the constraint specifying reflexivity of a relation `R`, `all x:A | (x->x) in R` is equivalent to the following:

```
(1->1) in R && (2->2) in R
```

To translate this constraint, an ordering on the set $A \times A$ is used. The same ordering on the tuples of a relation is used throughout the translation. If Figure 9 is the ordering for `A={1,2}`, the translation of the constraint is:

```
F ((TIME=1) & R) & F ((TIME=3) & R)
```

As shown in Figure 9, the tuple `(1->1)` is encoded as `1` and the tuple `(2->2)` is encoded as `3`. This approach is used to translate the constraint specified in the `fact` block of Figure 5 to the assertion `p1` of Figure 6 (lines 22-26).

Join: The join operator is translated by introducing a new relation and constraint. Suppose `R1` is a relation of type `A` $\times$ `B` and `R2` is a relation of type `B` $\times$ `C`. In translation, we introduce the new relation `R3`. Anywhere `R1.R2` is used in the Alloy model, we replace it with `R3`. Then, we add the following constraint to the model:

```
all x:A,z:C|(x->z)in R3 iff some y:B|(x->y)in R1 and (y->z)in R2
```

This constraint defines `R3` as the join of `R1` and `R2`.

Transitive Closure: The transitive closure of a relation that is defined over a finite set can be rewritten in terms of join and union. If RT is the transitive closure of the binary relation R of type $A \times A$ and the size of A is n, the following equation holds:

$$RT = R + R.R + \cdots + \overbrace{R.R...R}^{n \ times} \tag{2}$$

Our translator uses this equivalence to rewrite any uses of the transitive closure operator in terms of union and join and then translates these operations as explained previously.

3.3 Model Checking as a Decision Procedure

The final step (step 8 of Algorithm 1) is to add the specification to the SMV model that results from the previous steps of the translation process and an assertion for model checking. For all Alloys models, this assertion is as follows:

1	formula	::=	quantifiedFormula	
2		::=	simpleFormula | cardinalityFormula	
3		::=	formula `&&` formula	
4	quantifiedFormula	::=	`all` var:var `|` simpleFormula	`universal quantification`
5		::=	`some` var:var `|` simpleFormula	`existential quantification`
6	simpleFormula	::=	expr compOp expr	
7	cardinalityFormula	::=	# expr numCompOp number	
8	compOp	::=	`in` | `!in` | `=` | `!=`	`subset, not subset, equal, not equal`
9	numCompOp	::=	= | > | < | >= | <=	
10	number	::=	0 | 1 | 2 | ...	`none-negative integer`
11	expr	::=	var | var[var] | expr binOp expr	`simple set, function value, set operations`
12		::=	expr ++ tuple	`override`
13	binOp	::=	+ | & | -	`union, intersection, difference`
14	tuple	::=	var | var ->tuple	

Figure 10: Alloy constructs optimized at syntax-level

```
assert ~F (TIME = IntSize);
```

This property states that `TIME` can never be equal to `IntSize`. If the model checker outputs *true*, then this means that there is no computation path that satisfies the assumed LTL formulas in which `TIME` reaches the value `IntSize`; therefore, the original Alloy model is inconsistent. If the model checker outputs *false* and gives a counterexample, then it means there is an interpretation that satisfies all the constraints; therefore, the Alloy model is consistent and the counterexample is a valid instance of the model.

4 Optimization

The general approach of translation described in Algorithm 1 creates a transition system that produces *all* the finite interpretations of an Alloy model for a certain scope. The constraints of the Alloy model are enforced by *assuming* a set of LTL formulas on the model, which can result in a very large size set of LTL formulas. Reducing the size of LTL formulas that are model checked increases the performance of model checking. In this section, we describe how to optimize our translation process by reducing the size of the LTL assumptions and enforcing some constraints as part of the transition system's definition, rather than using assumed LTL formulas. This approach results in a smaller set of LTL formulas that need to be checked against the transition system and model checking is more efficient, meaning the size of the scopes that can be analyzed increases significantly.

We have found two kinds of optimizations: 1) syntax-level and 2) semantic-level. The applicability of syntax-level optimizations can be detected just by checking Alloy's syntax: no assistance from users is required. On the other hand, the applicability of semantic-level optimizations requires some assistance from users: in the current implementation of our translator, this aid can be provided as comments in Alloy models.

4.1 Syntax-Level Optimization

Figure 10 is a summary of the Alloy formulas that we can optimize by recognizing them syntactically.

Quantification: We can optimize the translation of non-nested formulas of the form `Q x:A | P(x)` where `Q` is either `all` or `some` and the bound variable, `x`, ranges over a simple set, `A`, and `A` is enumerated

by TIME (lines 4-6 of Figure 10). For example, in the property, `some x:A | (x->x) in R` , `A` is not enumerated by TIME, rather it is `A` $\times$ `A` that must be enumerated.

In this optimization, instead of expanding the quantified formula for all values of the set, we use TIME for quantification and directly map the quantifiers `all` and `some` to the temporal connectives `G` and `F` respectively. The Alloy constraint `all x:A | P(x)` is translated into the following LTL formula:

```
G ((TIME>0) & (TIME<=setSize(A)) -> P)
```

In the translation of `some`, `G` is replaced by `F`.

Example 5. Suppose `A` has 3 elements. The constraint `some x:A | f[x] != a` is translated as follows:

```
F ((TIME>0) & (TIME<=3) & (f~=a))
```

Since the value of `f` changes over time, there is no need to mention its argument, `x`, in the property.

Set Membership: In Alloy, relations, functions, and even elements are all viewed as sets. After the base sets, sets are defined by using other sets. For example, the range of a function is a set that is defined by the function and its domain. Many constraints can be rewritten in terms of set membership problems. We can optimize constraints that can be described as set membership properties to result in better performance in model checking.

Suppose A is a set with a definition based on the sets $S_1, .., S_n$. We say the constraint $x \in A$ can be evaluated *constructively* if and only if it only depends on $x \in S_1, .., x \in S_n$ For example, $x \in S_1 \cup S_2$ is a constructive constraint because it is equivalent to $x \in S_1 \vee x \in S_2$. The membership problem for the set of elements in the range of a function is not constructive. If a membership constraint is constructive (Figure 10, lines 11-14), we can optimize its translation by directly defining the relation constructively rather than letting its definition be nondeterministic and then constraining it through an assumed LTL property. In our translation, a relation is represented by a Boolean variable that changes over time. We can directly define the Boolean variable representing one relation in terms of the Boolean variables representing the relations of its definition.

Example 6. Suppose, we have the following Alloy model.

```
1  sig S {}
2  sig A,B,C in S {}
3  fact {C = A & B}
```

In this model, `A`, `B`, and `C` are subsets of `S` (line 2) and `C` is equal to the intersection of `A` and `B` (line 3). The characteristic function of `C` is equal to the conjunction of the characteristic functions of `A` and `B`. Therefore, the SMV code for enforcing this constraint on `C` can be stated as:

```
next(C) := next(A) & next(B);
```

This optimization is possible because checking whether the i^{th} tuple is in the intersection of sets `A` and `B` depends *only* on membership of the i^{th} tuple in the sets and no other tuple: it can be evaluated constructively.

Some other set operations, such as union, can be directly encoded into the transition systems with the same method explained in Example 6; also, constraints that explicitly state membership in sets, relations and functions can be directly mapped into the transition system definition.

Cardinality: Cardinality constraints (those that deal with the size of sets) can be optimized through a combination of changing the transition system and adding an LTL formula (Figure 10, lines 7 and 9-10).

Example 7. Consider the Alloy model in Example 6: suppose the constraint `# C = 2` is added to the `fact` block. This constraint states that the set `C`, which is equal to the intersection of `A` and `B`, must have

```
1 C_card : 0 .. SetSize(S);
2 init(C_card) := 0;
3 next(C_card) := C_card + next(C);
```

Figure 11: Example 7: Translation of cardinality

exactly 2 elements. To translate this constraint, a new variable, C_card, is introduced in the SMV model. This variable counts the number of elements in C, and is incremented whenever the Boolean variable C is set to 1. Figure 11 is the corresponding SMV model.

To enforce that C must have 2 elements, the following LTL property is assumed on the model:

```
assert F ((TIME=IntSize) & (C_card=2));
```

This property states that when the creation of an interpretation is done, TIME=IntSize, the value of C_card is equal to 2.

4.2 Semantic-Level Optimization

Semantic-level optimization is accomplished through user annotation of the Alloy model in comments that show a constraint in a fact block is an instance of a commonly used constraint, e.g., a function is one-to-one. Alloy does not have specific constructs or keywords to assist users in labeling a constraint by its common name. Figure 13 is a summary of the annotations that our translator currently recognizes through comments of the form //@ Annotation, before the constraint, and //@, after the constraint.

Semantic-level optimizations introduce Boolean flags into the transition system. These flags are set as soon as a property is violated by an interpretation. The advantage of this approach is that an invalid interpretation can be dismissed as soon as an inconsistency is detected.

Example 8. The following Alloy model specifies that the function f is one-to-one:

```
1 sig T {}
2 sig S {f: T}
3 fact {
4   //@ f OneToOne
5   all x,y:S | f[x]=f[y] => x=y
6   //@
7 }
```

Our translator recognizes the keyword OneToOne in the comment and optimizes the translation of this constraint by introducing an additional array of booleans in the model and a flag, f_flag, that is set to 1 whenever an element of T appears more than once in the range of f. Figure 12 shows the SMV translation of the OneToOne property. In this code, SetSize(S),SetSize(T) are replaced with their values. Whenever f is assigned the value x (line 2-4), the x^{th} Boolean in array f_array is set (line 17); if this flag has already been set then the flag, f_flag, is set (lines 14 and 15). In the assumed properties, we specify that the flag, f_flag, should always be false (line 18). Since the value of f is not important after TIME passes the domain size of f, SetSize(S), the checking and setting of flags is irrelevant after that moment (line 13).

Our other semantic-level optimizations listed in Figure 13 are accomplished in a similar manner: an array is used to record information about values chosen for the function or relation in the previous steps and a Boolean flag is set when there is a violation of the constraint. The constraints Reflexive and NonReflexive are specified by using only one quantifier, but they cannot be optimized via a syntactic

```
 1  output f: 0..SetSize(T);
 2  if (TIME < SetSize(S))
 3     next(f):= 1..SetSize(T);
 4  else
 5     next(f):=0;
 6
 7  f_array : array 1..SetSize(T) of boolean;
 8  f_flag : boolean;
 9  init(f_flag):=0;
10  for (i=1; i<=SetSize(T); i=i+1)
11    init(f_array[i]) := 0;
12
13  if (TIME<SetSize(S))
14     if(f_array[next(f)])
15       next(f_flag):=1;
16     else
17       next(f_array[next(f)]):=1;
18  Property : assert G ~ f_flag;
```

Figure 12: SMV translation of `OneToOne` property for Example 8

Annotation	Equivalent Alloy Constraint
`R AntiSymmetric`	`some x,y:A | (x->y) in R && (y->x) !in R`
`R NonReflexive`	`some x:A | (x->x) !in R`
`F OneToOne`	`all x,y:A | F[x]=F[y] => x=y`
`R Symmetric`	`all x,y:A | (x->y) in R <=> (y->x) in R`
`R Reflexive`	`all x:A | (x->x) in R`

Figure 13: Alloy constructs optimized at semantic-level

.

optimization because the bound variables range over sets that are not enumerated by `TIME`. For these constraints pairs are enumerated by `TIME`.

5 Case Studies

In this section, we present our experimental results. We have implemented the translation algorithm and optimizations described in the previous sections using Turing eXtender Language (TXL) [18]. Table 1 compares the results of checking for the existence of a consistent model instance by the Alloy Analyzer, version 4.2, with running Cadence SMV, release 10-11-02p36, on our translated models. This table shows the results for four models and analysis for different sizes of scopes. This table also shows the number of optimized constraints out of the total number of constraints (NOC) in a model. 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.

The "Harry Potter" model involves a total function from a set with seven elements to a set with three elements[1]. There are some constraints on the function that make the solution unique up to isomorphism. The "modeling infinity" model consists of a one-to-one function from a set to one of its proper subsets.

[1]This example is based on the riddle found in J.K. Rowling's book "Harry Potter and the Philosopher's Stone" (Bloomsbury Publishing Plc, Nov. 1998).

Harry Potter			Modeling Infinity			Memory Abstraction			SDR		
SS	Alloy	SMV	SS	Alloy	SMV	SS	Alloy	SMV	SS	Alloy	SMV
11	0.05 sec	2.66 sec	15	5.09 sec	5.03 sec	100	1.54 sec	5.01 sec	10	0.07 sec	0.01 sec
			20	29.95 min	29.98 sec	200	10.67 sec	26.28 sec	10^2	0.19 sec	0.25 sec
			25	>30 min	1 min 51 sec	300	39.71 sec	1 min 11 sec	10^3	7.78 sec	2.45 sec
			30	>1 h 30 min	9 min 9 sec	400	1 min 35 sec	1 min 19 sec	10^4	>30 min	1 min 46 sec
NOC: 3 out of 16			NOC: 2 out of 2			NOC: 3 out of 3			NOC: 8 out of 8		
consistent			inconsistent			consistent			inconsistent		

Table 1: Experimental Results (h: hour, min: minute, NOC: Number of Optimized Constraints, sec: second, SS: Scope Size)

This model does not have any finite valid interpretations. The translation of this model involves one property that can be semantically optimized. The third case study is a variation of the systems of distinct representatives (SDRs) [19]. For this model, all constraints are either constructive or they are cardinality constraints. The "memory abstraction" case study is a simple model of write and read operations on memory.

The significant difference in runtime between the Alloy Analyzer and SMV for the "modeling infinity" and SDR models is because optimizations were applicable to all the constraints in these Alloy models. As a result, the BDD that represents the transition systems of these models is very efficient; also, the use of dynamic BDD variable reordering for larger scopes significantly benefits from these optimization. The simple expansion of quantifiers and constraints described for the general translation does not result in model checking having better performance than the SAT solver of the Alloy Analyzer. For the memory abstraction model, on average, only 1.7 percent of the analysis time shown in Table 1 was used by the SAT-solver and the rest was spent on creating the CNF formula.

Overall, we have found that in the cases where the Alloy models are inconsistent and constraints in the translated models can be optimized, Cadence SMV is faster than the Alloy Analyzer. In these cases, it takes a longer time for the Alloy Analyzer's SAT solver to conclude the CNF formula representing the model is inconsistent than it does for SMV's BDD-based model checker to conclude its LTL specification is true for the translated model.

6 Related Work

The Alloy language together with its analyzer is unique in that it combines the ability to write high-level declarative specifications with fully automated analysis via SAT solving for finite scopes. Most other approaches that support high-level specifications are supported by theorem proving-based analysis. For example, Z [15] is another set-based modelling notation and its analyzers are focused on theorem proving rather than finite scope analysis. These theorem provers, such as ProofPower [20], are not fully automatic.

Marinov et al. propose a method to optimize Alloy models by transforming an Alloy model to another Alloy model based on the sizes of sets and the used constructs in the model [21]; as a result, the Alloy Analyzer generates an optimized CNF formula. Unlike their method, which only supports syntax-level optimization, our method supports semantic-level optimization as well.

B [22] is a modeling language that has many similarities with Alloy. It uses sets as elements of the state space, and it is mainly used for modelling critical systems. ProB [14] is an animation and modeling checking tool for the B method that uses Prolog to generate counterexamples automatically. B does not support the definition of arbitrary assertions: ProB checks the proof obligations generated by invariants and refinement claims.

The Abstract State Machine (ASM) method [23] is for high-level system design and analysis. The

ASM method is used to specify infinite transition systems. Analysis techniques for the ASM method include theorem proving [24, 25, 26] and model checking [16, 17]. Alloy models can be more abstract than ASMs due to the declarative aspects of Alloy and the logic that it provides for expressing constraints on models.

In [13], Chang and Jackson augment the traditional languages of model checkers by sets and relations and declarative constructs to specify a transition system. They developed a symbolic model checker for their language. We use model checking to analyze static properties, and they analyze dynamic properties of relational structures.

7 Conclusion

We have shown how model checking can be used for efficient analysis of static properties of declarative models; in particular, we presented a method for translating Alloy models to Cadence SMV and we used its BDD-based model checker as a decision procedure. The main idea is to view finite interpretations as dynamic entities, rather than static ones. In our approach, interpretations are encoded as parts of the computational paths of a transition system, and LTL formulas are used to enforce constraints on interpretations. A produced SMV model satisfies its LTL specifications if and only if the original Alloy model is inconsistent with respect to its finite scope; counterexamples produced by the model checker are valid instances of the Alloy model. Spreading interpretations over time and constructing them step-by-step makes it possible to discard invalid interpretation faster. Our experiments show that for many constructs of Alloy, we can optimize the translated models by encoding more information in transition systems (and less in the LTL assumptions). Model checking is more efficient than using the SAT-solver of the Alloy Analyzer. Some constraints we can recognize syntactically and optimize; others can be recognized and optimized via user annotations in the comments of the Alloy model.

In the future, we plan investigate whether additional constraints can be optimized via either syntactic recognition or user annotation. Additionally, we plan to translate Alloy into other model checkers (e.g., [27]) to see if alternative model checking methods perform better on certain constraints.

References

[1] A. Vakili and N. A. Day, "Using model checking to analyze static properties of declarative models," in *The IEEE/ACM Proceedings of the International Conference on Automated Software Engineering (ASE)*, Nov. 2011, (To appear as a short paper).

[2] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering Methodology*, vol. 11, no. 2, pp. 256–290, 2002.

[3] ——, *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[4] P. Zave, "A formal model of addressing for interoperating networks," in *FM 2005: Formal Methods*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, vol. 3582, pp. 633–633.

[5] M. C. Reynolds, "Lightweight modeling of Java virtual machine security constraints using Alloy," CS Department, Boston University, Tech. Rep. 2008-031, Dec. 30 2008.

[6] H. R. Nielson and F. Nielson, *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[7] E. Kang and D. Jackson, "Formal modeling and analysis of a flash filesystem in Alloy," in *Abstract State Machines, B and Z*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, vol. 5238, pp. 294–308.

[8] Alloy Community. [Online]. Available: http://alloy.mit.edu/

[9] E. A. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990, pp. 996–1072.

[10] J. R. Burch, E. M. Clarke, D. L. Dill, J. Hwang, and K. L. McMillan, "Symbolic model checking: 10^{20} states and beyond," *ic*, vol. 98, no. 2, pp. 142–171, 1992.

[11] K. L. McMillan, "Symbolic Model Checking: An Approach to the State Explosion Problem," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992, cMU-CS-92-131.

[12] ——, "The SMV system," Nov. 06 1992. [Online]. Available: http://www.kenmcmil.com/language.ps

[13] F. S.-H. Chang and D. Jackson, "Symbolic model checking of declarative relational models," in *28th International Conference on Software Engineering (ICSE '06)*, May 2006, pp. 312–320.

[14] M. Leuschel and M. Butler, "ProB: A model checker for B," in *FME 2003: Formal Methods*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, vol. 2805, pp. 855–874.

[15] *Information technology—Z Formal Specification Notation—Syntax, Type System and Semantics*, International Organisation for Standardization, 2000.

[16] G. Del Castillo and K. Winter, "Model checking support for the ASM high-level language," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2000, vol. 1785, pp. 331–346.

[17] A. Gawanmeh, S. Tahar, and K. Winter, "Interfacing ASM with the MDG tool," in *Proceedings of the Abstract State Machines 10th international conference on advances in theory and practice*. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 278–292.

[18] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow, "TXL: A rapid prototyping system for programming language dialects," *Computer Languages*, vol. 16, no. 1, pp. 97–107, 1991.

[19] H. B. Mann and H. J. Ryser, "Systems of distinct representatives," *The American Mathematical Monthly*, vol. 60, no. 6, pp. pp. 397–401, 1953.

[20] ProofPower. [Online]. Available: http://www.lemma-one.com/ProofPower/index/index.html

[21] D. Marinov, S. Khurshid, S. Bugrara, L. Zhang, and M. Rinard, "Optimizations for compiling declarative models into boolean formulas," in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science. Springerg, 2005, vol. 3569.

[22] J.-R. Abrial, *The B Book: Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[23] E. Börger, "The ASM method for system design and analysis. a tutorial introduction," in *Frontiers of Combining Systems*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, vol. 3717, pp. 264–283.

[24] G. Schellhorn and W. Ahrendt, "Reasoning about Abstract State Machines: The WAM case study," *Journal of Universal Computer Science*, vol. 3, no. 4, pp. 377–413, 1997.

[25] A. Dold, "A formal representation of Abstract State Machines using PVS," Universität Ulm, Verifix Technical Report Ulm/6.2, Jul. 1998.

[26] A. Gargantini and E. Riccobene, "Encoding Abstract State Machines in PVS," in *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications.* London, UK: Springer-Verlag, 2000, pp. 303–322.

[27] G. J. Holzmann, *The Spin Model Checker, Primer and Reference Manual.* Reading, Massachusetts: Addison-Wesley, 2003.