Using Model Checking to Analyze Static Properties of Declarative Models

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

***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.**

I. 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 *temporal* properties, e.g., "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 [1]. It has an analyzer, the Alloy Analyzer, which supports finite scope 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 [2]. As the sizes of the scope in an Alloy model increase, the time taken by the Alloy Analyzer increases, and it 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[1]. Constraints on Alloy models are encoded as linear temporal logic (LTL) formulas [4] over computational paths of the transition system. Then, we use a BDD-based symbolic model checker [5] 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 and use the Cadence SMV model checker [6], 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.

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 analysis approaches for high-level specifications are supported by theorem proving-based tools. For example, Z [7] 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 [8], are not fully automatic.

[1]An extended version of this paper is available as a technical report [3].

978-1-4577-1639-3/11/$26.00 ASE 2011, Lawrence, KS, USA

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 constructs used in the model [9].

B [10] is a modeling language that has many similarities with Alloy. It uses sets as elements of the state space, and is mainly used for modelling critical systems. ProB [11] 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 [12] 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 [13]–[15] and model checking [16], [17] that consists of translating a certain class of ASMs, which is not as declarative as our models, to SMV. Translation of abstract models to the input language of state-of-the-art model checkers may result in large models. Not having operational semantics for an abstract model leaves a model checking method based on translation with no choice other than fully expanding the transition relation of the abstract model. 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.

Chang and Jackson augment the traditional languages of model checkers by sets and relations and declarative constructs to specify a transition system [18]. 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.

II. Translation Process and Optimization

In this section, we describe our algorithm to translate Alloy models into SMV models and how to optimize the translation. 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.

```
 1 sig B {}
 2 sig C {func : A}
 3 sig A {rel: set B -> set C}
 4 fact{
 5    # rel = 2
 6    //@ func OneToOne
 7    all x,y:C | func[x]=func[y] => x=y
 8    //@
 9 }
10 pred show[]{}
11 run show for exactly 2 A,
12     exactly 1 B, exactly 2 C
```

Fig. 1. 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 output func: 0..2;
 8 init(func):=0;
 9 if (TIME < 2)
10    next(func):= 1..2;
11 else
12    next(func):=0;
13 output rel: boolean;
14 init(rel):=0;
15 if (TIME < 4)
16    next(rel):= 0..1;
17 else
18    next(rel):=0;
19 rel_card : 0 .. 4;
20 init(rel_card) := 0;
21 next(rel_card) := rel_card + next(rel);
22 func_array : array 1..2 of boolean;
23 func_flag : boolean;
24 init(func_flag):=0;
25 for (i=1; i<=2; i=i+1)
26   init(func_array[i]) := 0;
27 if (TIME<2)
28    if(func_array[next(func)])
29      next(func_flag):=1;
30    else
31      next(func_array[next(func)]):=1;
32 p1: assert F ((TIME=4) & (rel_card=2));
33 p2 : assert G ~ func_flag;
34 P: assert ~ F (TIME = 4);
35 using p1,p2 prove P;
36 assume p1,p2;
```

Fig. 2. Translated SMV model of Figure 1

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∗	..

Fig. 3. Example of a computation path of SMV model of Figure 2 (∗: not used for interpretation)

Translation Process: The different stages of the translation process are explained by using the simple Alloy model of Figure 1, which is translated into the SMV model of Figure 2. An example of a computation path of the model, which represents an interpretation of the model is shown in Figure 3.

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 the size of an interpretation of an Alloy model. This value is the maximum size of any entity in the Alloy model. We restrict the interesting part of a computation path to a prefix that is this size. The size of a relation or function is determined from the sizes of the sets of the Alloy model. The size of each set in Alloy is found from the scope command in the Alloy model. For all Alloy models, in SMV, we use `TIME` as an enumeration variable that ranges from `0` to the size of an interpretation (lines 1–6 of Figure 2). For a model with an interpretation size of 4, steps 1–4 inclusive on the computation path contain the elements of the interpretation. Each relation is modeled as a Boolean variable and we reserve one step in the computation for each possible tuple that could be in the relation. At a step, if the value is `1` (true) this means the tuple is in the relation in the interpretation and vice versa. The size 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 (Figure 2, lines 13-18, therefore, there is one computation path of the transition system for each interpretation of the Alloy model. In the model of Figure 1, `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`. In Figure 3, an example computation path is shown, and the parts of it that are not used for interpretation are marked by stars. This path 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 a sequential ordering of the tuples, the interpretation of Figure 1 has `rel` containing the tuples (a_1, b_1, c_1) and (a_2, b_1, c_2).

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. The size of a function is the size of its domain. The variable is nondeterministically assigned a value from the range at each step (Figure 2, lines 7-12), 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 function, we include the additional value 0 in the range, which represents the mapping from a domain element to undefined. For the model of Figure 1, `func` is a function of type $C \rightarrow A$. The finite scope command in the Alloy model (lines 11 and 12 of Figure 1) 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`. The example computation path in Figure 3 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 3 has `func` containing the mappings $\{c_1 \mapsto a_2, c_2 \mapsto a_1\}$.

In general, the constraints of an Alloy model are translated to a set of LTL properties that are *assumed* on the model. 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. 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. The details of expansion for the Cartesian product (`->`), join (`.`), and transitive closure (`^`) operators can be found in Vakili and Day [3].

The final step in the translation is to add the specification to the SMV model that `TIME` can never be equal to `4` (line 34 of Figure 2). 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 the value `4`; 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.

Optimization: Reducing the size of LTL formulas and optimizing the translation of constraints that are model checked increases the performance of model checking. We have found two kinds of optimizations: 1) syntax-level

The applicability of syntax-level optimizations can be detected just by checking Alloy's syntax: no assistance from users is required. For example, the cardinality constraint of Figure 1 (line 6) can be optimized directly. This constraint states that the relation `rel` must have exactly `2` elements. To translate this constraint, a new variable, `rel_card`, is introduced in the SMV model. This variable counts the number of elements in `rel` and is incremented whenever the Boolean variable `rel` is set to `1` (Figure 2, line 19-21). To enforce that `rel` must have `2` elements, the property `p2` is assumed on the model (Figure 2, line 42).

The applicability of semantic-level optimizations requires some assistance from users: in the current implementation of our translator, this aid needs to be provided as comments in Alloy models. 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. 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. For example, our translator recognizes the keyword `OneToOne` in the comment, line 5 of Figure 1, and optimizes the translation of this constraint by introducing an additional array of booleans in the model and a flag, `func_flag`, that is set to `1` whenever an element of `A` appears more than once in the range of `func`. Lines 22-31 of Figure 2 shows the SMV translation of the `OneToOne` property. The assumed property specifies that the flag, `func_flag`, should always be false while the assignments are relevant (Figure 2, lines 27 and 33).

The complete list of Alloy constructs that we can optimize by recognizing them syntactically and semantic optimization keywords for semantic-level optimization can be found in Vakili and Day [3].

Our experiments and preliminary results 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 [3]. In the future, we plan investigate whether additional constraints can be optimized via either syntactic recognition or user annotation.

REFERENCES

[1] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM TOSEM*, vol. 11, no. 2, pp. 256–290, 2002.

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

[3] A. Vakili and N. A. Day, "Using model checking to analyze static properties of declarative models: Extended version," Cheriton School of Computer Science, University of Waterloo, Tech. Rep. CS-2011-22, 2011.

[4] E. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[5] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 1020 states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.

[6] K. L. McMillan, "The SMV system," Nov. 06 1992. [Online]. Available: http://www.kenmcmil.com/language.ps

[7] *Information Technology–Z Formal Specification Notation-Syntax, Type System and Semantics*, International Organisation for Standardization, 2000.

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

[9] 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. LNCS. Springer, 2005, vol. 3569.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[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] F. S.-H. Chang and D. Jackson, "Symbolic model checking of declarative relational models," in *ICSE '06*, May 2006, pp. 312–320.