Initial Abstraction Generation Using Requirements

Yinghua Jia
Dept. of Computer Science
University of Toronto
yjia@cs.toronto.edu

Abstract

Many researchers have proposed approaches for automatic abstraction refinement. However, the role of requirement in initial abstraction generation is under-investigated. I believe that in order to scale model checking to realistic programs, aggressive abstraction based on requirement is necessary. In this project paper, the idea of requirements guided abstraction is outlined and a discussion of issues for realization of this idea is also presented.

1 Introduction

The major obstacle to applying model checking to realistic programs is the state explosion problem. Abstraction is one of the most commonly used techniques in reducing the size of the model the program describes. Generally speaking, applying the abstraction technique involves obtaining an abstracted model with a given *abstraction function*, model checking the abstracted model, and refining the abstraction (modifying the abstraction function) based on counter-examples [4] or the proof of unsatisfiability (*i.e.,* in bounded model checking, a proof that there is no counterexample to the property of k steps or fewer) [8]. The steps of model checking the abstracted model and refining abstraction are repeated until the coarsest abstraction that satisfies the properties is obtained, or the concrete model violates the property. As can be seen, the process of finding the right abstraction function is crucial to the effectiveness of reducing state space, and to what degree such a process can be automated determines whether the model checking can be used more widely in industry.

However, in most of the cases, the abstraction function is designed manually requiring considerable creativity. Even in some approaches where the initial abstraction can be obtained automatically, the automatic construction of the abstraction function is confined by variable dependency graph [1], or by atomic formulas [4] extracted from the program. Both of the approaches are affected either by the fact that it is awkward for the developers to write/specify temporal logic formulas, or by the concreteness of the actual program. It is under-appreciated that the requirements of the system are actually the coarsest model descriptions providing the natural basis for abstraction function construction, and that the requirements describes (either explicitly or implicitly) all of the properties of the system thus making it more important to verify whether the program satisfies the requirements rather than several segmented LTL formulas.

In this project paper, a brief overview of the techniques used in abstraction generation is given in Section 2. The idea of using requirements for abstraction function construction is outlined in Section 3, followed by a simple example study using the design of an elevator. Finally, I conclude with a discussion of the central issues that need to be addressed in applying the idea in practice.

2 Related Work

This section summarizes several popular techniques for states abstraction, focusing on the construction of abstraction functions. The transitions among the abstracted states can be obtained by over-approximation, under-approximation, or both.

Predicate Abstraction Predicate abstraction is first described by Graf and Saidi [6], and provides a means for combining theorem proving and model checking techniques. In their proposed approach, the abstract states correspond to truth assignments to a set of predicates. Therefore, an abstraction function is defined by the predicates. The user must supply the predicates and properties to be proven. Usually, properties or *invariants* are provided as part of the predicates, meaning that such predicates hold for every reachable state of a concrete system. The effectiveness of this approach depends on predicates selection; whether user knows what invariants he or she wants to prove.

Variable Dependency Graph Abstraction construction based on variable dependency (or process dependency) graph is described in detail by Kurshan [7], and similar approach is described by Balarin in [1]. The basic idea is based on *local property*. That is, some subset of coordinating processes satisfying property T is equivalent to the whole set of coordinating processes satisfying property T. In such a sense, the abstraction function is defined as the mapping between the whole set of coordinating processes and equivalent subset of processes with respect to the local property T. The process dependency graph is used in both initial abstraction generation and abstraction refinement. This type of approach is less effective when the size of the model is mainly caused by variable range. In such cases, aggressive data abstraction is necessary.

Atomic Formulas The counterexample-guided abstraction refinement approach [4] uses atomic formulas, which are similar to the predicates by Graf and Saidi [6], for abstraction. However, instead of using the atomic formulas to generate an abstract global transition system, they use them to construct an explicit abstraction function. The atomic formulas are automatically extracted from the program. For example, for a guarded assignment $\mathsf{next}(y) = (x < y) : y + 1$, $x < y$ would be extracted as one atomic formula, and may be clustered with other atomic formulas forming one formula cluster FC. One abstraction function is defined as the mapping between the equivalent classes of one FC and the cross product of the involved variables domains; multiple abstraction functions may be defined for a model. Automating the construction of abstraction functions may make the approach more acceptable to industry, but the degree of abstraction is likely to be affected by the concreteness of the program. Thus it is not ideal to be used for realistic program verification, rather preferably used for high-level model verification.

Realistic Program Abstraction Two representative approaches have been proposed for realistic program abstraction: Bandera [5], which is based on aggressive data abstraction, and Slam [2], which is based on *boolean programs*. The data abstraction in Bandera is realized by user providing the mapping between the concrete program values and abstract values and defining a collection of abstract primitive operations. For example, statement Integer $n < 0 \rightarrow$ NEG, abstracts integer n's negative values as NEG value. The disadvantages of data abstraction are: (1) the effectiveness of the abstraction largely depends on the selection of the domain of abstract values; (2) there is no natural way of incorporating properties into the abstract domain selection. The approach proposed in Slam is similar to predicate abstraction in that both approaches use provided properties to guide abstraction. Again, this approach relies on the developer' ability of writing effective properties, which is often an awkward thing for developers to do.

3 The Idea and The Example Study

The idea of using requirements as the basis for abstraction construction is similar to the Verification Library Language (VLL) project by Al Aho et al.. The vision of the VLL project is to regard verification as part of the development process. That is, the developers only need to verify that the program satisfies specifications (requirements) given that the specifications (requirements) satisfy the envisioned system properties.

As illustrated by solid arrows in Figure 1, in most cases, the system properties are believed to hold in concrete model by verifying that the properties hold in the abstracted model. When counter-examples occur, the abstracted model is refined until the concrete model violates the properties or the properties are verified. For requirements guided abstraction generation, as shown by dotted arrows in the figure, there are two options to verify whether the concrete model satisfies the system properties: first, using requirements only for guiding the construction of the initial abstraction function, and following the paradigm discussed before; second, verifying that the abstracted model fulfills the specification, and the specification satisfies the system properties.

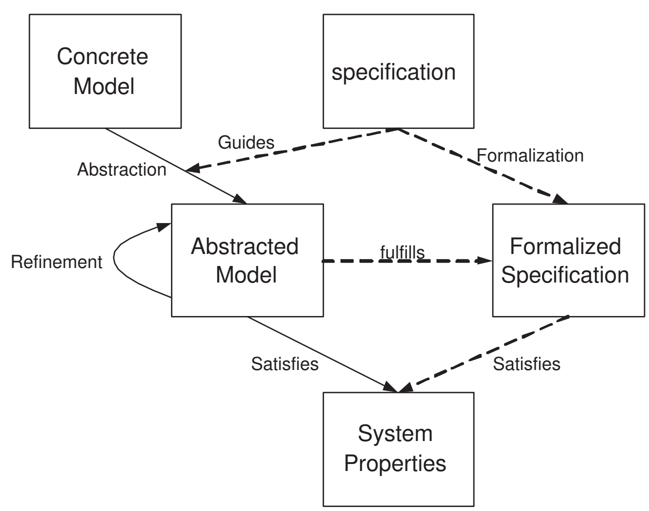

Figure 1. Requirements Guided Abstraction Generation

Let's look at the design of an elevator. The requirements (specifications) of different grain levels for a general elevator are shown in Table 1.

Basic Requirements: The elevator is able to get people up and down the building. The elevator is safe to use and won't keep people waiting forever.
Refined Requirements: The elevator arrives at that floor and opens the door whenever there is a request to use the elevator at that floor, or there is a request to be delivered to that floor. The elevator never moves with its door open. The elevator also allows people to hold the door open for a limited time.
Specification: The elevator moves up if all of requests are from or destined for upper floors, moves down if all of the requests are from or destined for lower floors, or keeps original (previous) moving direction otherwise. The elevator can start moving only when the door is closed and the door is not or can no longer be requested to open.
Formalized Specification: *currFlr* is the floor where the elevator is at, r is the level of the floor where there are requests, and *dir* is the moving direction $\begin{aligned} \textit{next(currFlr)} := & \forall r\ r > \textit{currFlr} : \textit{currFlr} + 1 \\ & \forall r\ r < \textit{currFlr} : \textit{currFlr} - 1 \\ & \textit{dir} = \textit{up} : \textit{currFlr} + 1 \\ & \textit{dir} = \textit{down} : \textit{currFlr} - 1 \\ & \ldots \\ & \ldots \end{aligned}$

Table 1. Requirements for a General Elevator

The concrete model of a 3-level elevator in can be implemented in SMV as follows, where currFlr represents the floor where the elevator is, req1, req2, and req3 represent the requests to use the elevator from floor 1, 2 and 3 respectively, dest1, dest2, and dest3 represent the requests to be delivered to floor 1, 2, and 3 respectively, and dir represents the moving direction.

```
...
...
next(currFlr) := case
(currFlr = 1) ∧ (req2 ∨ dest2 ∨ req3 ∨ dest3) ∧ ¬(req1 ∨ dest1) : 2;
(currFlr = 3) ∧ (req1 ∨ dest1 ∨ req2 ∨ dest2) ∧ ¬(req3 ∨ dest3) : 2;
(currFlr = 2) ∧ (req3 ∨ dest3 ∨ req1 ∨ dest1) ∧ ¬ (req2 ∨ dest2) ∧ (next(dir) = up): 3;
(currFlr = 2) ∧ (req3 ∨ dest3 ∨ req1 ∨ dest1) ∧ ¬ (req2 ∨ dest2) ∧ (next(dir) = down): 1;
esac;
next(dir) := case
(currFlr = 1) : up;
(currFlr = 3) : down;
(currFlr = 2) ∧ (req3 ∨ dest3) ∧ ¬ (req1 ∨ dest1 ∨ req2 ∨ dest2): up;
(currFlr = 2) ∧ (req1 ∨ dest1) ∧ ¬ (req3 ∨ dest3 ∨ req2 ∨ dest2): down;
(currFlr = 2) : dir;
esac;
```

. . .

. . .

Considering the concrete model that involves only the variables shown above, the concrete model has $3 \times 2^7 = 384$ states. If we use the specification for the construction of abstraction function, meaning that we choose the predicate $\forall$ requests, reqFlr $>=$ currFlr, we can get an abstracted model of $2{+}2{+}2{\times}3 = 10$ states, which is depicted in Figure 2. As can be easily verified from the Figure 2, the abstracted model fulfills the specification.

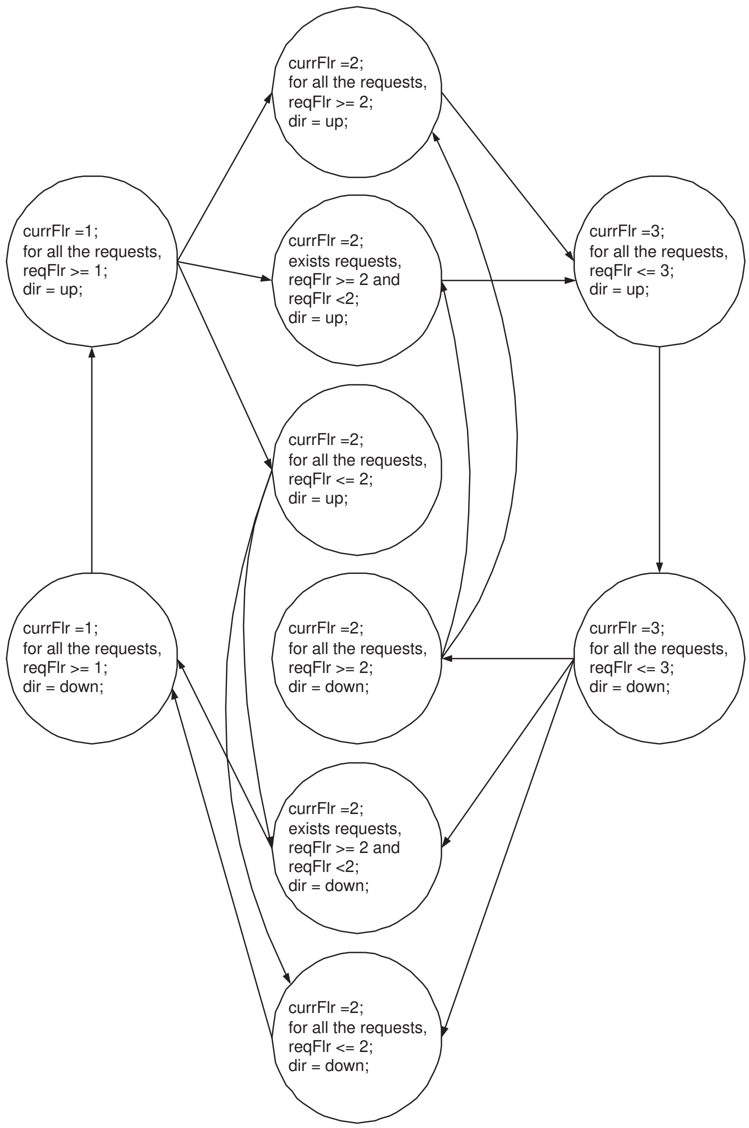

Figure 2. Abstracted Model by Specification

It becomes extremely difficult when we try to use atomic formulas for the construction of the abstraction functions, because no atomic formulas extracted from the SMV code are abstract enough to reduce the states significantly. Similarly, there is no further abstraction can be done given that the system property is any request eventually being served, because the property concerns request and the elevator status, and the requests and the elevator status are mutual-dependent.

Another immediate advantage of using requirements (specification) to guide abstraction is that different level abstractions can be obtained depending on the level of the requirements used. For example, if we use refined requirements for an elevator in Table 1 for abstraction, the abstracted model would have 6 states as shown in Figure 3.

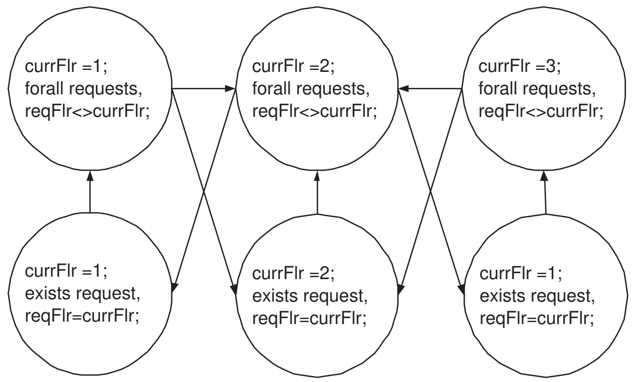

Figure 3. Abstracted Model by Refined Requirements

4 Conclusion and Discussion

In this project paper, I used the design of an elevator to illustrate the idea of using requirements for initial abstraction generation. Compared with other techniques, I consider the idea as a supplementary facility to the existing abstraction techniques. It makes the abstraction refinement process more effective by starting at a proper abstract level and by providing meaningful abstraction functions. The mapping and the predicates that are used in the abstraction functions can be used for different refinement techniques. Another advantage of using specifications (requirements) for abstraction generation is that the techniques used for managing specifications (requirements) evolution and integration can be modified for managing the corresponding abstraction functions, thus these functions can be re-used for the verification of the new model.

There are still several obstacles to realizing the idea in practice and to automating this process.

1. **how can the construction of abstraction functions be automated?** As can be seen from the elevator example study, the predicates and the mappings are partially embedded in the specifications (requirements) either explicitly (*i.e.,* $\forall r$ $r <$ *currFlr* : *currFlr* -1) or implicitly (*i.e.,* the elevator *arrives at that floor* ... implies the predicate whether elevator arrives at the requested floor — (currFlr = reqFlr)). Therefore, three steps are needed for automating the abstraction generation. First, atomic formulas as predicates are automatically extracted from the specifications (requirements). It is easy for formalized specification. For example, for the specification statement $\forall r$ $r <$ *currFlr* : *currFlr* -1, the atomic formulas are: r, currFlr, $r <$ currFlr, and currFlr -1. But for the narrative or graphic specifications (requirements), either formalizing the specifications (requirements) then extracting atomic formulas, or using existing formal framework for reasoning requirements

diagrams such as the framework by [3] can be used. Second, the mappings between the variables in the concrete model and variables in the atomic formulas are constructed. For example, when req2 being true in the concrete model is equivalent to $r = 2$. This could be the hardest part to be automated. Intuitively, a dictionary that maintains the relationship between the variables in the concrete model and the names of the attributes in the specifications (requirements) can be used to automate this step. Third, concrete states are clustered according to the atomic formulas.

2. **what level of the requirement should be chosen for deriving abstraction functions?** Different level of specifications (requirements) can generate different level of abstractions. The more general of the specifications (requirements), the more reduction of the states. Different design and implementation decisions can both be abstracted using the same abstraction functions by choosing the more general specifications (requirements). For example, if the "refined requirements" in Table 1 are used for abstraction generation, another implementation that using the following moving strategy can also be abstracted the same abstraction functions as the SMV concrete model. Therefore, depending on the needs of the development, the more general specifications (requirements) should be used for system integrity verification and the more specific specifications (requirements) should be used for module (component) verification
The moving strategy: if there exists requests and the top floor is not reached, keep moving up; if there exists requests and the ground floor is not reached, keep moving down; if the top floor or the ground floor is reached, flip moving direction.

3. **Can we trust the abstracted model?** In the context of this project paper, the degree to which we can trust the verification results from an abstracted model largely depends on the coverage, the consistency, and the integrity of the specifications (requirements). That is to say, the abstracted model cannot guarantee the properties that are not covered by the specifications, and the verification results are hard to predict if the specifications (requirements) contain conflicting concepts, decisions, or statements.

4. **What if the abstracted model is still too huge to be verified?** It is usually the case that when the system is extremely huge and complicated such as Windows operating system, the specifications (requirements) are too much to be used for generating the entire abstracted model. One possible approach is to combine compositional reasoning with abstraction.

References

[1] F. Balarin and A. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer Aided Verification*, pages 29–40, 1993.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. Microsoft Research.

[3] A. Cal, D. Calvanese, G. D. Giacomo, and M. Lenzerini. A formal framework for reasoning on uml class diagrams. In *13th Int. Sym. on Methodologies for Intelligent Systems*, 2002.

[4] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[5] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Lauback, C. S. Pasareanu, Robby, H. cheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *International Conference on Software Engineering*, pages 177–187, 2001.

[6] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Computer Aided Verification*, pages 72–83, 1997.

[7] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[8] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *TACAS*, pages 2–17, 2003.