Chapter 1 Introduction

Automatic verification techniques such as model checking have been viewed as promising method to ensure the quality of complicated system. Many hard-to-detect errors, such as deadlocks, can be manifested by these techniques. In past decades, considerable progress has been made in these techniques. Several prototype tools, such as SPIN[12][13] , SMV[16] , have been built and applied to many software or hardware systems. In this paper we focus on software verifications.

1.1 Overview

Typically, software has more states than hardware. The wide variety of software designs makes software verification a more difficult task than hardware. For example, famous Binary Decision Diagram (BDD) [3] which is widely used in hardware verification to save memory resources may have no obvious merits in software verification (see Corbett's work in [7]). Besides, modeling a software system requires much more efforts, experiences, and human wisdom. In that work [7], Corbett also found subtle modeling differences for the same system may produce obvious differences in the results of analysis in different tools. However, the objective of Corbett's work is to compare performance of various verification tools, so, finding

and eliminating these differences to have a fair comparison of verification tools are his first priority.

In principle, most verification tools share the similar technology in exploring reachable states. They can be used for either software verification or hardware verification. Nevertheless, software verification researchers often prefer one tool over another tool because of the tool capability for modeling software. For example, SPIN provides a model description language (MDL) called Promela, which has syntax close to a high-level programming language. Although it is originally designed to verify communication protocols, many researchers have chosen it to verify concurrent programs written in Ada, Java and C.

We use tool SPIN as an example. In a post-mortem scenario, target programs are first abstracted and modeled into a Promela code. Later, in the analysis stage, the Promela code is translated into some form of labeled transition system (LTS) by SPIN to explore the reachable states. The Promela code or the abstracted LTSs are often called "*model*", because they are not the actual systems with real behaviors. The abstraction and modeling process requires human wisdom and experiences and is error-prone. So, in [14], Holzmann argued that a blindly derived mode (for example, either by a naïve automatic/manual extraction) is unlikely to work for verification in most cases. In other words, constructing an efficient and correct model requires

human wisdom from experienced personals.

To bridge the gap between programming languages and model description language, tools such as Bandera [8] and Pathfinder [10] have been developed to automatically extract models from Java source code. Their goal is to model-check Java source code by smartly extracting a model from Java code for analysis engines like SPIN. Bandera also introduces slicing techniques to abstract away the program behaviors that do not concern the interested properties (particularly liveness properties) so that the state explosion problem can be alleviated. In our opinions, these progresses mark an important milestone for automatic software verification.

Despite the progress described above, the fundamental barriers of verification, however, still remain. First, verification tools which analyze all processes at once are inevitably limited by the PSPACE lower bound in worst case; that is, the number of reachable states grows exponentially as the number of processes increases, In other words, any attempt to alleviate the state explosion is bound to fail in general but may work for some cases. For example, Bandera uses property to guide the program slicer to slice away the behaviors that are not concerned by the property, particular the liveness properties defined in linear-time temporal logic (LTL) formula. Such approach does not work for property like freedom of deadlocks. Reachable deadlocks must be manifested from all the possible behaviors.

To tackle the state explosion problem, a promising approach is compositional analysis. Compositional analysis avoids state explosion by dividing a whole system into many subsystems. Then, the techniques described above are used to analyze these subsystems. Ideally, the analysis of each subsystem would produce manageable and smaller state space and then each subsystem can be replaced and smaller state space and then each subsystem can be replaced by a simple interface process. The process is continued by combining the analysis of subsystems into a larger subsystem in a hierarchical fashion until the whole system is analyzed. Unfortunately, this ideal scenario seldom happens in practical cases. In many systems, no feasible hierarchies exist in their as-built architecture. The power of divide-and-conquer is prohibited by the system architecture.

In [4], Cheng et. al. proposed an approach called *model refactoring* to enable compositional analysis for systems which are originally prohibited by their as-built architecture. The refactoring consists a set of transformations. Each transformation maintains the behavioral equivalence (weak bisimulation) of the model. By applying a sequence of transformations, a model *P* is gradually transformed into a model *P'* with new structure which is more amenable to compositional analysis. The key transformations are to decompose centralized, complicated behaviors of a process into several small new processes while behavioral equivalence is preserved. In [5], Cheng,

Huang, and Pan described the basic tool support for refactoring and showed that a refactored elevator system can be analyzed up to hundreds of elevators but global analysis and compositional analysis (without refactoring) can only analyze up to 4 elevators. However, there are behaviors which our refactoring engine fails to refactor. These behaviors typically use array to implement arbitrary abstract data types. Our refactoring engine has no enough information to recognize the behavioral patterns in order to apply the transformations correctly. This problem initiated the work of this thesis, but later we found it is a more general issue (i.e., not only concerned by refactoring).

The second fundamental barrier is: Automatic model extraction may not change the fact that original code is written in a way that is “bad” for analysis engine, particularly when abstract data types are implemented by array. In this thesis, we show two functionally equivalent process behaviors with two modeling choices can produce significant differences in analysis, particularly compositional analysis with refactioring. In other words, analysis is sensitive to modeling choices. To address the problem, we support abstract data types in an *rc-Promela* language the front-end language of *COCOV*. We argue that encouraging the use of abstract data types and avoiding the use of array in modeling complicated processes can reduce feasible modeling choices, therefore, lessening the sensitivity of analysis. More importantly,

using abstract data type to write process behaviors encourages the code to speak for itself. Our *COCOV* which includes *rc-Promela* translator and refactoring engine can mechanically make the best decision to translate code into state graph and decompose process behaviors.

1.2 Organization

This thesis is organized as follow. In chapter 2, we describe background of model checking, communicating finite state machine, related tools, compositional analysis, and model refactoring. In chapter 3, we give two examples to explain why analysis is sensitive to modeling choices. Chapter 4 describes our work towards to supporting abstract data types in model description language. Finally, we end the thesis with discussion, future work, and conclusions in section 5 and 6.