Chapter 1 Introduction

1.1. Introduction

With the inevitable increase in complexity of both hardware and software systems, the quality of software is more difficult to be assured. To address the issue, automated finite-state verification techniques and tools have developed for the past decades. Although several verification techniques have been proposed, state-space explosion problem remains to be the main obstacle. As the size of systems increased, the state-space of these systems will grow exponentially, which is known to be a PSPACE problem. A variety of techniques have been proposed to mitigate state-space explosion problems. Nonetheless, approaches to increasing the size of system that can be accommodated in a single analysis step must eventually be combined with effective compositional techniques [2][5][6] that divide a large system into smaller subsystems, analyze each subsystem, and combine all the results of these subsystems to verify the full system.

Seemingly, compositional analysis could avoid state-space explosion since it does not require the construction of the full system state-space. However, compositional analysis can’t be applied to many systems because their as-built structure may not be suitable for compositional analysis. Systems must contain loosely coupled components so that every component can be replaced by a simple interface process in the incremental analysis. Besides, the processes in a component must not yield intractable analysis. Unfortunately, an ideal structure like this seldom

exists in practice. It is impractical to ask designers to structure a design in the beginning for the purpose of obtaining correctness. This is why compositional analysis has been prompted for years as the most promising technique for analyzing large-scale systems, but its applications are still quite limited to particular systems.

If it is difficult to prove the correctness of a program under its original design, one may need to prove the correctness on a transformed, equivalent version of the program. This is a notion known as program transformation, which has been widely studied in the area of the functional and logic languages [7][8]. Here, we apply the idea to transform finite state models to aid automated finite-state verification. In [3], Cheng proposed an approach for obtaining, starting from a model *P*, a semantically equivalent one, which is "more amenable to compositional analysis" than *P*. It consists in building a sequence of equivalent models, each obtained by the preceding ones by means of the application of a rule. The rules are aimed for restructuring the as-built structure which is not suitable for compositional analysis. The goal is to obtain a transformed model whose structure contains loosely coupled components, where processes in each component do not yield state explosion. This transformation is called *refactoring*. In [3], the process of refactoring is semi-automated, the designer must choose from among a set of idiomatic transformation, but each transformation is applied and checked automatically. Meanwhile, the transformations described in [3] were derived on an ad hoc basis; they are unlikely to be automated and applicable for general systems.

1.2. Objective

To automate refactoring, here we proposed an implementation to accommodate previous ad hoc transformations, extend refactoring to larger class of systems, and

provide automated tool support. Our approach is to adopt Promela as our front-end language. Promela is a well-known design language due to the popularity of SPIN [10]. We select a subset of Promela's syntax and add some keywords for refactoring. These keywords are used to automate refactoring transformation. The syntax is called *rc-Promela*. We build a parser which can parse *rc-Promela* models to construct an abstract syntax tree（AST）. Then, in [9], Huang uses the AST to generate CCS state graph with or without refactoring transformation.

There are several problems we need to address before the AST is used for generating CCS state graphs with refactoring. First, we need to identify which variables in the process are "relevant" to the CCS state graph generation. Using variables that are not "relevant" to CCS state graph generation may produce larger size of states/transitions. We propose an approach that uses data flow analysis to traverse abstract syntax tree to identify such variables which are "relevant" to generation of CCS states/transitions. With those variables we can generate compact CCS state graphs.

Figure 1.1 shows the process flow of automated refactoring. In the beginning, we use *rc-Promela* as a design language to define processes. Next, our parser translates each process into CCS finite-state graphs. If *refactoring* is enabled, the parser produce state graphs into small units called segments. Refactoring transformations read segments to construct new processes, while behavior equivalence is preserved. With the presence of new processes, a new architecture is formed, which may be more suitable for compositional analysis.

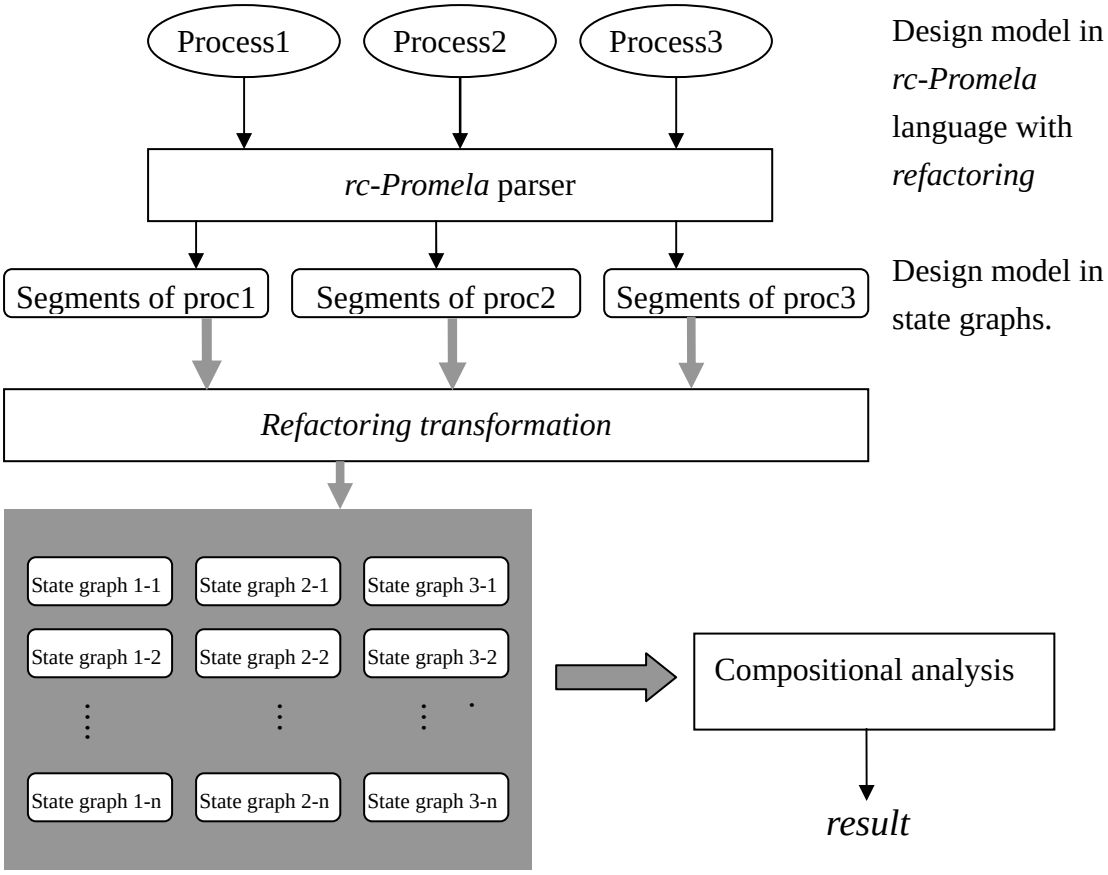

Figure 1.1. Process flow of automated refactoring.

1.3. Thesis organization

This thesis is organized as follows: In Chapter 2, we provide the background starting from basis of verification, existing software verification tools, compositional analysis, and to the basis of data flow analysis. In Chapter 3, we give an overview of refactoring. From Chapter 4 to Chapter 6, we described implementation of our research. In Chapter 4, we describe to build *rc-Promela* parser and construct AST. Chapter 5 describes new functionality that we added in *rc-Promela*. In Chapter 6, we describe a method which uses data flow analysis to generate compact CCS state graphs. Several examples are given in Chapter 7 to show the effectiveness of our approach. Finally, we end the thesis with conclusions and future work in Chapter 8.