Chapter 1

Introduction

The current trend for the development of large parallel systems indicates that the design of concurrent programs that run reliably and efficiently on networks of interconnected computers will remain an important challenge for the foreseeable future, as the size and complexity of such programs and networks will continue to grow.

The complexity of large systems is usually managed by formulation of appropriate design principles that permit the development of a system by composing smaller subsystems, and by automated aids to assist in the design and the development of such systems. Despite many research efforts over the last decade that have resulted in the formulation of design formalisms and the construction of support systems, the design and verification of complex concurrent software systems remain a difficult task.

In this work we focus on providing suitable automated assistance for design and verification of concurrent programs. We present our contributions to the area of formal verification of concurrent programs in section 1.2; here, we discuss briefly the notions of design and verification of such programs in general terms and provide a classification of different approaches that have been taken in order to build reliable systems.

1.1 Design and Verification of Concurrent Programs

Over the last decade the area of design and verification of concurrent systems has received an enormous amount of interest by researchers worldwide. Many different formal systems for reasoning about concurrent programs have been defined, different languages have been introduced for writing such programs, and many methodologies for effective designs have been proposed.

There are several criteria according to which methods for verifying concurrent programs can be described and differentiated. In the following, we discuss some of these criteria, explain how our approach to verifying concurrent programs can be characterized according to these criteria, and suggest how our work contributes to the current practice of program design and verification.

Simulation versus Formal Verification: The term verification has different meanings for different people: in most scientific and academic environments program verification refers to the task of formally establishing that a given program satisfies a given specification, or is somehow equivalent to another program; in industrial environments, however, verification is often synonymous with program validation by testing or simulation.

While for very small sequential programs a simulation-based method can establish correctness, it is impossible to make such a claim for larger, in particular *reactive* programs, for which the number of possible interactions or orderings of events is so large that only a negligible part of them can be validated by simulation. Such a validation can increase a designer's confidence in the program; it cannot, however, establish its correctness for all possible executions.

A Posteriori versus Design Oriented Verification: Traditionally, design and verification are viewed as separate tasks, with verification being attempted after a program has been written. This *a posteriori* verification has two major limitations:

first, no guidance is provided to the designer to come up with a program that meets a given set of specifications; second, many design decisions are no longer visible in the final program description, thereby making the verification task often excruciatingly difficult.

This situation can be contrasted with a design process in which the design and the verification of a program are performed hand-in-hand. Ideally, the program construction proceeds by repeated transformations from the initial specifications, where each transformation is guaranteed to preserve required properties, thereby establishing an implicit proof of correctness. Even if such a systematic way of deriving a program from its specification and proving it at the same time is not feasible – be it due to the size of the program, due to lack of a suitable formalism, or due to a certain inexperience of the designer – such an attempt leads to stating the assumptions and assertions about the program explicitly. This is a way to document design decisions and to prevent this form of design knowledge from disappearing between the lines of the final program text.

Interactive versus Automated Methods: Another way of characterizing a verification method is by the degree of automation of the verification process. At one end of the spectrum are fully automated methods that establish correctness of a program (or the lack thereof) without any user intervention. At the other end are paper-and-pencil methods (possibly enriched with some mechanical support) that provide a method for verifying programs, but leave the discovery of a suitable proof to the user.

A fully automated verification method does not exist for all kinds of systems, but it is obvious that a high degree of automation is especially important for effective use in an industrial production environment.

Monolithic versus Compositional Approaches: While a monolithic verification method attempts to deal with a program as a whole, a compositional method

makes it possible to decompose a program into smaller components, verify the individual components separately, and then combine the verification results for the components to assert properties of the entire program. It is generally accepted that the compositionality of a method is crucial in order to be able to handle large and complex systems of concurrent programs.

Nature of Programs: Finally, verification methods can be distinguished by the type of programs they can be applied to, and by the application domains for which these programs are designed. Different realizations encompass software protocols, hardware circuits, synchronous programs, or asynchronous ones. It is expected that a successful verification method capable of dealing with very complex systems will have to take advantage of specific features of the program realization and of the application domain.

1.2 Overview of the Thesis

With respect to the above criteria we can classify our research interests as follows: we want to improve the applicability of formal verification; our approach should allow the designer to take advantage of design knowledge in a direct way; our method should be mechanically supported with as much automation as possible and as little user intervention as necessary; our work should be placed in a context that supports compositional design and reasoning; finally, our methods should be applicable to a wide range of systems and applications.

Our goal, in summary, is to improve the practicability of formal verification for a wide range of concurrent systems by demonstrating how the judicious use of available design knowledge can improve the efficiency and the effectiveness of an otherwise automated verification method.

1.2.1 Foundations

As a starting point for our work we need both a suitable formalism and a powerful verification method.

A formalism suitable for our proposed work is required to have a clean and rich logical structure, has to have simple yet powerful concepts supporting both the design of concurrent programs and reasoning about them, and has to be capable of dealing with many different systems. Out of the many existing theories for design and verification of concurrent systems, very few meet these requirements. UNITY ([CM88]) has been found to be such a formalism, mainly because of its rich proof system, general computational model, economy of concepts, compositional structure, and design methodology.

As far as the choice of a verification method is concerned we want to build on a method that has proven to be practically useful, that can deal with sizable programs, and that can be used in a fully automated way. The verification technique called *model checking* ([CES86]) meets these requirements and has, furthermore, achieved great practical relevance.

1.2.2 Contributions

This dissertation makes three main contributions to the area of verification of concurrent programs:

1. We have combined UNITY logic with model checking to obtain efficient model checking procedures for safety and basic progress properties. These procedures take advantage of the structure of the UNITY proof system; they allow the user to utilize state-based design knowledge, in form of invariants, to replace expensive fixpoint computations by evaluations of simple local verification conditions.

2. We have developed an extension of UNITY logic for progress called *generalized progress* that allows the user to provide hints as part of progress properties. Hints take on a form similar to regular expressions; they characterize how progress is achieved in a computation. We have developed a logic to reason about such generalized progress properties. Additionally, the verification procedure can take advantage of the hints to improve the efficiency of checking progress properties. The theory we have developed comprises a predicate transformer semantics, a deductive proof system, the characterization of the algebraic structure of the new properties, and an operational semantics.

3. We have implemented the model checking procedures for the extended logic as part of a verification system called the *UNITY Verifier System*. The system is both a useful tool for verifying concurrent programs, and an extensible platform for exploring future research ideas. Our preliminary experiments show that our approach can be substantially faster then existing model-checking procedures.

1.2.3 Structure of the Thesis

The remainder of this thesis consists of seven chapters and four appendices, and is organized as follows:

In chapter 2, *Preliminaries*, we introduce definitions and notation that we use in the formal development of our theory. We also present a brief summary of the UNITY notation and logic, and some facts about the verification technique known as *model checking*.

The first part of our work deals with the theory of verifying properties of UNITY logic: in chapter 3, *Model Checking for UNITY*, we show how combining UNITY logic and model checking results in a verification technique suitable for dealing with safety properties and basic progress properties efficiently. Verification of arbitrary progress properties is addressed in chapter 4, *A Generalization of Progress*,

where we develop the theory of generalized progress properties. In chapter 5, *Checking Progress Properties*, we demonstrate how the theory in chapter 4 can be used in a model checker.

The second part of our work is devoted to the practical application of our theory: in chapter 6, *The UV System*, we describe our implementation of the model checking algorithms based on the previously developed theory. In chapter 7, *Experimental Results*, we apply the UV system to a series of examples and demonstrate the advantages of our verification techniques.

We summarize our results in chapter 8, *Conclusions*, and discuss possibilities for extending the work presented here.

The four appendices provide additional information on various aspects of our work: in appendix A, *The UV Input Language*, we present the complete grammar of the input language of the UV system and discuss some extensions of it. A detailed description of the scripting interface of the UV system that makes the functionality of the system accessible to the user follows in appendix B, *The Tcl Interface*. An overview of the UV source files in appendix C, *The UV Source Structure*, is intended for those who may wish to extend our system. Finally, in appendix D, *Additional Proofs*, we include proofs of some of the theorems that were omitted in chapter 4.