Chapter 8

Conclusions

In this thesis, we have investigated how a combination of the programming notation and temporal logic of UNITY and the verification technique of model checking can aid in the verification of concurrent programs. We have shown that the structure of UNITY logic can be exploited to obtain efficient model checking procedures for safety and basic progress properties that take advantage of user-supplied state-based design knowledge. We have extended the UNITY logic by the theory of generalized progress, which we have used to improve the checking and the reasoning about progress properties by utilizing user-supplied action-based design knowledge. Finally, we have designed and implemented a model checker for UNITY that makes our methods available for practical use, and have demonstrated the advantages of our techniques by verifying several example programs.

It is important to note that the techniques we have developed and investigated in this work contribute mostly to the algorithmic and methodological aspects of program verification. In particular, they are orthogonal to issues related to the choice of suitable symbolic representations and – to some lesser degree – programming notations. For instance, it can be expected that improvements in representation techniques (OBDDs or others), can be combined with our methods to further increase the performance of program verification.

Our results on model checking for UNITY have not only been encouraging, they also suggest several research directions for future work. In the following section, 8.1, we briefly discuss some of these ways of extending our work. We conclude this thesis with some final remarks in section 8.2.

8.1 Directions for Future Research

Throughout this thesis we have pointed out several possible extensions to our work on model checking for UNITY. Among these are the improvement of some aspects of the implementation of the UV system, in particular of the symbolic representation by OBDDs, the application of our methods to larger examples, and the investigation of how our methods can be adapted to other logics and formalisms – including the combination of our theory with fairness assumptions other than the unconditional fairness of traditional UNITY logic [CM88, Rao95, Fra86]. In the following, we briefly discuss some additional topics for future research that can build on our work presented here.

Verification for Seuss

Recently, Misra has proposed a new discipline for multiprogramming [Mis94] called Seuss that is aimed at addressing both the need to reason about complex concurrent systems, and the requirement for efficient execution of the designed systems.

A principal problem in dealing with multiprograms is that the requirements for a modular structure (being essential for understanding a complex system) and for fine-grained interaction of the program components (being important for an efficient implementation) are contradictory in nature: when reasoning about a program one wants to be able to consider its components independently, whereas an efficient execution implies the possibility of interference.

Seuss addresses this conflict by disentangling *sequential* and *multiprogramming* aspects of concurrent systems. This approach is based on the observation

that concurrent systems typically consist of substantial sections of sequential code embedded into multiprograms. These sequential sections can be understood and reasoned about like sequential programs, in particular they can be given a *transformational* semantics based on pre- and postconditions. Furthermore, they can be regarded as *atomic* as far as the interference with other parts of the concurrent system is concerned. On the other hand there are multiprogramming aspects of concurrent systems that cause synchronization and waiting of different parts of a system. These multiprogramming parts of a concurrent system can be given semantics as reactive systems, reflecting their ongoing interaction with their environment. The computational model of Seuss separates the transformational from the reactive aspects of a concurrent system and allows for disciplined interactions between them.

For the Seuss programming model one can define computations in two ways. A *tight execution* maintains a single thread of control and is well suited for reasoning about a program. On the other hand, a *loose execution* of a Seuss program allows for a fine-grained asynchronous distributed implementation. Central to a realization of such an execution strategy is the notion of *compatibility* of atomic actions. Roughly speaking, two such actions are compatible if their order of execution can be serialized in the context of a certain class of program executions. Compatibility can be seen as a generalization of the notion of *commutativity* of procedures, which proves to be too restrictive for most practical forms of process interaction.

The fundamental theorem about Seuss relates the two notions of execution strategies by asserting that for every loose execution of a given program there exists an equivalent tight execution of the same program (conversely, every tight execution is trivially also a loose execution). By virtue of this theorem it becomes possible to reason about a program and deduce its properties in terms of the simple tight execution strategy, and still obtain all properties even for the more general and efficient loose executions.

While the model of computation of Seuss extends much beyond that of

UNITY with respect to hierarchical program structuring, communication by procedure calls, and the distinction between the sequential and concurrent aspects of a program, there are many features of UNITY logic that are applicable to the Seuss model as well. By virtue of the disciplined interaction of sequential and concurrent parts and the hierarchical structure of programs, the Seuss model is very suitable for the design of actual concurrent systems (e.g. GUI or window managers) that are well beyond the size and complexity of typical UNITY programs. Therefore, automated or interactive design support for Seuss can have a significant impact in both a theoretical and a practical way: it can take advantage of the additional structure of programs by providing a compositional theory of concurrency, and it can be used for a class of applications that could not be dealt with effectively using traditional formalisms. Work on defining a UNITY-like logic for reasoning about Seuss programs and their properties is under way ([Ada95, Mis96]).

Composition and Closure

The importance of compositional reasoning as a means for dealing with large systems is well understood, and several approaches have been proposed to exploit compositional techniques for model checking (e.g. [GL94]). The performance gains observed in checking safety and basic progress properties with the UV system are closely related to the inherent compositionality of UNITY logic. However, there is an even richer theory of compositionality of UNITY related to *closure* properties [Mis], that should be explored for its applicability to model checking. By hierarchically structuring programs and imposing syntactic restrictions (e.g. by typing and by access restrictions) on shared variables, a notion of program composition can be defined that makes it possible to deduce even progress properties of a composite system from properties of its components.

This work may lead to a simplification of the rely/guarantee style of reasoning [Col93, CK93, McM92]. Any results obtained in this direction will also be usable in

the context of the hierarchically structured program model of Seuss.

Model Checking of Parameterized Systems

A very desirable extension to traditional model checking is the ability to verify not just one fixed program but a whole class of parameterized programs simultaneously while not having to resort to general theorem-proving techniques.

Several ideas for dealing with parameterized systems have been investigated in the literature. On one hand, one can try to eliminate the parameterization altogether without the use of induction [EN95]; on the other hand, methods for eliminating induction in certain cases have been proposed by either choosing a canonical parameterized representation [GF93], or by reducing the parameterized checking problem to a few finite model checking problems of fixed size [KM89, McM92, MCB89]. Common to these approaches to reducing parameterized systems is their linear structure, i.e., the constant difference (by some measure) of the structures of successive instantiations of the parameterized system.

Two ideas for expanding the class of parameterized problems that can be handled in a (mostly) automated way should be investigated. First, the linearity requirement could be relaxed by developing a second- (or higher-) order induction scheme, in which the difference between successive instantiations is no longer required to be constant, but could be in the form of a parameterized system as well, requiring an inductive treatment of a lower degree. Secondly, a combination of model checking with the deductive system of UNITY logic could be attempted in such a way that an interactive proof checker could verify the validity of the transformation of proof obligations within the logic, while the model checker could deal with the generated finite state checking tasks. The idea of combining theorem proving and model checking is certainly not new ([Hun93, RSS95]), but the simplicity and the deductive system of UNITY logic might be used effectively.

8.2 Final Remarks

The formal verification of concurrent systems will remain a challenging research area for the foreseeable future. As concurrent systems, in particular safety-critical systems such as traffic or process control systems, continue to grow in size and complexity, we will have to find ways of extending formal methods to be able to cope with increasingly complex verification tasks. Arguably, there are three ways in which we can hope to improve the applicability of formal methods and the feasibility of their use in an industrial environment:

1. It will be of utmost importance to take advantage of the modular structure of systems in order to overcome the complexity problems related to the *state explosion problem*, i.e., the exponential growth of the size of state spaces when combining component programs to larger systems. In general, one does not have to find a modularization of a complex system, as any reasonable design method relies on some form of modularization in order to manage design complexity. However, it is necessary to devise methods for taking advantage of such modularizations, and to suggest suitable modularizations to the designer of complex systems.

2. The exploitation of design knowledge will prove to be crucial for verifying intricate components and systems. It seems that one cannot afford to ignore the information about a design, the assumptions that led to certain design decisions, and the designer's understanding of how and why he expects his program to meet its specifications. Much of this information is no longer present in the final program text, but was available at some point during the design of the program. This observation emphasizes again the need of performing design and verification hand-in-hand. It will be necessary to find ways of exploiting this design knowledge, even if it is not possible to formalize the entire design process.

3. It cannot be expected that a general design and verification method will be equally applicable to all different kinds of concurrent systems. Clearly, the requirements and properties of a highly synchronous circuit are very much different from the ones of an asynchronous communication protocol. It will be necessary to take advantage of the specifica of systems and application domains as part of a successful design and verification method.

It is widely recognized that the application of formal techniques to the design and verification of concurrent systems yields a degree of understanding of such systems and confidence in their correct and reliable operation that is unmatched by any other design method. However, the acceptance of formal methods in an industrial environment is greatly hindered by complexity issues, by the lack of training and experience in using formal methods, and – as a consequence – by the adherence to established design processes and tools, and the reluctance to adopt new ideas.

It would be naive to assume that the effective use of formal techniques in an industrial setting could be accomplished without having to redesign at least parts of the program development process. However, the use of simple and elegant formalisms that have highly automated support can lessen the impact of adopting formal methods and enlarge the class of practical problems that can be dealt with effectively. Our work constitutes a step in this direction, in particular by showing novel ways for taking advantage of user supplied design knowledge. Our approach can be considered successful if it is able to contribute to lowering the threshold of applying formal design and verification methods to practical problems, thereby helping to establish design methodologies that result in the construction of more reliable and maintainable systems.