Copyright

by

Markus Kaltenbach

1996

Interactive Verification Exploiting Program Design Knowledge: A Model-Checker for UNITY

by

Markus Kaltenbach, Dipl. Inf.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 1996

Interactive Verification

Exploiting Program Design Knowledge:

A Model-Checker for UNITY

Approved by
Dissertation Committee:

To My Parents Albert and Ingrid Kaltenbach

To Francesca

Acknowledgments

The work described in this thesis was possible only because of the guidance, encouragement, and support of many people. First and foremost, I want to thank my advisors, Jayadev Misra and Allen Emerson. I have been deeply influenced by their high standards of mathematical rigor and elegance, by their creativity and wisdom, and by their unfailing support from my first day in the Ph.D. program at the University of Texas at Austin.

I also want to extend my deepest thanks to the other members of my dissertation committee: to Edsger W. Dijkstra, who invited me to the Austin Tuesday Afternoon Club and taught me to strive for elegance in my work – if there is any beauty in this thesis, it is due to him; to Carl Pixley, who arranged for my summer position at Motorola and with whom I had many beneficial conversations; to Edmund Clarke, whose many ideas have greatly contributed to my work; and to the late Woody Bledsoe, who taught me to have confidence in my ideas, and who was a great inspiration much beyond the realm of academia. I consider myself very lucky to have enjoyed the great fortune and privilege of learning and working with these extraordinary teachers and researchers.

I want to thank my fellow graduate students for all the insightful discussions, for their help, and their companionship: Will Adams, Al Carruth, Rajeev Joshi, Edgar Knapp, Jacob Kornerup, Ingolf Krüger, Kedar Namjoshi, Lyn Pierce, Josyula R. Rao, Richard Trefler, and Anish Arora, who is also a most wonderful friend. My special thanks go to Rutger M. Dijkstra and Ernie Cohen: the discussions I had with Rutger M. Dijkstra during his visit at UT sparked important ideas for this thesis; Ernie Cohen provided several suggestions that helped to improve the thesis and was instrumental in solving a previously open problem about progress algebras.

This research was made possible by the financial support provided by the following institutions and agencies, which I gratefully acknowledge: IBM Corporation for the IBM Graduate Fellowship, the National Science Foundation for grants number CCR-9504190 and CCR-9415496, and The University of Texas at Austin for the Microelectronics and Computer Development Fellowship.

My very special thanks go to my family: my parents Albert and Ingrid Kaltenbach who never failed to encourage me to pursue my goals, and who provided me with all the opportunities and freedom to do so, even if it meant that I would be very far away from them; my wonderful sister Regine and her husband Reinhard Herzog, who have been my best friends over all these years. I am also very grateful to my friends Terri Behrle, Tina Kien, Michelle Kim, and Karen Shaffer, who all supported and encouraged me on countless occasions.

Finally, and most importantly, I want to thank Francesca Chiostri: she provided me with more motivation, encouragement, and emotional support than I could ever have asked for. Without her this work would not have been possible.

Markus Kaltenbach

The University of Texas at Austin
December 1996

Interactive Verification

Exploiting Program Design Knowledge:

A Model-Checker for UNITY

Publication No. ____________________

Markus Kaltenbach, Ph.D.
The University of Texas at Austin, 1996

Supervisors:
Jayadev Misra
E. Allen Emerson

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 systems will continue to grow. Verification techniques based on appropriate design formalism and complemented by mechanical support will play an important role for asserting the correctness and quality of these concurrent systems.

In this dissertation we focus on providing suitable automated assistance to the design and verification of concurrent systems by developing a model checker for finite state programs and propositional UNITY logic. Combining the verification technique of model checking with the temporal logic of UNITY was motivated by two goals, namely to exploit the simplicity and structure of UNITY logic as to

provide efficient checking algorithms for a mostly automated verification, and to allow the user to interactively supply design knowledge in order to improve the system performance.

These goals have been met in three ways: (i) we have derived a model checking procedure for safety and basic progress properties that is based on the proof rules of UNITY logic, increases the efficiency of verification by making it possible to replace fixpoint computations by simple verification checks, and, moreover, takes advantage of state-based design knowledge in the form of invariants; (ii) we have developed and formally investigated a new theory of generalized progress, in which action-based hints can be provided to indicate how progress is achieved and which can be used to improve the efficiency of checking and reasoning about arbitrary progress properties; (iii) finally, we have implemented the resulting model checking procedures as part of the UNITY Verifier System and have used our implementation to demonstrate the improved verification performance with several examples.

Contents

List of Figures

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.

Chapter 2

Preliminaries

The formal development of our theory in chapters 3, 4, and 5 requires some notation and definitions which we introduce in section 2.1. We also give a brief overview of two of the main ideas our work is built on: the temporal logic and programming notation UNITY [CM88] is summarized in section 2.2, and the verification technique of *model checking* [CE81](cf. [QS82, CES86]) is introduced in section 2.3.

2.1 Notation and Terminology

In the following we introduce some notation and collect some basic results that are used throughout this thesis. We begin with some notational conventions in section 2.1.1, followed by a description of the fundamental concepts of programs, states and predicates in section 2.1.2, and of the calculational proof format used in this thesis, in section 2.1.3. The important notion of predicate transformers and some of their properties are introduced in section 2.1.4, followed by the definition of extreme solutions of equations and the characterization of some properties of such solutions in section 2.1.5. Some definitions and notation concerning regular expressions, in section 2.1.6, conclude the presentation of notation and terminology.

2.1.1 Notational Conventions

We use formulae of propositional and predicate calculus following the conventions laid out in [DS90]; in particular, the infix "." is used for denoting function application. As usual, function application is left-associative. The following boolean and arithmetic operators, with their usual meanings, are used for writing expressions; we list them in order of increasing binding power (operators in the same line have the same binding power):

$$\begin{array}{c} \equiv \quad \not\equiv \\ \Leftarrow \quad \Rightarrow \\ \wedge \quad \vee \\ \neg \\ = \quad \neq \quad < \quad \leq \quad \geq \quad > \\ + \quad - \\ . \end{array}$$

Additional operators for relations and regular expressions will be introduced as needed. We make sure that such an introduction does not cause ambiguities in the operator precedence; for instance, when adding regular expression operators (cf. section 2.1.6), precedence can be resolved by taking type information (i.e., regular expressions versus predicates) into account.

Quantification

For quantified formulae of predicate calculus we use the following notation: for $Q \in \{\forall, \exists\}$ we denote by

$\langle Q i : r.i : t.i \rangle$

the quantification over all $t.i$ for which i satisfies $r.i$. We call i the *dummy*, $r.i$ the *range*, and $t.i$ the *term* of the quantification. If the range is understood from the context, we may omit it.

This notation is generalized for arbitrary associative and commutative binary operators: for any such operator **op** we write

$$\langle \mathbf{op}\, i : r.i : t.i \rangle$$

to denote the value of any expression obtained by substituting the instances of the dummy satisfying the range predicate in the term expression and folding the resulting expressions using **op**. Since **op** is associative and commutative this value is well defined. If **op** has a unit element u, we permit the range to be empty, in which case the denoted value is u. Instead of introducing a binary set constructor we use the more convenient notation

$$\{i : r.i : t.i\}$$

to denote set comprehension, namely the set of all $t.i$ where the dummy i ranges over all values satisfying $r.i$.

Sequences

We use $\mathbf{N}$ to denote the set of natural numbers. For any natural number n we denote by $\mathbf{Z}_n$ the set $\{i : 0 \leq i < n : i\}$ of the first n elements of $\mathbf{N}$. In the following let S be any non-empty set. A finite *sequence* of length k over S is a mapping from $\mathbf{Z}_k$ to S. An infinite sequence over S is a mapping from $\mathbf{N}$ to S and has length ω. We write $\langle\rangle$ for the empty sequence (i.e., the sequence of length 0), and $|\sigma|$ for the length of a sequence σ. We denote by S^* the set of all finite sequences over S, by S^+ the set of all non-empty finite sequences over S, by S^ω the set of infinite sequences over S, and by S^∞ the set of all sequences over S, i.e., $S^\infty = S^* \cup S^\omega$.

We also define a few functions on sequences in S^∞: **finite** maps sequences to boolean values indicating whether a sequence is finite ($\mathbf{B}$ denotes the boolean domain):

$$
\begin{array}{lll}
\mathbf{finite}: & S^\infty \to \mathbf{B} & \\
\mathbf{finite}.\sigma & \equiv & |\sigma| < \omega
\end{array}
$$

The function **tail** strips a sequence of its first element (if present):

$$
\begin{array}{llll}
\mathbf{tail}: & S^\infty \to S^\infty & & \\
\mathbf{tail}.\langle\rangle & = & \langle\rangle & \\
|\,\mathbf{tail}.\sigma| & = & |\sigma| - 1 & \text{if} \quad 0 < |\sigma| < \omega \\
\mathbf{tail}.\sigma.i & = & \sigma.(i+1) & \text{if} \quad 0 < |\sigma| < \omega \ \wedge \ 0 \leq i < |\sigma| - 1 \\
|\,\mathbf{tail}.\sigma| & = & \omega & \text{if} \quad |\sigma| = \omega \\
\mathbf{tail}.\sigma.i & = & \sigma.(i+1) & \text{if} \quad |\sigma| = \omega \ \wedge \ i \in \mathbf{N}
\end{array}
$$

The binary concatenation operator $+\!\!+$ appends two sequences:

$$
\begin{array}{llll}
+\!\!+: & S^\infty \to S^\infty \to S^\infty & & \\
|\sigma +\!\!+ \tau| & = & |\sigma| + |\tau| & \text{if} \quad |\sigma| < \omega \ \wedge \ |\tau| < \omega \\
|\sigma +\!\!+ \tau| & = & \omega & \text{if} \quad |\sigma| = \omega \ \vee \ |\tau| = \omega \\
(\sigma +\!\!+ \tau).i & = & \sigma.i & \text{if} \quad 0 \leq i < |\sigma| \\
(\sigma +\!\!+ \tau).i & = & \tau.(i - |\sigma|) & \text{if} \quad |\sigma| < \omega \ \wedge \ |\tau| < \omega \ \wedge \ |\sigma| \leq i < |\sigma| + |\tau| \\
(\sigma +\!\!+ \tau).i & = & \tau.(i - |\sigma|) & \text{if} \quad |\sigma| < \omega \ \wedge \ |\tau| = \omega
\end{array}
$$

We also extend the notation for quantified expressions to arbitrary binary operators where the ranges are sequences, as follows:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\langle \mathbf{op}\, i : \sigma : t.i \rangle} \\
\quad = & u & \text{if} \quad |\sigma| = 0 \text{ and } u \text{ is a unit of } \mathbf{op} \\
\quad = & t.(\sigma.0) & \text{if} \quad |\sigma| = 1 \\
\quad = & t.(\sigma.0) \ \mathbf{op} \ \langle \mathbf{op}\, i : \mathbf{tail}.\sigma : t.i \rangle & \text{if} \quad |\sigma| > 1
\end{array}
$$

Finally, we adopt the convention that all formulae are universally quantified over all free variables occurring in them.

2.1.2 Predicates, Programs, and States

The intended model for the theory developed in this thesis – both for defining the underlying computational domain, and for the treatment of the operational semantics of the proposed logic – is the structure of *total deterministic labeled state transition systems* (TDLSTS): they are commonly used in modeling concurrent systems, they are very general in that they allow both state-based and action-based formulations, and they serve as a model for both UNITY logic and our extensions of it.

Programs

Formally, a *labeled state transition system* F is a tuple (S, A, T, I) where S is a non-empty set of states, A is a finite set of actions, $T \subseteq S \times A \times S$ is a labeled transition relation, and $I \subseteq S$ is the non-empty set of initial states. For an LSTS F we write $F.S$ to denote its set of states, and similarly $F.A$, $F.T$, and $F.I$ for its other components. If $(s, \alpha, t) \in F.T$, we say that t is an α-successor of s. For a labeled state transition relation T and an action α, we denote by T_α the α-successor relation, i.e., the projection onto the first and third arguments of the restriction of T to the second argument α: $((s,t) \in T_\alpha) \;\equiv\; (s, \alpha, t) \in T)$.

For our intended model of computation we require actions to be *total* and *deterministic*: an action α is total if it is enabled in every state of the state space, i.e., if every state s has at least one α-successor; it is deterministic if for any state s there is at most one α-successor. A TDLSTS is a labeled state transition system in which all actions are total and deterministic. It follows that in any TDLSTS F for any action α the α-successor relation $(F.T)_\alpha$ is a total function, which we henceforth associate with α. We thus write $\alpha.s$ to denote the unique α-successor of state s.

Every TDLSTS we encounter in this thesis is described as a program in the UNITY notation (cf. section 2.2, where we discuss briefly how such a program defines a TDLSTS). Due to the close correspondence between a UNITY program and the corresponding TDLSTS, henceforth, we use the term *program* to refer to both concepts.

A *run* of a program F is a finite or infinite sequence of actions, i.e., an element of $(F.A)^\infty$. We associate with every finite run x a function[1] from $F.S$ to $F.S$, mapping a state s to the state $x.s$ obtained by executing the actions of x in order:

$$\begin{array}{lcl} \langle\rangle.s & = & s \\ x.s & = & (\mathbf{tail}\,.x).((x.0).s) \text{ if } |x| > 0 \end{array}$$

An *execution* of program F is a pair (s, x) consisting of an initial state s in $F.I$ and an infinite run x of F. An execution (s, x) of F is *unconditionally fair* if and only if every α in $F.A$ occurs infinitely often in x, i.e., for every α in $F.A$ the set $\{i : i \in \mathbf{N} \wedge x.i = \alpha : i\}$ is infinite.

The Assertion Language

For any program F we assume the existence of an assertion language for denoting state predicates over the state space of F, that is sufficiently expressive to characterize the sets of states that arise in various constructions of our theory. We denote by $\mathcal{P}_F$ the class of all state predicates over the state space $F.S$. We assume that the assertion language is equipped with a semantics that determines whether any given state in $F.S$ satisfies a given state predicate: for a predicate p in $\mathcal{P}_F$ and a state s in $F.S$ we denote by $s \models p$ the fact that s satisfies p. With such a semantics in place, every predicate p in $\mathcal{P}_F$ characterizes a subset of $F.S$, namely the set of states satisfying p; also, by the above assumption, every *interesting* set of states we encounter can be characterized by some predicate in $\mathcal{P}_F$. Therefore, we often do not distinguish between predicates in $\mathcal{P}_F$ and sets of states of $F.S$.

Given a state s in $F.S$ and a run x of F, we can consider the sequence of

[1] We, thereby, overload the sequence x as both a mapping from states to states and a mapping from some naturals to actions. Due to the distinct domains of states and naturals this does not pose a problem.

states obtained by executing x action-by-action starting in s. For any such s and x and any predicate p in $\mathcal{P}_F$ we introduce the notation $(s, x) \models p$ to denote the situation in which execution of x starting in s reaches some state (after a finite number of steps) satisfying p. Formally,

$$(s, x) \models p \quad \equiv \quad \langle \exists y, z : x = y +\!\!+\, z \wedge \mathbf{finite}\,.y : y.s \models p \rangle$$

The Everywhere Operator

It is often desirable to quantify a predicate universally over the state space of a program F in order to assert that the predicate is satisfied by every program state. Following [DS90] we use the *everywhere* operator, a unary operator that has all the properties of universal quantification over a non-empty range[2]. There are two different ways in which the range of this universal quantification can be defined, either as quantification over the full *syntactic* state space, (i.e., over $F.S$), or as quantification over the *reachable* part of the state space consisting of all states that are reached from some initial state by some finite run (i.e., over the reflexive transitive closure of $F.I$ under the relation $\langle \bigcup \alpha : \alpha \in F.A : (F.T)_\alpha \rangle$).

The distinction between syntactic and reachable state space is important when defining the semantics of UNITY programs with non-trivial initial sets, where a complete axiomatization requires the so-called *substitution axiom* that corresponds to limiting observations of the program behavior to only its reachable set of states (for a detailed discussion of the substitution axiom and completeness of UNITY logic, the reader is invited to consult [San91, Kna92]). As a consequence, we use two variations of the everywhere operator: we surround a predicate by square brackets $[\,]$ to denote quantification over the reachable part of the state space, and by double square brackets $[\![\,]\!]$ to denote quantification over the full syntactic state space. The notion of reachable state space is formalized in section 2.1.5, where we introduce the concept of extreme solutions of predicate transformers.

[2] Recall that we require the set of initial states of any program to be non-empty.

2.1.3 Proof Format

Most of our proofs will be conducted in a *calculational* style, in which proof steps consist of a number of syntactic transformations rather than being based on semantic reasoning. In particular, for manipulating formulae of the predicate calculus we use a proof format that was proposed by Dijkstra, Feijen, and others, which greatly facilitates this kind of reasoning (for a thorough discussion of this format, see [DS90]).

In this format a proof is a sequence of formulae related by $\equiv$ (*equivales*), $\Rightarrow$ (*implies*), or $\Leftarrow$ (*follows-from*), interspersed with *hints* justifying the transformation from one formula to the next. For instance, a proof that $[A \Rightarrow D]$ could be written in our format as

$$
\begin{array}{ll}
 & D \\
\Leftarrow & \{\text{hint why } [C \Rightarrow D]\} \\
 & C \\
\equiv & \{\text{hint why } [B \equiv C]\} \\
 & B \\
\Leftarrow & \{\text{hint why } [A \Rightarrow B]\} \\
 & A
\end{array}
$$

The use of $\Leftarrow$ instead of the more traditional $\Rightarrow$ can result in proofs that are easier to understand. It is often the case that performing a proof step in one implication direction requires a significant amount of clairvoyance, whereas the step is dictated by the syntactic form of the involved formulae when performed in the opposite direction.

We assume a certain familiarity with the predicate calculus (e.g. as presented in [DS90]) and will justify many common transformations based on predicate or propositional calculus with the hint *predicate calculus*.

2.1.4 Predicate Transformers

A predicate transformer for a program F is a mapping[3] from $\mathcal{P}_F^n$ to $\mathcal{P}_F$ for some n in $\mathbf{N}$. We extend the $[\,]$ and $[\![\,]\!]$ operators to predicate transformers by defining their application to any unary predicate transformer τ by

$$
\begin{aligned}
[\tau] &\equiv \langle \forall Z : Z \in \mathcal{P}_F : [\tau.Z] \rangle, \\
[\![\tau]\!] &\equiv \langle \forall Z : Z \in \mathcal{P}_F : [\![\tau.Z]\!] \rangle,
\end{aligned}
$$

and similarly for predicate transformers of higher arity.

An important property of a predicate transformer is the extent to which it distributes over disjunction or conjunction of predicates. A predicate transformer τ is said to be *conjunctive* with respect to a set S of predicates if and only if

$$[\![\langle \forall p : p \in S : \tau.p \rangle \quad \equiv \quad \tau.\langle \forall p : p \in S : p \rangle]\!]$$

Similarly, a predicate transformer τ is said to be *disjunctive* with respect to S, if and only if

$$[\![\langle \exists p : p \in S : \tau.p \rangle \quad \equiv \quad \tau.\langle \exists p : p \in S : p \rangle]\!]$$

There are several notions of junctivity[4] depending on S. In particular, τ is called *finitely conjunctive* (*finitely disjunctive*) if it is conjunctive (disjunctive) for all non-empty finite S, it is called *and-continuous* (*or-continuous*) if it is conjunctive (disjunctive) for all non-empty linear[5] S, and it is called *universally conjunctive* (*universally disjunctive*) if it is conjunctive (disjunctive) for arbitrary sets S. It is well known that conjunctivity and disjunctivity over all non-empty, finite, and linear sets S are the same and coincide with the traditional notion of monotonicity of τ. A complete discussion of various junctivity properties can be found in [DS90].

[3] As usual, $\mathcal{P}_F^n$ denotes the n-times Cartesian product of $\mathcal{P}_F$.

[4] We use the term *junctive* and its noun form to stand for either *conjunctive* or *disjunctive*.

[5] A set of predicates is called *linear* if its elements can be arranged in a monotonic (strengthening or weakening) denumerable sequence.

In the following, we make use of three predicate transformers characterizing the semantics of actions [Dij75]. These are

Weakest Precondition (wp): For an action α and a state predicate q, $\mathbf{wp}.\alpha.q$ characterizes precisely those initial states beginning in which each execution of α terminates in a state satisfying q.

Weakest Liberal Precondition (wlp): For an action α and a state predicate q, $\mathbf{wlp}.\alpha.q$ characterizes precisely those initial states beginning in which each execution of α either fails to terminate or terminates in a state satisfying q.

Strongest Postcondition (sp): For an action α and a state predicate q, $\mathbf{sp}.\alpha.q$ characterizes precisely those final states for which there exists an execution of α starting in some state satisfying q and terminating in that final state.

Since the actions we consider are total and deterministic, these predicate transformers satisfy the following conditions. Proofs of these theorems can be found in [DS90]:

1. Since actions always terminate, we have for all actions α: $[\![\mathbf{wp}.\alpha \equiv \mathbf{wlp}.\alpha]\!]$. Henceforth, we will use **wp** only.

2. For every action α, the predicate transformer $\mathbf{wp}.\alpha$ is universally conjunctive (again due to actions being total).

3. For every action α, the predicate transformer $\mathbf{wp}.\alpha$ is universally disjunctive (because actions are deterministic).

It can also be shown that under our assumptions, for every action α the predicate transformers $\mathbf{wp}.\alpha$ and $\mathbf{sp}.\alpha$ are converse in the following sense:

$$[\![p \Rightarrow \mathbf{wp}.\alpha.q]\!] \;\equiv\; [\![\mathbf{sp}.\alpha.p \Rightarrow q]\!]$$

This establishes that $\mathbf{sp}.\alpha$ is universally conjunctive and disjunctive as well. From our notation for actions we see that for action α and state predicate q the predicate transformer $\mathbf{sp}.\alpha$ can be characterized as $[\![\mathbf{sp}.\alpha.q \equiv \alpha.q]\!]$. Combining this characterization with the above condition yields the following property of $\mathbf{wp}.\alpha$:

$$[\![p \Rightarrow \mathbf{wp}.\alpha.(\alpha.p)]\!]$$

We also extend the above predicate transformers to whole programs: for a program F with action set $F.A$ we define the predicate transformers $\mathbf{wp}.F$ and $\mathbf{sp}.F$ as

$$[\![\mathbf{wp}.F.q \;\equiv\; \langle\forall\alpha : \alpha \in F.A : \mathbf{wp}.\alpha.q\rangle]\!]$$
$$[\![\mathbf{sp}.F.q \;\equiv\; \langle\exists\alpha : \alpha \in F.A : \alpha.q\rangle]\!]$$

If the program F is understood from the context, we often use the predicate transformer **wco** (pronounced *weakest constrains* because of its connection to the constrains operator **co** of UNITY logic, cf. section 2.2.2) to denote $\mathbf{wp}.F$. Clearly, $\mathbf{wp}.F$ is universally conjunctive, and $\mathbf{sp}.F$ is universally disjunctive.

2.1.5 Some Results on Extreme Solutions of Equations

In developing our theory for generalized leads-to properties we will encounter several predicate transformers that are defined as extreme solutions of certain equations. Given the implication ordering on predicates we call p the strongest (weakest) solution of the equation E in the unknown Z – written as $Z : E$ – if and only if p solves E and any solution of E follows from (implies) p.

For equations of the form $Z : [\![Z \equiv f.Z]\!]$ for some monotonic predicate transformer f, we denote by $\langle\mu Z :: f.Z\rangle$ the strongest solution of $Z : [\![Z \equiv f.Z]\!]$, and call it the *least fixpoint* of f. Similarly, we denote by $\langle\nu Z :: f.Z\rangle$ the weakest solution of $Z : [\![Z \equiv f.Z]\!]$ and call it the *greatest fixpoint* of f. Existence of such solutions is established by the well known theorem of *Knaster-Tarski* [Tar55]:

Theorem 1 (Knaster-Tarski) *For a monotonic function f, the equation*

$$Z : [\![Z \equiv f.Z]\!]$$

has a weakest and a strongest solution. Furthermore, the strongest solution is the same as the strongest solution of the equation

$$Z : [\![Z \Leftarrow f.Z]\!]$$

and the weakest solution is the same as the weakest solution of the equation

$$Z : [\![Z \Rightarrow f.Z]\!]$$

The following very useful theorem from [DS90] shows that certain junctivity properties are inherited by the extreme solutions of equations from the functions defining them:

Theorem 2 *For monotonic f, denote the strongest solution of $Z : [\![f.(X, Z) \equiv Z]\!]$ by $g.X$ and the weakest solution by $h.X$. Then any type of conjunctivity enjoyed by f is enjoyed by h as well, and any type of disjunctivity enjoyed by f is enjoyed by g as well.*

We also note that the fixpoint operators are monotonic:

Theorem 3 *For monotonic f and g:*

$$[\![f \Rightarrow g]\!] \;\Rightarrow\; [\![\langle \mu Z :: f.Z\rangle \Rightarrow \langle \mu Z :: g.Z\rangle]\!]$$
$$[\![f \Rightarrow g]\!] \;\Rightarrow\; [\![\langle \nu Z :: f.Z\rangle \Rightarrow \langle \nu Z :: g.Z\rangle]\!]$$

Proof . We show the proof for the least fixpoint; the greatest fixpoint is dealt with analogously.

Since f and g are monotonic, their strongest fixpoints exist and satisfy in particular

$$[\![\langle \mu Z :: g.Z\rangle \;\equiv\; g.\langle \mu Z :: g.Z\rangle]\!] \qquad \textbf{(0)}$$
$$\langle \forall X :: [\![X \Leftarrow f.X]\!] \;\Rightarrow [\![\langle \mu Z :: f.Z\rangle \Rightarrow X]\!]\rangle \qquad \textbf{(1)}$$

With this we observe

$$\begin{array}{cl} & [\![\langle \mu Z :: f.Z\rangle \Rightarrow \langle \mu Z :: g.Z\rangle]\!] \\ \Leftarrow & \{(1) \text{ with } X := \langle \mu Z :: g.Z\rangle\} \\ & [\![\langle \mu Z :: g.Z\rangle \Leftarrow f.\langle \mu Z :: g.Z\rangle]\!] \\ \equiv & \{(0)\} \\ & [\![g.\langle \mu Z :: g.Z\rangle \Leftarrow f.\langle \mu Z :: g.Z\rangle]\!] \\ \Leftarrow & \{\text{predicate calculus}\} \\ & [\![g \Leftarrow f]\!] \end{array}$$

End of Proof.

Using predicate transformers we can now formalize the notion of *reachable state space*: for any program F we define the set of reachable states of F as the set characterized by the predicate $\mathbf{si}.F$, called the *strongest invariant*[6] of F. A state is reachable if it can be reached from some initial program state by a finite number of transitions; hence we define

$$[\![\mathbf{si}.F \ \equiv \ \langle \mu Z :: F.I \vee \mathbf{sp}.F.Z\rangle]\!]$$

From the monotonicity of $\mathbf{sp}.F$, it follows that the predicate transformer in the body of the above fixpoint application is monotonic; therefore, the fixpoint exists and is well defined. Using $\mathbf{si}.F$ we can also define the $[\,]$ operator in terms of the $[\![\,]\!]$ operator by postulating for all p in $\mathcal{P}_F$

$$[p] \ \equiv \ [\![\mathbf{si}.F \Rightarrow p]\!].$$

As an immediate consequence of this definition we have for all p in $\mathcal{P}_F$:

$$[\![p]\!] \ \Rightarrow \ [p]$$

which allows us, for instance, to replace an assertion of the form $[\![p]\!]$ by $[p]$.

[6] A justification for this term is given in section 3.2.

2.1.6 Regular Expressions

Part of our theory will make extensive use of a restricted form of regular expressions. We briefly summarize a few basic definitions and some notational conventions related to regular expressions that will be used later on. We start with some definitions concerning strings and languages.

For a given finite alphabet A we denote by A^* the monoid $(A, \cdot)$ of strings over A, where $\cdot$ is the concatenation operator. The neutral element of A^* is denoted by ε and is called the empty string. We often omit $\cdot$ and denote the concatenation by juxtaposition of elements of A^*.

A *language* over A is a subset of A^*. For two strings $s, t \in A^*$ we say that s is *subsumed* by t (written as $s \leq t$) if and only if s is a subsequence of t, i.e., s can be obtained from t by removing zero or more symbols. Formally, for $s, t \in A^*$, and $x \in A$:

$$\begin{array}{lll} \varepsilon \leq t & & \\ (x \cdot s) \leq t & \equiv & \langle \exists u, v : u, v \in A^* : t = uxv \ \wedge \ s \leq v \rangle \end{array}$$

For two languages K, L over A we say that K is subsumed by L if and only if every string of K is subsumed by some element of L:

$$K \leq L \qquad \equiv \quad \langle \forall s : s \in K : \langle \exists t : t \in L : s \leq t \rangle \rangle$$

It is easily seen that subsumption ($\leq$) is an ordering relation on languages over A that is weaker than the language containment ordering ($\subseteq$).

We define *regular expressions* over a finite alphabet A as the free universal algebra [Wec92, Jac80, Con71] with the nullary constructors $\emptyset$, ε and α for each $\alpha \in A$, the unary constructor * (repetition), and the binary constructors $+$ (alternation) and $\cdot$ (sequencing). We write $+$ and $\cdot$ as infix operators and * as a postfix operator. As usual, we often omit the $\cdot$ operator from expressions, writing for instance UV instead of $U \cdot V$. We also associate different orders of precedence with the operators to reduce the number of parentheses required when writing expressions: * binds strongest, followed by $\cdot$, followed by $+$. For instance, $U + VW^*$ denotes the element $U + (V \cdot (W^*))$.

Later, we will be mostly interested in a sub-algebra of this free algebra of regular expressions, which is obtained from it by omitting the $\emptyset$ constructor. For a program F with action set $F.A$ we denote this free algebra of $\emptyset$-free regular expressions over $F.A$ by $\mathcal{R}_F$[7].

Any regular expression over some alphabet A denotes a regular language ([HU79]) over A by virtue of the following mapping $\mathcal{L}$: $\mathcal{L}.\emptyset = \emptyset$, $\mathcal{L}.\varepsilon = \{\varepsilon\}$, $\mathcal{L}.(UV) = \{x, y : x \in \mathcal{L}.U \wedge y \in \mathcal{L}.V : xy\}$, $\mathcal{L}.(U+V) = \mathcal{L}.U \cup \mathcal{L}.V$, and $\mathcal{L}.U^* = \langle \cup i : i \in \mathbf{N} : \mathcal{L}.U^i \rangle$, where $U^0 = \varepsilon$ and for all i in $\mathbf{N}$, $U^{i+1} = UU^i$.

2.2 UNITY

In the following, we give a very brief overview of the UNITY programming notation and temporal logic. A more thorough and detailed introduction can be found in [CM88], while some more recent developments are described in [Mis95b, Mis95a].

2.2.1 Programming Notation

The computational model for UNITY is that of deterministic, total, labeled state transition systems (TDLSTS) described earlier. The TDLSTS model is well suited for describing many common classes of systems (e.g. hardware circuits or protocols), and is familiar to many designers of such systems. In the following we briefly describe the form of a UNITY program, show how the syntactic form corresponds to a TDLSTS, present an example program, and describe how UNITY programs are executed.

[7]Note that at first we are indeed dealing with the terminal algebra over the given constructors. Only later in chapter 4 will we investigate a coarser equational theory for $\mathcal{R}_F$.

Program Sections

A UNITY program consists of four parts: (1) a collection of variable declarations, called the *declare* section, defines the state space of the program; (2) optionally, a set of abbreviations, called the *always* section, defines certain *transparent* variables used to write programs succinctly; (3) a set of initial conditions, called the *initially* section, characterizes the set of initial states of the program, and (4) a finite set of statements, called the *assign* section, defines the actions of the program and, thereby, its transition relation. In accordance with our requirements for actions (cf. section 2.1.4) statements are guarded multiple assignments, deterministic and always terminating.

The TDLSTS corresponding to a program F consists of the set of states of F which is the Cartesian product of the domains of all variables declared in the *declare* section, the set of actions corresponding to the statements of F listed in the *assign* section, the transition relation defined as the union of all the labeled transition relations corresponding to the individual program statements, and the set of initial states characterized by the conditions in the *initially* section.

An Example

As an example of a UNITY program we consider the scheduling problem known as Milner's cycler [Mil89]. A cycler of size N consists of a cyclic arrangement of N processes P_0 through P_{N-1} in which each process receives a signal as input from its one neighbor, sends a signal to its other neighbor, and performs some further observable actions. More precisely, in a ring of N processes, in which process i sends signals to process $(i+1) \bmod n$ for each i with $0 \leq i < n$, process P_i has the state transition diagram shown in figure 2.1.

As can be seen from the diagram, process P_i repeatedly performs an observable a-action, then synchronizes with process $(i+1) \bmod n$ via a communication action c, and performs a b-action and synchronization with process $(i-1) \bmod n$ in

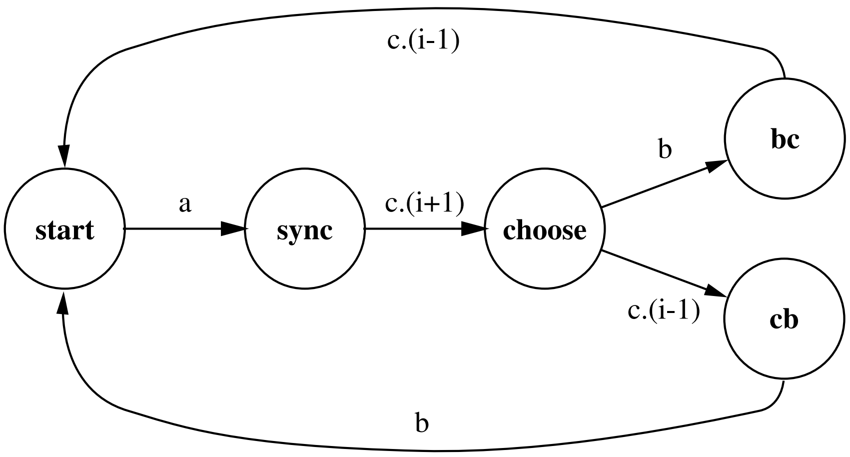


Figure 2.1: Transition Diagram for Process P_i of Milner's Cycler

either order. A property of the cycler is that the a-actions occur in cyclic order over the processes in the ring. Initially, process P_0 is ready to perform an a-action, while all other processes are ready to receive synchronization signals from their respective neighbors.

Figure 2.2 shows an encoding of the cycler for N processes as a UNITY program. The variable $cyc.i$ indicates the state of process P_i (for $0 \leq i < N$) and variable $last\_a$ records the most recently occurred a action.

Program Executions

An execution of a UNITY program is any unconditionally fair execution of the associated TDLSTS. Operationally, an execution is obtained by selecting a start state satisfying the conditions of the *initially* section and then repeatedly selecting statements of the *assign* section and executing them (if the guard of a selected statement evaluates to false in the current state, the entire statement is equivalent to a *skip* operation, i.e., it doesn't change the state). The selection of statements is subject to the unconditional fairness constraint, i.e., every statement is selected infinitely often.

Concerning the example in figure 2.2 it should be noted how the asynchronous

program *Cycler*
  declare
    type $PC = enum(start, sync, choose, bc, cb)$;
    type $Index = int(0..N-1)$;
    var $last\_a : Index$;
    var $cyc : Index \rightarrow PC$;
  initially
    $last\_a = N - 1$;
    $cyc.0 = start$;
    $\langle \forall i : 1 \leq i < N : cyc.i = bc \rangle$;
  assign
    $\langle [] i : 0 \leq i < N :$
$$\begin{array}{lll} & last\_a, cyc.i := i, sync & \textbf{if } cyc.i = start \\ [] & cyc.i := bc & \textbf{if } cyc.i = choose \\ [] & cyc.i := start & \textbf{if } cyc.i = cb \\ [] & cyc.(i-1), cyc.i := choose, start & \textbf{if } cyc.(i-1) = sync \ \wedge \ cyc.i = bc \\ [] & cyc.(i-1), cyc.i := choose, cb & \textbf{if } cyc.(i-1) = sync \ \wedge \ cyc.i = choose \end{array}$$
    $\rangle$
end

Figure 2.2: UNITY program of Milner's Cycler

aspects of the scheduler (i.e., the interleaving execution of a and b actions of different processes) are expressed by separate statements in the UNITY program, whereas the synchronous communication is modeled by multiple assignment statements, updating several state variables simultaneously.

2.2.2 UNITY Logic

The logic of UNITY is a simple temporal logic providing many proof rules for reasoning about programs and their properties. Different from many state-based computational models that reason about individual executions of programs, the UNITY operators characterize properties of programs, i.e., properties of *all* unconditionally fair program executions.

The UNITY programming theory provides many powerful rules for reasoning about program properties. Using these rules one can replace an often tedious and error-prone operational argument by a calculational proof, in which properties of programs are derived by applying inference rules. Moreover, these rules can be used to derive properties of programs, as well as – when applied in *reverse order* – to guide the designer of a program who has to meet certain specifications, by suggesting program refinements corresponding to the structure of the rules [CM88, Kna92].

In the following we introduce the UNITY operators and some rules for reasoning with them to the extent needed for this work. Proofs of most rules are straightforward and can be found in [Mis95b, Mis95a].

Safety

The most fundamental safety property of UNITY logic is the constrains property, or **co** property for short. The **co** operator is a binary relation over state predicates and is defined by the following inference rule:

$$\frac{\langle \forall \alpha : \alpha \in F.A : [\![p \Rightarrow \mathbf{wp}\,.\alpha.q]\!] \rangle,\ [\![p \Rightarrow q]\!]}{p \ \mathbf{co}\ q}$$

The property p **co** q asserts that in any execution a state satisfying p is always followed by a state satisfying q. In order to model stuttering steps p is required to imply q. Several other safety operators are expressed in terms of **co** :

$$\frac{p \ \mathbf{co}\ p}{\mathbf{stable}\ \ p}$$

$$\frac{\langle \forall e :: \mathbf{stable}\ \ x = e \rangle}{\mathbf{constant}\ \ x}$$

$$\frac{\mathbf{stable}\ \ p,\ [\![F.I \Rightarrow p]\!]}{\mathbf{invariant}\ \ p}$$

$$\frac{p \wedge \neg q\ \ \mathbf{co}\ \ p \vee q}{p\ \ \mathbf{unless}\ \ q}$$

A predicate is *stable*, if it remains true once it becomes true. An expression x is *constant* if for any possible value e the predicate $x = e$ is stable. A predicate is *invariant* if it is stable and holds in all initial program states. Finally, the **unless** operator is a binary relation over state predicates, such that p **unless** q holds if in any state satisfying p either p continues to hold forever, or holds up to (but not necessarily including) a state satisfying q.

2.2.3 The Substitution Axiom

The operational semantics of UNITY programs is usually given with respect to the reachable part of the state space of the program. For instance, program F enjoys the property $p \mapsto q$ if and only if for any unconditionally fair execution of F starting in an initial state of F any state satisfying p is also a state satisfying q or is followed by such a state.

On the other hand, the UNITY proof rules do not explicitly refer to the set of reachable states. Instead, the so-called *substitution axiom* can be invoked in order to restrict attention to a superset of the reachable part of the state space as needed. The substitution axiom can be formulated as follows:

$$\frac{\textbf{invariant } p}{[\![p]\!]}$$

Since an invariant of a program is true over the reachable part of the state space, it is equivalent to true in any program property. Thereby the substitution axiom allows us to replace any invariant predicate of a program F by true (and vice versa) in any proof of a property of F.

Progress

The most fundamental progress property is called **transient** and is a unary relation on state predicates. A predicate p is transient in a program F, if there is a program action that falsifies p *in all program states* in which p holds:

$$\frac{\langle \exists \alpha : \alpha \in F.A : [\![p \Rightarrow \mathbf{wp}.\alpha.(\neg p)]\!]\rangle}{\textbf{transient } p}$$

The other basic progress property of UNITY logic is called **ensures** . It is a binary relation on state predicates and is defined in terms of **unless** and **transient** :

$$\frac{p \textbf{ unless } q, \textbf{ transient } p \wedge \neg q}{p \textbf{ ensures } q}$$

If p **ensures** q holds for a program F then there is, by virtue of the **transient** part, an action of F that establishes $\neg p \vee q$ when executed in any state in which $p \wedge \neg q$ holds; together with the **unless** part we have that q is established in any execution starting in a state satisfying p and that p holds up to the point at which q is established.

Progress properties are expressed in UNITY in general with the $\mapsto$ (leads-to) operator. The $\mapsto$ operator is a binary relation on state predicates and is formally defined as the transitive, disjunctive closure of the **ensures** relation, i.e., as the strongest relation satisfying the following three conditions:

$$\frac{p \ \mathbf{ensures}\ q}{p \ \mapsto \ q}$$

$$\frac{p \ \mapsto \ q,\ q \ \mapsto \ r}{p \ \mapsto \ r}$$

$$\frac{\langle \forall w : w \in W : p.w \ \mapsto \ q \rangle}{\langle \exists w : w \in W : p.w \rangle \ \mapsto \ q} \quad \text{for any set } W$$

The UNITY programming theory provides many laws for reasoning about progress properties [CM88]. Among them is the so-called *induction principle* for leads-to properties, which can be stated as follows: for a well-founded set $(W, \prec)$ and a function M mapping program states to W we have

$$\frac{\langle \forall w : w \in W : p \wedge M = w \ \mapsto \ (p \wedge M \prec w) \vee q \rangle}{p \ \mapsto \ q}$$

Since there is only one rule for establishing an **ensures** property of a program F, we can derive from the **ensures** rule and the substitution axiom the following equivalence:

$$p \ \mathbf{ensures}\ q \quad \equiv \quad [p \wedge \neg q \Rightarrow \mathbf{wco}\,.(p \vee q)] \wedge [\langle \exists \alpha : \alpha \in F.A : p \wedge \neg q \Rightarrow \mathbf{wp}\,.\alpha.q \rangle]$$

We will also make use of another property, $\mathbf{ensures}_\alpha$, which is similar to **ensures** but explicitly names the helpful action. For an action α in $F.A$ we define:

$$p \ \mathbf{ensures}_\alpha\ q \quad \equiv \quad [p \wedge \neg q \Rightarrow \mathbf{wco}\,.(p \vee q)] \wedge [p \wedge \neg q \Rightarrow \mathbf{wp}\,.\alpha.q]$$

2.2.4 Program Composition

For dealing with large programs a methodology is needed that makes it possible to decompose large programs into smaller components, to reason about the components individually, and to derive properties of the large program from the properties of its components. The UNITY programming theory is compositional in the sense that it provides many rules for reasoning about multiple programs and their composition.

Although we do not exploit the UNITY compositionality theory in this work, we mention a few results because of their relevance to future work on automated program verification.

UNITY defines two forms of program composition: the symmetric program *union* combines two programs with compatible state spaces and initial conditions by forming the union of their actions. The *union theorem* [CM88] makes it possible to derive safety and basic progress properties of the composed program from its components. In program *superposition*, an underlying program is augmented by superimposed variables, additional actions, and restricted synchronous extensions to existing actions. Due to the restricted way in which the program is augmented, it is possible to state the *superposition theorem*, saying that every property of the underlying program is also a property of the superimposed one.

Another way of dealing with program composition is by defining a notion of *closure properties* [Mis] as a generalization of the ordinary UNITY properties. A program F is said to have a certain closure property, if the corresponding ordinary property is enjoyed by any union of F with a program G that satisfies certain compatibility constraints with respect to F. These compatibility constraints have the form of syntactic link constraints, restricting the access to certain program variables according to their interface characterizations.

Although, in general, progress properties are not compositional, recent research has provided methods for establishing some progress properties of programs from properties of their components [Coh93, CK93, Rao95, Pra95].

2.3 Model Checking

Model checking [CE81, QS82, CES86] has become one of the most successful techniques for verifying and analyzing certain classes of finite-state programs. It has been used to find subtle errors in industrial designs.

2.3.1 Basic Idea of Model Checking

A model checking task consists of determining whether a given system satisfies a specification. Typically, the system is a program, circuit, or process, whereas the specification is a formula of some temporal or modal logic.

When attempting to classify different model checking problems, one can distinguish different approaches by the different kinds of models used (such as automata, process algebras, Kripke structures, or labeled transition systems), and by the temporal logic employed for specifying properties (such as CTL, LTL, CTL* or the modal mu-calculus).

Another way to characterize different model checking approaches is to ask whether they are *global* (i.e., they attempt to determine all the states of a given system satisfying a given specification), or whether they are *local* (i.e., they attempt to determine whether a given specification is satisfied for a given set of states). Although the worst-case complexity of local and global algorithms for many systems and logics is the same in general, the local approach can often be more practical because it often avoids the (explicit or symbolic) construction of the entire state space.

The term model checking is usually reserved for decision procedures, most of them dealing with finite state spaces. However, some model checking algorithms have been extended to unbounded or some restricted infinite state spaces, and some procedures have been developed – typically based on the local paradigm – to deal with infinite state spaces in general [Bra93].

A feature of model checking algorithms that is essential for their use in industrial environments is their ability to generate counterexamples from failed model checking attempts. In large scale applications it is often a serious problem to model programs in the context of insufficiently specified environments. Since this problem cannot be solved by formal verification techniques alone, model checking algorithms tend to be used not so much as *checkers* but more often as *analyzers* and *debuggers*:

instead of having the model checker simply decide whether a given program meets its specification, it is used as a tool for allowing the designer to experiment with different ways of modeling various aspects of the system. In these applications a simple yes/no answer to a model checker invocation is not satisfactory; instead a detailed counterexample exhibiting more information about the nature of an existing problem is highly desirable.

2.3.2 Symbolic Model Checking and OBDDs

Another characterization of model checking algorithms divides such algorithms into *enumerative* and *symbolic* methods. Enumerative methods were the first ones introduced. They attempt to explicitly build (possibly in a lazy fashion) the state space of the system under investigation as part of the checking procedure. As such they are limited by the size of the (part of the) state space that needs to be examined; in particular, they are generally restricted to finite state systems. For some applications, however, enumerative methods have been used with success, especially when combined with reductions of symmetric state spaces ([ID93]).

Symbolic methods, on the other hand, utilize some form of symbolic representation for the models under consideration (such as set of states and transition relations), and are thus capable of dealing with potentially much larger systems, provided a suitably compact symbolic representation can be found.

A practical breakthrough in the application of model checking techniques to large programs was accomplished through the introduction of *ordered binary decision diagrams* (OBDDs) [Bry86] and their incorporation into model checking algorithms [McM92, Pix90, CBM89]. The idea behind OBDDs is to encode sets of states and transition relations that are relevant for the calculations of a symbolic model checking algorithm by boolean functions over the state space of the program. Using OBDDs it is possible in many applications to represent these functions in a concise way and to manipulate them efficiently. Much of the practical success of model checking can be attributed to the clever use of this symbolic representation in model checking algorithms for logics such as CTL.

Recently, model checking algorithms have become even more powerful by exploiting symmetry, by using compositionality of the underlying models, and by utilizing abstraction mappings in order to reduce the size of the state space that needs to be examined. As a result some systems with more than 10^{1000} states can be handled today [CGL94]. Some of the inherent limitations of OBBDs have also been successfully tackled for many practical applications. In particular, the effects of the strong dependence of OBDDs sizes on variable orderings and the difficulty of finding good orderings reliably, have been partially overcome by domain-specific ordering heuristics [Gro94] and by dynamic reordering algorithms [Rud93]. Moreover, some programs that do not have an efficient encoding using OBDDs (e.g. multiplier circuits) have been successfully verified using extensions of OBBDs. By encoding word-level operations (instead of bit-level operations as done with OBDDs), data structures such as *binary moment diagrams* [BC95] made it possible to solve previously intractable problems [CZ96].

Altogether, model checking has become a very powerful verification technique, that can be easily used in practice (at least compared with approaches based on theorem proving), due to its mostly automated nature. However, circuits and protocols built today are still far too big and complex to be handled in general. It is expected that techniques capable of dealing with such large systems will have to take advantage of modular and compositional verification and of user supplied design knowledge.

Chapter 3

Model Checking for UNITY

The programming theory UNITY combines a simple yet expressive temporal logic with a programming notation that is suitable for the formal specification, design, and analysis of concurrent programs. Since its introduction in [CM88] the UNITY theory has been thoroughly investigated [M$^+$90] and simplified [Mis95b], it has been applied to a variety of interesting problems [Sta92], and been used as a foundation for other theories and formalisms [Car94, CWB94, CR90].

UNITY is Suitable for Program Design and Verification

There are several reasons why UNITY is an interesting formalism for designing, reasoning about, and verifying concurrent systems. First, UNITY logic is simple and, therefore, more likely to be effectively used by designers of concurrent systems than more expressive though more complicated formalisms. In spite of its simplicity UNITY logic is expressive enough to state many important and desired properties of concurrent systems, such as invariants and progress properties. Secondly, the programming model makes it possible to abstract away from program control flow, one of the greatest hindrances in understanding concurrent systems. It is also capable of describing synchronous and asynchronous aspects of concurrent systems. Moreover, UNITY logic has a well developed deductive system that allows formal reasoning about concurrent programs, thus providing several results about program composition. This is, undoubtedly, one of the most important requirements of any formalism intended to deal with large and complex systems. Finally, UNITY also comprises a programming methodology that can aid the designer in building correct concurrent systems.

Combining UNITY with Model Checking

Because of its suitability for designing and verifying concurrent programs, it is a promising idea to combine UNITY with one of the most successful techniques for program verification, namely model checking. The goals of this project, model checking UNITY, are threefold: first, such a combination can result in a valuable tool for designers and users of UNITY, aiding them in developing and analyzing their programs by automating certain verification tasks which are often tedious and error prone to perform manually; second and more fundamentally, we aim at supporting the claim that a simple logic like UNITY is well suited for practical use in designing concurrent systems, both because of its simplicity that allows the user to conveniently reason within the logic, and because of its restrictions, that make an efficient implementation of verification procedures for that logic possible. Finally, a system implementing model checking for UNITY is expected to serve as a platform for exploring advanced ideas for verifying and reasoning about concurrent systems in future research. In the remaining chapters of this thesis we demonstrate how each of these goals has been achieved in our work.

In the remainder of this chapter we investigate how the existing deductive system of UNITY logic can be exploited to derive efficient model checking algorithms: in section 3.1 we derive verification conditions for UNITY properties from the UNITY proof rules. In section 3.2 we discuss the important role invariants play in verifying properties with the UNITY verification conditions. We present the UNITY model checking method in section 3.3 where we also discuss its advantages and limitations. An example in section 3.4 illustrates several aspects of the proposed method. We conclude this chapter with some remarks in section 3.5.

3.1 Verification Conditions for UNITY

The most interesting challenge in designing a model checking algorithm for UNITY logic is to determine how one can take advantage of the simplicity of the logic in order to improve its efficiency compared to model checking algorithms for more general logics such as CTL [CES86].

The starting point for a derivation of a model checking method that takes advantage of the deductive structure of UNITY is the UNITY proof system, presented in section 2.2. As suggested by the substitution axiom, invariants play an important role in proving properties of programs. In the following we make invariants explicit by tagging properties with the invariant used in a proof:

Definition 1 (Tagged Properties) *For a program F, a property π and an invariant J of F, $(\pi)^J$ denotes the property obtained by replacing each predicate p in π by $p \wedge J$.*

Tagging a property with an invariant J effectively restricts the property to the part of the state space characterized by J. Since the semantics of traditional untagged properties of a program F is given with respect to the reachable set of states characterized by the strongest invariant of F, $\mathbf{si}.F$ of F, it follows that a property π is satisfied by F in the traditional sense, if and only if $(\pi)^{\mathbf{si}.F}$ is satisfied. The idea of tagging properties with a restricting invariant is similar to the treatment of invariants in [San91].

We call a property of a program F *directly provable* if it can be proved in the UNITY proof system for F without using the substitution axiom. The importance of direct provability is emphasized by the theorem about *normal forms of proofs* due to Misra [Mis90b]:

Theorem 4 *For any property π of a program F there is an invariant J such that $(\pi)^J$ is directly provable.*

The significance of this theorem for our derivation of a verification method is that any proof of a property π can be split into two parts, namely first finding a suitable invariant J and then proving the tagged property π^J directly. In the following we determine verification conditions for characterizing direct provability. To this end, we restate the UNITY proof rules of sections 2.2.2 and 2.2.3 in terms of tagged properties for a given program F:

$$\frac{\textbf{invariant } J, [\![p \wedge J \Rightarrow q]\!], \langle \forall \alpha : \alpha \in F.A : [\![p \wedge J \Rightarrow \textbf{wp}.\alpha.(q \wedge J)]\!] \rangle}{(p \textbf{ co } q)^J} \qquad \textbf{(co)}$$

$$\frac{(p \textbf{ co } p)^J}{(\textbf{stable } p)^J} \qquad \textbf{(stable)}$$

$$\frac{\langle \forall e :: (\textbf{stable } x = e)^J \rangle}{(\textbf{constant } x)^J} \qquad \textbf{(constant)}$$

$$\frac{(\textbf{stable } p)^J, \ [\![F.I \Rightarrow p]\!]}{\textbf{invariant } p} \qquad \textbf{(invariant)}$$

$$\frac{(p \wedge \neg q \textbf{ co } p \vee q)^J}{(p \textbf{ unless } q)^J} \qquad \textbf{(unless)}$$

$$\frac{\textbf{invariant } J, \ \langle \exists \alpha : \alpha \in F.A : [\![p \wedge J \Rightarrow \textbf{wp}.\alpha.(\neg p \wedge J)]\!] \rangle}{(\textbf{transient } p)^J} \qquad \textbf{(transient)}$$

$$\frac{(p \textbf{ unless } q)^J, \ (\textbf{transient } p \wedge \neg q)^J}{(p \textbf{ ensures } q)^J} \qquad \textbf{(ensures)}$$

$$\frac{(p \textbf{ ensures } q)^J}{(p \mapsto q)^J} \qquad \textbf{(promote)}$$

$$\frac{(p \mapsto q)^J, \ (q \mapsto r)^J}{(p \mapsto r)^J} \qquad \textbf{(trans)}$$

$$\frac{\langle \forall w : w \in W : (p.w \mapsto q)^J \rangle}{(\langle \exists w : w \in W : p.w \rangle \mapsto q)^J} \quad \text{for any set } W \qquad \textbf{(disj)}$$

$$\textbf{invariant}\ \text{true} \qquad \textbf{(true)}$$

$$\frac{(\pi)^J}{\pi} \text{ for any property } \pi \qquad \textbf{(lift)}$$

There are two additions to the usual UNITY proof rules dealing specifically with the tagging of properties: **(true)** serves as a base case for establishing invariants and the lifting rule **(lift)** relates tagged properties to ordinary untagged ones. It should be noted that in rules **(co)** and **(invariant)** some conjunctions of predicates with J have been omitted, because they are equivalent to the predicates alone: in **(co)** the tagged condition $[\![p \wedge J \Rightarrow q \wedge J]\!]$ is propositionally equivalent to $[\![p \wedge J \Rightarrow q]\!]$, and in **(invariant)** the tagged condition $[\![F.I \Rightarrow p \wedge J]\!]$ is equivalent to $[\![F.I \Rightarrow p]\!]$ because J is an invariant and therefore satisfies $[\![F.I \Rightarrow J]\!]$.

By virtue of tagging the properties with invariants we are assured that the rules **(co)**, **(stable)**, **(constant)**, **(unless)**, **(transient)**, and **(ensures)** are equivalences; the same holds for **(invariant)** as well if the occurrence of J in the antecedent is existentially quantified. Only the rules for leads-to – **(promote)**, **(trans)**, and **(disj)** – are proper implications in general.

As a consequence of these equivalences, the proof rules for **co** , **stable** , **unless** , **invariant** , **transient** , and **ensures** properties can be transformed into formulae that refer only to the given program text (via **wp**) and to some invariant needed for its proof. This form can be obtained by repeated substitution of equivalences. For instance, we obtain for the **stable** operator:

$$\begin{array}{ll} & (\textbf{stable}\ \ p)^J \\ \equiv & \{\textbf{(stable)}\} \\ & (p\ \ \textbf{co}\ \ p)^J \\ \equiv & \{\textbf{(co)}\} \\ & (\textbf{invariant}\ \ J)\ \wedge\ \langle \forall \alpha : \alpha \in F.A : [\![p \wedge J\ \Rightarrow\ \textbf{wp}\,.\alpha.(p \wedge J)]\!]\rangle \end{array}$$

This derivation makes it clear that **stable** p is provable for F if and only if there is some invariant J of F satisfying the last line of the derivation. Together with the soundness and completeness of the UNITY logic [Kna92] we obtain a verification condition for F and the property **stable** p . Similar derivations for the other kinds of properties result in the following list of verification conditions:

$F \models p$ **co** q iff for some invariant J of F:

$$[\![J \wedge p \Rightarrow q]\!] \;\wedge\; \langle \forall \alpha : \alpha \in F.A : [\![J \wedge p \;\Rightarrow\; \mathbf{wp}.\alpha.(J \wedge q)]\!]\rangle$$

$F \models$ **stable** p iff for some invariant J of F:

$$\langle \forall \alpha : \alpha \in F.A : [\![J \wedge p \;\Rightarrow\; \mathbf{wp}.\alpha.(J \wedge p)]\!]\rangle$$

$F \models$ **constant** x iff for some invariant J of F:

$$\langle \forall e :: \langle \forall \alpha : \alpha \in F.A : [\![J \wedge x = e \;\Rightarrow\; \mathbf{wp}.\alpha.(J \wedge (x = e))]\!]\rangle\rangle$$

$F \models$ **invariant** p iff for some invariant J of F:

$$[\![F.I \Rightarrow p]\!] \;\wedge\; \langle \forall \alpha : \alpha \in F.A : [\![J \wedge p \;\Rightarrow\; \mathbf{wp}.\alpha.(J \wedge p)]\!]\rangle$$

$F \models p$ **unless** q iff for some invariant J of F:

$$\langle \forall \alpha : \alpha \in F.A : [\![J \wedge p \wedge \neg q \;\Rightarrow\; \mathbf{wp}.\alpha.(J \wedge (p \vee q))]\!]\rangle$$

$F \models$ **transient** p iff for some invariant J of F:

$$\langle \exists \alpha : \alpha \in F.A : [\![J \wedge p \;\Rightarrow\; \mathbf{wp}.\alpha.(J \wedge \neg p)]\!]\rangle$$

$F \models p$ **ensures** q iff for some invariant J of F:

$$\langle \forall \alpha : \alpha \in F.A : [\![J \wedge p \wedge \neg q \;\Rightarrow\; \mathbf{wp}.\alpha.(J \wedge (p \vee q))]\!]\rangle \wedge$$
$$\langle \exists \alpha : \alpha \in F.A : [\![J \wedge p \wedge \neg q \;\Rightarrow\; \mathbf{wp}.\alpha.(J \wedge (\neg p \vee q))]\!]\rangle$$

The important feature of the above verification conditions is that they are *local*, i.e., that they only refer to individual transitions (via **wp**) and do not rely on any fixpoint computations. We use these verification conditions as the basis of a model checking method for UNITY. Before we present the method in section 3.3 we need to discuss the role of invariants for these verification conditions in greater detail.

3.2 The Role of Invariants

It is obvious that finding a suitable invariant is essential for taking advantage of the previously mentioned locality of the verification conditions. An invariant suitable for proving a property π of a program F is any invariant of F that is strong enough to make the verification condition for π true. We formalize this characterization with the following lemma:

Lemma 5 *For any program F, the set of invariants of F is a complete lattice with boolean implication as ordering relation,* true *as top, and* $\mathbf{si}.F$ *as bottom element.*

Proof . It suffices to show that $\mathbf{si}.F$ is an invariant of F, and that for any invariant J of F and any predicate K with $[\![J \Rightarrow K]\!]$, K is also an invariant of F. By virtue of the fixpoint definition of $\mathbf{si}.P$ we have

$$[\![\mathbf{si}.F \;\equiv\; F.I \vee \mathbf{sp}.F.(\mathbf{si}.F)]\!] \qquad \textbf{(SI)}$$

from which $[\![F.I \Rightarrow \mathbf{si}.F]\!]$ follows immediately. For the stability we observe with $J = \text{true}$ for all α in $F.A$:

$$\begin{array}{ll}
 & [\![\text{true} \wedge \mathbf{si}.F \Rightarrow \mathbf{wp}.\alpha.(\mathbf{si}.F)]\!] \\
\Leftarrow & \{\text{property of } \mathbf{wp}.\alpha\text{: } [\![Y \Rightarrow \mathbf{wp}.\alpha.(\alpha.Y)]\!]\} \\
 & [\![\mathbf{wp}.\alpha.(\alpha.(\mathbf{si}.F)) \Rightarrow \mathbf{wp}.\alpha.(\mathbf{si}.F)]\!] \\
\Leftarrow & \{\mathbf{wp}.\alpha \text{ is monotonic}\} \\
 & [\![\alpha.(\mathbf{si}.F) \Rightarrow \mathbf{si}.F]\!] \\
\Leftarrow & \{\text{definition of } \mathbf{sp}.F\} \\
 & [\![\mathbf{sp}.F.(\mathbf{si}.F) \Rightarrow \mathbf{si}.F]\!]
\end{array}$$

which follows from **(SI)**. Together with the verification condition for invariants we, therefore, have **invariant** $\mathbf{si}.F$.

For any invariant J of F, and any predicate K for which $[\![J \Rightarrow K]\!]$ holds, we have, by virtue of the verification condition for invariants and the transitivity of implication, that $[\![F.I \Rightarrow K]\!]$; also, since $[\![J \wedge K \equiv J]\!]$ holds, we have, by virtue of the verification condition for invariants and the monotonicity of $\mathbf{wp}.\alpha$, that for all α in $F.A$ the stability condition $[\![J \wedge K \Rightarrow \mathbf{wp}.\alpha.K]\!]$ is satisfied. Hence K is an invariant.

End of Proof.

It follows that a property π is satisfied by a program F if and only if the verification condition for π with respect to $\mathbf{si}.F$ is true. On the other hand, any invariant of F for which the verification condition of π becomes true suffices to establish π. Therefore, it is not required to find the strongest invariant, but any sufficiently strong invariant. In the following we describe different ways for finding such invariants. A presentation of different techniques in the context of linear temporal logic can be found in [MP95].

3.2.1 The Strongest Invariant

As mentioned before, the strongest invariant of F, $\mathbf{si}.F$, is sufficient for proving any property of F. The main problem with $\mathbf{si}.F$ is, however, that it may not be possible to be computed for a given program F. For infinite state spaces and recursively enumerable (r.e.) sets of initial states, the set of reachable states is r.e. but not necessarily decidable as can be seen by reduction from the Halting Problem (for the definitions see for instance [HU79]): a deterministic universal Turing machine can be modeled as a TDLSTS, hence it cannot be decided whether a final state is reachable. But even for finite state programs F, where the computation of $\mathbf{si}.F$ suggested by the fixpoint definition of $\mathbf{si}.F$ is guaranteed to terminate, it might not be feasible to actually compute the strongest invariant due to limited resources (memory and time), even when using a symbolic representation.

3.2.2 Automatically Generated Invariants

As we have seen, it may not always be possible to compute the strongest invariant for large and complex programs. In such a case we have to find ways of computing sufficiently strong invariants with which we are then able to prove the properties at hand. The first guidance for this task can be found in the program for which we want to verify properties: we can take advantage of type declarations and possibly other *syntactic* restrictions in order to derive certain invariants automatically. A *type invariant* for some variable x simply asserts that x takes on values only from its declared domain. Other invariants can be derived for variables on which only a restricted set of operations is performed, e.g. integer variables which are incremented or decremented only by some fixed constant. More elaborate techniques analyze certain dependencies of sets of variables; e.g. the *linear invariants* of [M$^+$94, MP95] are obtained by determining linear dependencies between certain program variables.

3.2.3 User Supplied Invariants

In most of the interesting cases automatically generated invariants do not suffice to prove the desired properties of a given program. The reason for this is that typing and other syntactic features of a program description typically do not capture the full semantic content of a program. A certain amount of design knowledge has gone into the construction of such a program and cannot be extracted easily from the program description alone. In a situation in which the program verification is performed hand-in-hand with the program design, however, it is possible to transfer some of this design knowledge to the verification task.

A particular form of such design knowledge is state-based and has the form of invariants: the designer of a program often has an understanding about restrictions on possible values of certain variables that are present as part of the program design, but are very difficult to calculate without that knowledge.

It is therefore useful for the designer to supply such *design invariants* to the verification procedure. The verification procedure has to establish first that the provided invariants indeed are invariants of the program, and then to strengthen the automatically generated invariants by the newly provided ones. It is often the case that some of these design invariants suffice to prove the desired properties of a given program. Heuristic evidence of this fact is presented in chapter 7.

It should be noted that providing design invariants amounts to supplying a *partial proof* of a property. In fact, if a deductive proof of some property uses the substitution axiom on a series of invariants, it is clear that the conjunction of all these invariants serves as a sufficiently strong invariant for the verification condition. In chapter 5 we illustrate the relationship between proofs and design knowledge in more detail.

3.2.4 Strengthening Invariants

Sometimes the above methods still do not suffice: the automatically generated invariants are too weak, design invariants are not available or are not strong enough, and the strongest invariant cannot be computed; another problem might be that a suggested design invariant cannot be established since it does not meet the stability requirement with respect to the available invariants. In such cases an attempt can be made to use a *goal-oriented* technique that takes the properties to be proved into account.

While all the suggested techniques for finding invariants were independent of properties to be established, the idea behind *invariant strengthening* is to utilize the information obtained from a failed property verification to strengthen the used invariant; this is done under the assumption that the given property, in fact, holds.

The procedure is as follows: a failed verification condition for a property π results in a predicate characterizing certain illegal states. For instance, a failed check for the property **stable** p of program F with respect to the established invariant J characterizes the set of states violating the stability condition as $\langle \exists \alpha : \alpha \in F.A : J \wedge p \wedge \neg\, \mathbf{wp}\,.\alpha.(J \wedge p) \rangle$. Under the assumption that **stable** p holds, we can strengthen the invariant from J to $J \wedge \langle \forall \alpha : \alpha \in F.A : J \wedge p \Rightarrow \mathbf{wp}\,.\alpha.(J \wedge p) \rangle$. This, of course, requires us to establish that the strengthened invariant is indeed an invariant of

F. We can repeat this strengthening procedure with respect to other properties or with respect to the alleged invariant until we succeed in establishing the invariant and, thereby, confirm the validity of all properties used for strengthening, or until we arrive at an alleged invariant that does not cover all reachable states (a simple check is to test whether the set of initial states is covered). In the latter case we have established indirectly that some property used for strengthening is not satisfied by the program.

It should be noted that for infinite state programs the repeated strengthening of alleged invariants need not terminate by reaching a stable predicate or by excluding some reachable states. However, for finite state programs this method is guaranteed to strengthen invariants successfully. Of course, even in the finite case the method might not be practicable if the representation of the strengthened invariants grows too big.

3.3 A Model Checking Procedure for UNITY

After discussion of the two main ingredients of a verification procedure for UNITY, namely the verification conditions based on the notion of direct provability, and methods for finding sufficiently strong invariants, we describe below a model checking procedure for finite state programs and propositional UNITY properties, and discuss its advantages and limitations.

3.3.1 Description of the Procedure

A model checking procedure for UNITY logic can be constructed from three ingredients: the verification conditions for the various properties, a method for finding suitable invariants, and a representation for state predicates that is both concise and allows efficient calculation of the verification conditions.

Before we can present such a method for full UNITY logic we need to address the fact that we do not have a local verification condition for leads-to properties.

Even though for finite state programs the disjunctivity rule **(disj)** is subsumed by the transitivity rule **(trans)** [Mis95a], it is still not possible to define an equivalence that directly relates leads-to properties to state or transition predicates[1].

Dealing with leads-to

We can, however, express an equivalence involving leads-to by using the predicate transformer **wlt** (*weakest leads-to*) from [JKR89]: for a state predicate q, the predicate $\mathbf{wlt}\,.q$ characterizes all states with the property that any unconditionally fair execution starting from such a state eventually reaches a state in which q holds. Formally, $\mathbf{wlt}\,.q$ can be defined as

$$\llbracket \mathbf{wlt}\,.q \;\equiv\; \langle \mu Z :: q \vee \mathbf{we}\,.Z \rangle \rrbracket$$
$$\llbracket \mathbf{we}\,.q \;\equiv\; \langle \exists \alpha : \alpha \in F.A : \mathbf{stp}\,.\alpha.q \rangle \rrbracket$$
$$\llbracket \mathbf{stp}\,.\alpha.q \;\equiv\; \langle \nu Z :: (\mathbf{wco}\,.Z \wedge \mathbf{wp}\,.\alpha.q) \vee q \rangle \rrbracket$$

and satisfies the following characterization:

$$[p \Rightarrow \mathbf{wlt}\,.q] \quad \equiv \quad p \;\mapsto\; q.$$

From this we can derive the following verification condition for leads-to properties:

$$F \;\models\; p \;\mapsto\; q \qquad \text{iff for some invariant } J \text{ of } F\text{:}$$
$$\llbracket J \wedge p \;\Rightarrow\; \mathbf{wlt}\,.(J \wedge q) \rrbracket.$$

With the given verification conditions for UNITY properties, we present a (symbolic) model checking procedure. The procedure takes as input a UNITY program F and a set P of UNITY properties of F. The model checking procedure can be executed in one of two *modes*, *automated* or *interactive*. The automated mode is available if the strongest invariant $\mathbf{si}\,.F$ can be computed.

[1] A *transition predicate* of program F is a state predicate over $F.S \times F.S$, thereby characterizing a successor relation.

Automated Mode

If $\mathbf{si}.F$ has been computed, the verification conditions for all input properties can be evaluated with respect to $\mathbf{si}.F$. A verification condition evaluating to true indicates that the property is satisfied by F, whereas false means that the property is not met by F. It can be expected that all verification conditions with the possible exception of the one for leads-to properties can be handled efficiently in the automated case, since they are typically much simpler than the computation of the strongest invariant.

Interactive Mode

In case it is not possible to compute $\mathbf{si}.F$ the procedure is executed in interactive mode. As long as there are unproven properties the user can select any such property together with an established invariant, evaluate the verification condition, and – depending on the result – be either done with the property by having successfully established it, or be provided with some debugging information in case of failure.

Throughout the interactive execution, properties are paired with sets of invariant predicates, indicating which invariants have been used in checking the property. Two sets of tagged properties are maintained by the procedure: *SUCCESS* contains all the properties that have been proved for F, while *TODO* contains the properties that have been checked but could not be proved yet. Furthermore, the variable *INV* contains the strongest invariant established for F during the verification session. *INV* is initialized to the invariant automatically generated from the program text (or to true if no such procedure is available).

Choosing a Property When choosing a property π and an invariant J for the verification condition evaluation, two conditions have to be met: J has to be implied by the current value of *INV*, and J has to be stronger than all invariant predicates π is tagged with. The first condition guarantees that J is an established invariant of

F, the second makes sure that the subsequent verification condition evaluation has some chance of being successful. Unless the size of the representation of the current value of *INV* is large, one should choose the current value of *INV* for J and select a property that has not been checked with respect to the current value of *INV*. If no such property exists, the user has to strengthen *INV* either by supplying a stronger invariant property and successfully checking it, or by attempting an automated invariant strengthening with respect to one of the properties in *TODO*.

Success of Checking Condition If the verification condition for a property π and invariant J evaluates to true, the property is placed into the *SUCCESS* set; it has been proved to be satisfied by F. If π is an invariant property, its predicate is also conjoined to *INV*.

Failure of Checking Condition If the verification condition does not evaluate to true, it could mean that J was not strong enough to establish π. In particular, π is definitely not a property of F if the check with respect to the strongest invariant, $\mathbf{si}.F$, fails. The result of the computation of the verification condition characterizes a set of states that violate the verification condition. For instance, the negation of the verification condition for **stable** properties characterizes the states for which the execution of some action violates the stability requirement. Based on this information the user can decide either that there is an error in the property or the program – in which case the property is removed, or the program is modified – or that the used invariant needs to be strengthened. This strengthening can be done either manually by submitting a new invariant property to the model checking procedure, or by attempting an automatic invariant strengthening with respect to π.

Automatic Strengthening Automatic strengthening proceeds by conjoining to *INV* the negation of the predicate characterizing the violating states. If the resulting predicate is not implied by the initial condition of F, the property used for automatic strengthening is established as not satisfied by F. Otherwise a new invariant property with the resulting predicate is added as input to the model checking procedure, starting another verification round.

Simple Optimizations

We point out four optimizations of the described procedure. First, when checking an invariant property π with respect to some invariant J, where J is stronger than the invariant predicate of π, the result of the check can be asserted to be true.

Second, when checking the condition for a leads-to property, an early termination of a successful check is possible due to the monotonicity of the **wlt** fixpoint iterations.

Third, since the computation of **wlt** does not depend on the invariant with respect to which a leads-to property is checked, the **wlt** result of an unsuccessful check can be cached and reused for a later check of the same property with respect to some stronger invariant.

Finally, the conjunction with J can be dropped from the arguments of **wp** and **wlt** in all verification conditions provided J is *inductive*, i.e., satisfies the condition $[\![J \Rightarrow \mathbf{wco}\,.J]\!]$. It can be shown that for any inductive invariant J and predicates p and q the following equivalences hold:

$$[\![(J \wedge p \Rightarrow \mathbf{wp}\,.\alpha.(J \wedge q)) \;\equiv\; (J \wedge p \Rightarrow \mathbf{wp}\,.\alpha.q)]\!]$$
$$[\![(J \wedge p \Rightarrow \mathbf{wlt}\,.(J \wedge q)) \;\equiv\; (J \wedge p \Rightarrow \mathbf{wlt}\,.q)]\!]$$

The suggested simplification of the verification conditions is an application of these equivalences. It is straightforward to show that the inductive invariants of a program F form a complete sub-lattice of the lattice of invariants of F with true as top element and $\mathbf{si}.F$ as bottom element. Furthermore, if predicate p has been shown to be an invariant by using the verification condition with respect to some invariant

J, then predicate $J \wedge p$ is an inductive invariant. Also, the type invariant of a program is inductive. It follows that if only the type invariant of F, the strongest invariant of F, or the conjunction of all established invariants of F are used for checking other properties, the simplified verification conditions can be used.

3.3.2 Properties of the Verification Conditions

The verification conditions for safety and basic progress properties have two characteristics that make them well suited for efficient model checking: they are *simple* formulae of the underlying (non-temporal) logic, i.e., they do not contain any fixpoint operations; furthermore, they are naturally partitioned by the actions of F.

The simple form of the verification conditions is a consequence of the fact that every action α in $F.A$ corresponds to a deterministic conditional multiple assignment, for which the weakest precondition $\mathbf{wp}\,.s$ can be computed easily. More precisely, a deterministic conditional multiple assignment in the assign section of a UNITY program has the form

$$\vec{x} \;:=\; \langle [\!]\, i :: \vec{f}.i.(\vec{x},\vec{y}) \quad \text{if} \quad b.i.(\vec{x},\vec{y}) \rangle$$

where $\vec{x}$ and $\vec{y}$ are tuples of state variables, the $\vec{f}.i$ and $b.i$ are functions expressible in the underlying logic, i ranges over some finite set, and the choices satisfy the condition that whenever any two of them are enabled in a state (i.e., their guards evaluate to true in that state), then their right hand sides evaluate to the same tuples (this condition ensures determinism):

$$\langle \forall i, j :: [\![b.i.(\vec{x},\vec{y}) \wedge b.j.(\vec{x},\vec{y}) \;\Rightarrow\; \vec{f}.i.(\vec{x},\vec{y}) = \vec{f}.j.(\vec{x},\vec{y})]\!] \rangle$$

Such conditional multiple assignment statements always terminate, and the weakest precondition $\mathbf{wp}\,.\alpha.q$ of a state predicate q with respect to a statement α can be computed as

$$[\![\mathbf{wp}\,.\alpha.q \;\equiv\; \langle \forall i :: b.i.(\vec{x},\vec{y}) \Rightarrow (\sigma.i).q \rangle \;\wedge\; (\langle \forall i :: \neg b.i.(\vec{x},\vec{y}) \rangle \Rightarrow q)]\!]$$

where $\sigma.i = \{\vec{x} \leftarrow \vec{f}.i.(\vec{x}, \vec{y})\}$ is the substitution of the components of $\vec{f}.i$ for the corresponding variables of $\vec{x}$.

Due to the expressiveness requirements of the logic we obtain a form of the checking conditions entirely within the underlying logic without having to rely on fixpoint characterizations. We, thereby, eliminate fixpoint computations which may require a number of iterations equal to the diameter of the state graph, by a single formula evaluation whose complexity is comparable to a single step of the iteration.

The second characteristic that makes checking of UNITY conditions favorable for an actual implementation of a model checking algorithm is the partitioning of the global state transition relation by individual statements: the transition relation of program F is the disjunction of the transition relations corresponding to the individual statements of F. Instead of computing the relational product for the full transition relation, we are able to use the representation of the transition relation by a set of statements. It is actually not necessary to compute a pre-image of some set of states under the global transition relation. Instead, the form of the checking conditions makes it possible to compute independent pre-images (corresponding to **wp** computations) for each disjunct of the transition relation corresponding to each statement of the program. Not only can we avoid building the global state transition relation, but the form of the checking conditions allows us to work exclusively with the disjuncts of the global relation. This form of partitioning is long known to result in a significant increase of the applicability of BDD-based symbolic computations [BCM91], and can be directly derived from a given UNITY program at no extra cost.

Although the locality of the verification conditions is responsible for the improved efficiency of verifying properties, it reduces the availability of debugging information that can be directly obtained from a failed verification attempt: the failed verification condition typically contains only local information about some local violation of the required condition. By itself this information is not sufficient to generate complete counterexample traces. For instance, a failed check of the verification condition for a **stable** property characterizes a set of states and an action, so that execution of the action in any one of the states violates the stability requirement. This violating transition represents valuable information that can be used for debugging. A complete execution trace of the program starting in an initial state, however, cannot be derived from this result. In fact, there might be no such trace, because the stability may be verifiable with respect to a stronger invariant. For generating full traces non-local methods have to be used (cf. [McM92]).

3.3.3 Limitations

The proposed method for model checking UNITY programs has the potential to improve the efficiency of verification tasks significantly. However, there are also a few considerations that limit the applicability of the method: in particular, the non-locality of the verification condition for leads-to properties, the reliance on state-based design knowledge, and the dependence on asynchronous programs.

The verification condition for leads-to properties is non-local, since it involves a fixpoint computation of alternation depth two (due to the fairness constraint of UNITY) instead of just a simple evaluation involving pairs of states. Moreover, design invariants are only exploited for detecting a possible early termination of the outer fixpoint iteration. As a result, checking a general leads-to property amounts to a rather complicated fixpoint computation that is typically even more resource-consuming than the reachability computation for the strongest invariant. In that sense our method does not improve the situation for checking general progress properties. We will address this problem at length in chapter 4, where we extend the logic by a generalization of progress properties that often can be checked more efficiently.

The fact that the elimination of fixpoint computations relies on the availability of suitable invariants is a clear disadvantage if the model checking method is used in *a posteriori* verification. When used as a tool during the design process, however, it is likely that some design knowledge is available in a form that can be exploited by our method.

A more fundamental restriction to our approach is its dependence on asynchronous programs. Although UNITY can express synchronous composition, programs that are mostly synchronous tend to consist of only a few very big statements, thereby increasing the complexity of local **wp** calculations while reducing the degree of partitioning at the same time. The resulting size of the formulae that need to be manipulated makes it mandatory to use very efficient symbolic representations.

3.4 An Example

We illustrate some aspects of the UNITY model checking procedure with a small example, an instance of Milner's cycler for two processes. A description of the program and its formulation in the UNITY programming notation has been presented in section 2.2.1. Figure 3.1 shows the transition diagram for the two-process cycler for the state space determined by the variables *cyc.0* and *cyc.1* (i.e., we do not consider the auxiliary variable *last_a*). The initial state is marked with a bold outline; also, self loops, which exist for all states, have been omitted to improve the readability of the diagram.

For this program we demonstrate the verification of two properties, one safety and one progress property. With the safety property we illustrate the locality of the checking conditions, whereas with the progress property we show how the strengthening of invariants is used in the model checking procedure.

We start with a property asserting that process P_0 leaves the *sync* state only through the *choose* state:

$$cyc.0 = sync \quad \mathbf{co} \quad cyc.0 = sync \vee cyc.0 = choose$$

The verification condition for this property with respect to the trivial invariant true is

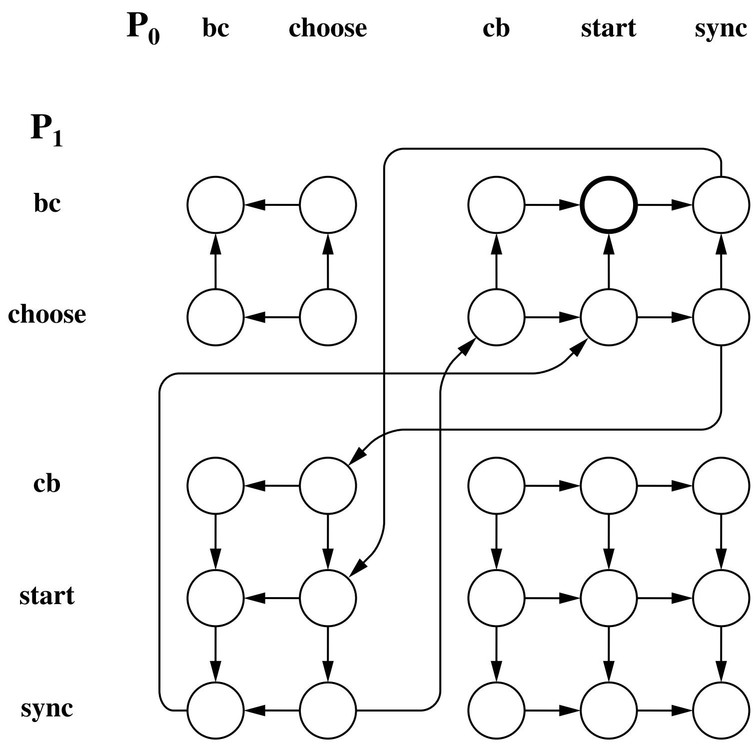


Figure 3.1: Transition Diagram for Milner's Cycler with 2 Processes

$$[\![cyc.0 = sync \Rightarrow cyc.0 = sync \vee cyc.0 = choose]\!] \wedge$$
$$\langle \forall \alpha : \alpha \in Cycler.A : [\![cyc.0 = sync \;\Rightarrow\; \mathbf{wp}\,.\alpha.(cyc.0 = sync \vee cyc.0 = choose)]\!]\rangle$$

which is easily shown to be true. Evaluating this verification condition corresponds to checking for each pair of successive states in the transition diagram that, if the first state satisfies $cyc.0 = sync$, then the second state satisfies $cyc.0 = sync \vee cyc.0 = choose$. This check is performed symbolically: all transitions with the same statement label are checked simultaneously. This property is established automatically without a fixpoint computation because of the locality of the verification condition. A traditional model checking procedure, on the other hand, would have explored the reachable state space starting from the initial state while checking that each new transition encountered does not violate the safety condition.

With the second property we want to illustrate a situation in which a property cannot be established directly. We consider the progress property

$$cyc.0 = start \;\mapsto\; cyc.1 = start$$

which states that whenever process P_0 is in the *start* state then, eventually, process P_1 is in the *start* state as well.

The verification condition for this property requires us to compute the predicate $\mathbf{wlt}\,.(cyc.1 = start)$. The states characterized by $\mathbf{wlt}\,.(cyc.1 = start)$ are shown as filled circles in figure 3.2(a). Having computed this set of states, the model checker determines that the state in which $cyc.0 = start$ and $cyc.1 = sync$ violates the verification condition $[\![cyc.0 = start \Rightarrow \mathbf{wlt}\,.(cyc.1 = start)]\!]$. In other words, the model checker provides the debugging information that there exists a fair execution of the program starting in this marked state that never reaches a state in which $cyc.1 = start$ holds. This violating state is marked with a cross in figure 3.2(a).

After this initial verification check failed the user can attempt to provide a stronger invariant based on his design knowledge while taking the information about the violating state into account. For instance, the user might provide the following

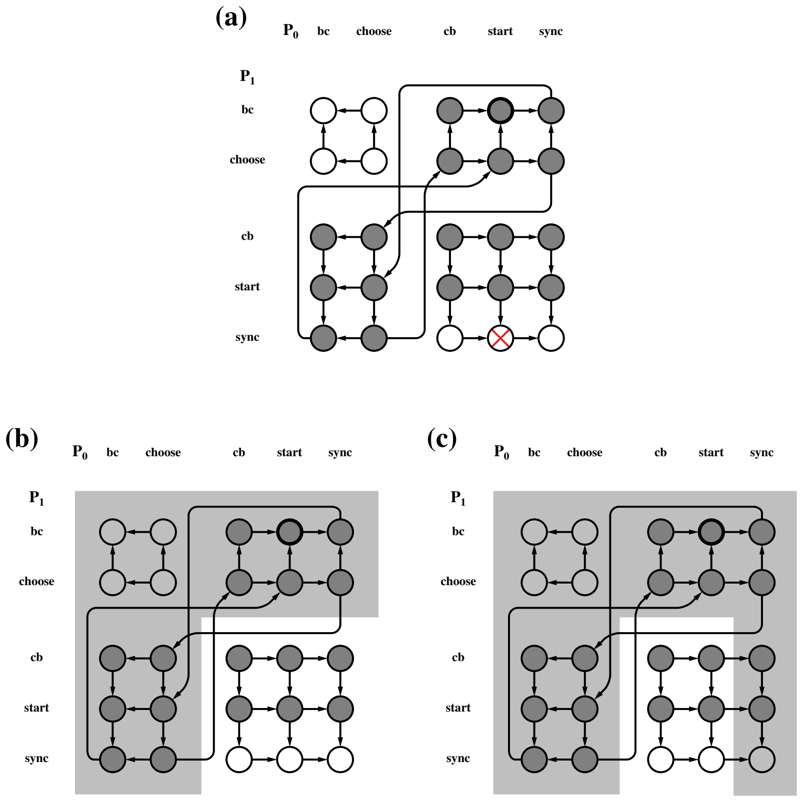


Figure 3.2: **wlt**-check: (a) failed, (b) with J, (c) with K

invariant J:

$$\textbf{invariant}\ \ cyc.0 \in S \ \Rightarrow\ cyc.1 \notin S$$

where $S = \{cb, start, sync\}$[2]. Invariant J is established directly using the verification condition for invariants. Furthermore, using J, the leads-to property is also established because there are no violating states in the restriction of the state space to J. This situation is illustrated in figure 3.2(b) in which the states characterized by J are shaded.

Alternatively, after the failure of the initial verification check the model checker can be used to strengthen the current invariant automatically. This is accomplished by starting with the current invariant (true in our example), eliminating the violating state(s), and then repeatedly eliminating those states that violate the stability of the remaining set of states (i.e., the predecessors of previously eliminated states). This process terminates with a stable predicate K, shown in figure 3.2(c) as the shaded set of states. Since K contains the initial state of the program, it is an invariant. It follows that the leads-to property is established automatically since there is no state in the state space restricted to K that violates the leads-to condition.

3.5 Discussion

In this chapter we have derived a model checking procedure for UNITY logic that takes advantage of the structure of the UNITY proof system to obtain efficient checking conditions for all safety and basic progress properties. A big performance gain is achieved when fixpoint computations can be replaced by evaluation of certain local checking conditions, possibly as a result of supplying state-based design knowledge in the form of invariants.

[2] The rationale for this invariant is given in section 7.4.

While a complete automation of mechanical verification is an important goal in particular for industrial applications, we can argue that for a situation in which program design and verification are carried out together, it is at least worth trying to utilize the available design knowledge for speeding up the verification. If these attempts fail, one can always resort to the fully automated methods based on fixpoint computations.

We also want to point out again that the suggested verification method based on direct provability and the finding of suitable invariants is not per se limited to finite state systems. For finite state systems the method is decidable and, when combined with suitable representation techniques, practical for possibly very large systems. However, it can also be applied in a more general context such as a part of a theorem prover potentially capable of dealing with infinite state spaces, or with unbounded families of parameterized programs.

The results obtained in this chapter can also be applied to logics that contain (part of) UNITY as a sub-logic. For instance, the safety properties of UNITY logic are contained in LTL and in CTL. Therefore a similar treatment of invariants and local checking conditions can be incorporated into LTL and CTL model checkers provided the specification formulae are restricted to the UNITY subset. In order to take advantage of the checking conditions for the basic progress properties **transient** and **ensures** , a logic must be able to meet the unconditional fairness of UNITY, and must be able to distinguish different actions (e.g., indexed CTL).

We also note that with an encoding of transition labels into successor states and with a suitable fairness constraint, Fair CTL ([EL85]) contains the leads-to properties of UNITY logic. However, the verification of such properties using the UNITY checking conditions is not improved; in fact, the calculation of **wlt** corresponds exactly to the fixpoint computations for leads-to properties in Fair CTL with the appropriate fairness constraint. The checking of general progress properties is, therefore, not improved by the proposed method. In the next chapter we investigate a new extension to UNITY logic that will enable us to utilize action-based design knowledge for increasing the efficiency and effectiveness of verification of progress properties.

Chapter 4

A Generalization of Progress

Progress properties constitute a large class of properties frequently encountered in the specification of concurrent systems. Informally characterized as expressing that "*something good will happen*" [Lam77], progress properties are used for specifying or asserting important achievements of concurrent systems such as guarantee of response to service requests, convergence, or absence of starvation. Different from safety properties, which are informally described as expressing that "*nothing bad ever happens*", progress properties inherently refer to infinite program executions in that finite prefixes of program executions do not suffice to characterize them [AS85].

Automatic Checking of Progress Properties is Difficult in Practice

In chapter 3 we have seen how checking safety properties of UNITY programs can be reduced to checking certain local conditions and to finding suitably strong invariants. A similar approach fails in general for progress properties, because such properties cannot be characterized by local checking conditions asserting how pairs of successive states are related to each other (if this were possible then progress properties could be characterized by finite execution prefixes as well); instead, some global variance information needs to be provided in order to measure the progress towards some goal.

There has been significant success with various techniques and methods for mechanical verification or automated verification support, in particular theorem provers and model checkers, in dealing with and in reasoning about safety properties. In practice, these verification methods have been found to be often less useful and attractive when applied to progress properties: establishing progress properties is complicated for theorem proving by the need to perform transfinite induction over a well-founded set, whereas for model checking the alternation depth 2 of characterizations of progress under fairness limits symbolic approaches that are so far the most successful ones. As a consequence, it is not uncommon to formally verify only safety aspects of concurrent systems and to argue about progress properties informally, or even to consider progress properties as practically not very important. With our work we want to help with improving the manageability of proving progress properties while making mechanical checking of progress properties more efficient.

It is our goal in this chapter to propose a new notion of generalized progress properties and to develop a theory for it that makes it possible to derive ordinary[1] progress properties from generalized ones, to utilize design knowledge during reasoning about progress properties, and to improve the performance of automated checking procedures based on the new progress properties.

Generalized Progress Theory Improves Checking Progress Properties

The key idea for accomplishing these goals is to formalize and thereby to make explicit the way in which progress is achieved by a program. We do so by introducing *generalized progress properties* of the form

$$p \overset{W}{\longmapsto} q$$

(pronounced *p leads-to q by W*) for state predicates p and q and for regular ex-

[1]We use the term *ordinary* in connection with progress properties from now on to indicate the traditional notion of progress and to distinguish it from our new notion of *generalized* progress.

pression W over the alphabet of actions of the program under consideration. The regular expression W captures how progress is achieved from states satisfying p to states satisfying q. Using generalized progress properties in the design process for concurrent systems is advantageous in two ways: the designer can incorporate action based design knowledge in the design and verification process, and the actual mechanical verification can possibly be performed more efficiently. The increase in efficiency is due to the fact that a verification system can utilize the information contained in the hints supplied in form of regular expressions in order to eliminate some unnecessary fixpoint computations and to simplify others. The effective use of such action based design knowledge, however, is of great importance beyond improving verifier performance, by making it possible for the designer to provide hints in a formal manner that become part of the verification process and can be used in exploring and debugging programs and specifications. Moreover, the level of provided design knowledge is scalable in the sense that whatever is made available can be used, while without any specific design knowledge the generalized progress properties coincide with the ordinary ones.

The UNITY Advantage

In section 3.1 we have seen how the local checking conditions for safety properties and for basic progress properties were derived from the corresponding proof rules for safety and basic progress of UNITY logic (cf. section 2.2). In a similar fashion the formalization of progress via regular expressions is mostly determined by the UNITY proof rules for leads-to properties: progress by single actions corresponds to **ensures** properties, where the helpful action is named explicitly; sequencing corresponds to transitivity, alternation to finite disjunctions, and repetition to application of leads-to induction. The combination of simplicity of formal specifications and of a rich structure of the deductive system makes UNITY a very suitable basis for the development of the theory of generalized progress and its practical application.

Different Semantics for Generalized Progress

There are different ways for developing and presenting an extension to an existing logic and its semantics: properties can be described operationally in terms of program executions; they can be characterized in relational terms by a set of axioms and proof rules; or they can be characterized by suitable predicate transformers. In our presentation we use different approaches in order to deal with different aspects of the theory in the most suitable way, while demonstrating the close relationship of these approaches to each other.

The *operational characterization* relates generalized progress properties to program executions that often play a significant role in the conception and design of programs, either due to an informal operational specification of the desired program, or as a complementary technique enabling the designer to deal with and to concretize inherently informal aspects of a program specification. An operational semantics is therefore an important tool for capturing and expressing design knowledge.

Defining a property as a *relation over predicates* gives rise to a proof system which is often the most suitable formalism to establish properties of programs under consideration.

Finally, a characterization of generalized progress properties based on *predicate transformers* is appealing because its uniformity makes it suitable for calculational proofs to be used for proving meta-theorems about properties as well as automating proofs of properties for specific programs by model checking.

Organization of This Chapter

The remainder of this chapter is organized as follows: in section 4.1 we introduce the idea of generalized progress properties and relate such properties informally to both program executions and to proofs of ordinary progress properties. In section 4.2 we present a formal semantics for generalized progress properties based on predicate transformers. In section 4.3 we provide a deductive system for proving generalized progress properties of given programs and relate it to predicate transformer semantics. An investigation of the algebraic structure of the family of generalized progress properties for different regular expressions is given in section 4.4. A presentation of an operational semantics for the generalized progress properties in section 4.5 and a discussion of the new theory in section 4.6 conclude the chapter.

4.1 Introduction

Progress properties are generally specified using some temporal operator, be it the *leads-to* operator of UNITY logic, formulae of the form $\mathbf{AG}(p \Rightarrow \mathbf{AF}q)$ of CTL, or formulae like $\mathbf{G}(p \Rightarrow \mathbf{F}q)$ of linear temporal logic. Common to the ways these operators are defined is the fact that they abstract away from how progress is achieved by hiding any reference to individual program actions or variant functions. This abstraction makes it possible to characterize progress simply by pairs of state predicates; however, establishing such progress properties for a given program requires that individual program actions be made explicit (for instance in carrying out a deductive proof), or at least be handled anonymously (for instance in model checking algorithms).

The Advantages of Providing Design Knowledge

Although the formal verification of progress properties is highly desirable, it is often unrealistic to require the designer to provide a *complete* proof of such a property. The level of detail required to carry out a formal proof is often well beyond what is considered practical during the design process; moreover, the designer's expertise might not be sufficient to carry out such a proof efficiently. In spite of these practical difficulties, it is undoubtedly the case that the designer has some (possibly partial) understanding of how progress is achieved by the designed program, either in the form of some operational argument, in the form of a high level proof sketch, or simply based on experience with similar programs. In all these cases a formalism that allows the designer to express such partial, often operational, design knowledge in a simple way and that takes advantage of that knowledge in the verification process, has the potential of improving the effectiveness of the verification task by conducting a proof (by model checking) of some properties without forcing the designer to supply a complete proof herself.

If the partial knowledge supplied by the designer is sufficient to derive such a proof efficiently, the interactive verification has been completed successfully. If such a proof cannot be found efficiently (or cannot be found at all), the designer needs to be assisted in supplying further (more specific) design knowledge. If she cannot provide such knowledge, or if the provided knowledge causes any inconsistency, appropriate debugging information should be made available to aid in either debugging the design, or in pointing out parts that require additional design knowledge in order to be verified efficiently. However, not more information should be asked for from the designer than what is necessary for establishing the required properties.

By providing a way of making this form of design knowledge formally accessible to reasoning about programs, and by taking advantage of such knowledge in the verification process, we could meet our goals of providing a more powerful and effective way of dealing with progress properties than by the use of the leads-to operator alone.

An Example

In the following we present a small example to illustrate the idea of how progress properties are generalized by including explicit action-based progress information. We then argue how this new theory and its associated methodology can be used in program verification.

Let us consider the following program *UpDown*, in which n is an integer counter, that can always be decremented, but can only be incremented if the boolean variable b is false:

program *UpDown*

declare

var n : *integer*

var b : *boolean*

assign

[up]	$n := n+1$	**if** $\neg b$
[down]	$n := n-1$	
[set]	$b :=$ true	

end

A progress property of program *UpDown* is that in any execution eventually n becomes negative, which is expressed by the following UNITY leads-to property:

$$\text{true} \quad \mapsto \quad n < 0.$$

The following informal argument serves as a justification for this claim: given any execution and any state reached during the execution in which n is not negative, there is some subsequent state in which b holds: at the latest the first state after an execution of [set] [2] satisfies b, due to the unconditional fairness there always is such a next occurrence of [set], and no other action falsifies b. In such a state satisfying b, either n is negative or a finite number of executions of [down] (namely, one more than the value of n in that state) makes n negative. Again, due to the unconditional fairness there are sufficiently many such occurrences of [down], and no other action interferes with [down] by increasing n or falsifying b. Therefore, for any execution and any state there is a future state[3] satisfying $n < 0$.

The above operational argument suggests that one can think of the progress from true to $n < 0$ as being achieved by virtue of a strategy expressed by the regular expression

[2] Square brackets are used in the following as part of the action labels.

[3] possibly the current state

$$[\text{set}][\text{down}]^*,$$

over the alphabet of actions of *UpDown*, i.e., by one [set] action followed by some finite number of [down] actions. Using the notation for generalized progress properties, we can state that program *UpDown* satisfies the property

$$\text{true} \overset{[\text{set}][\text{down}]^*}{\longmapsto} n < 0.$$

Before we formalize the notion of a strategy for achieving progress and define the operational semantics of generalized progress properties, we show that there is not only an operational interpretation of such regular expression strategies, but also a close connection to proofs in the deductive system of UNITY logic.

To this end we present a proof in UNITY logic of the property true $\mapsto n < 0$ of program *UpDown*:

0. true **ensures** b

   ; from program text via [set]

1. true $\mapsto b$

   ; promotion from 0

2. $b \wedge n = k$ **ensures** $b \wedge n < k$

   ; from program text via [down]

3. $b \wedge n = k \mapsto b \wedge n < k$

   ; promotion from 2

4. $b \wedge |n| = k \mapsto (b \wedge |n| < k) \vee n < 0$

   ; arithmetic case split and disjunction on 3

5. $b \mapsto n < 0$

   ; leads-to induction on 4 with metric $|n|$ over the naturals

6. true $\mapsto n < 0$

   ; transitivity with 1 and 5

Steps 0 and 1 of this proof correspond to the [set] action in our proposed strategy; similarly steps 2 and 3 correspond to the [down] action. The inductive argument of steps 4 and 5 establishes the progress via $[\text{down}]^*$; finally, step 6, combines the substrategies by sequencing. We therefore can argue that the above proof has a structure corresponding to the regular expression $[\text{set}][\text{down}]^*$. However, this regular expression is much less detailed than the complete proof: in particular, the state predicates needed to combine the different parts of the proof together, are omitted. Hence, even a general idea about the structure of a proof of a progress property can be turned into a strategy without requiring a complete and detailed proof.

Some Ways for Providing Design Knowledge

In summary, we have suggested that there are two ways in which the designer of a concurrent system could formulate her design knowledge and come up with strategies in the form of regular expressions for progress properties of the program under consideration: suitable regular expressions can either be proposed based on the operational understanding of the program under construction, or can be derived from a high-level approximation of a deductive proof of the property. Both techniques help the designer to express design knowledge and to convey it to a verifier that can take advantage of the information provided. Moreover, the expressed design knowledge can play an important role in reasoning about the program, in testing hypotheses, and in debugging both programs and properties.

In the rest of this chapter we develop the theory of generalized progress. The practical application of this theory to model checking, as well as a demonstration of the techniques for utilizing design knowledge, follows in chapter 5.

4.2 A Predicate Transformer Semantics

Predicate transformers have been successfully used for defining and reasoning about semantics of both sequential ([Dij76, DS90]) and concurrent ([JKR89, Kna92]) programs. In this section we present a predicate transformer semantics for the generalized progress properties and use it for investigating some fundamental characteristics of these properties.

We are interested in a predicate transformer semantics for several reasons: the use of predicate transformers allows us to stay completely in the realm of predicate calculus, instead of having to introduce axioms for each relation of a deductive system, thereby facilitating a calculational style of reasoning; certain characteristics of the generalized leads-to operator, such as monotonicity and continuity, can be stated and answered more easily using predicate transformers; moreover, the predicate transformers prove to be invaluable for investigating the algebraic structure of the generalized progress properties (cf. section 4.4); and finally, the fixpoint definitions of the predicate transformers for generalized progress give rise to symbolic algorithms which we subsequently use in our new model checking procedures for progress properties in chapter 5.

The treatment of the predicate transformers for generalized progress is organized as follows: in section 4.2.1 we introduce a family of predicate transformers **wltr** (for *weakest leads-to by regular expression*) by giving a fixpoint characterization and exhibiting some basic properties of them. In section 4.2.2, we investigate the junctivity properties of the **wltr** predicate transformers. In section 4.2.3 we establish the close connection between the **wltr** predicate transformers and the **wlt** predicate transformer (for *weakest leads-to*, [JKR89]) characterizing the ordinary leads-to properties of UNITY logic.

4.2.1 Predicate Transformers for Generalized Progress

We begin the formal treatment of the semantics of generalized progress properties by defining a family of predicate transformers $\mathbf{wltr}.W$ for any W in $\mathcal{R}_F$ inductively over the structure of W:

Definition 2 *For any W in $\mathcal{R}_F$, the predicate transformer* $\mathbf{wltr}.W$ *is defined inductively over the structure of W such that for all α in $F.A$, all U, V in $\mathcal{R}_F$, and all state predicates q in $\mathcal{P}_F$:*

$$[\![\mathbf{wltr}.\varepsilon.q \quad \equiv \quad q]\!] \qquad \textbf{\textit{(wltrEps)}}$$

$$[\![\mathbf{wltr}.\alpha.q \quad \equiv \quad \langle \nu Z :: (\mathbf{wco}.(Z \vee q) \wedge \mathbf{wp}.\alpha.q) \vee q \rangle]\!] \qquad \textbf{\textit{(wltrAct)}}$$

$$[\![\mathbf{wltr}.(UV).q \quad \equiv \quad \mathbf{wltr}.U.(\mathbf{wltr}.V.q)]\!] \qquad \textbf{\textit{(wltrSeq)}}$$

$$[\![\mathbf{wltr}.(U+V).q \quad \equiv \quad \mathbf{wltr}.U.q \ \vee \ \mathbf{wltr}.V.q]\!] \qquad \textbf{\textit{(wltrAlt)}}$$

$$[\![\mathbf{wltr}.U^*.q \quad \equiv \quad \langle \mu Z :: q \ \vee \ \mathbf{wltr}.U.Z \rangle]\!] \qquad \textbf{\textit{(wltrStar)}}$$

Informally, the predicate $\mathbf{wltr}.W.q$ is intended to characterize all those states from which any execution characterized by W leads to a state satisfying q, where the notion of executions characterized by regular expressions will be made precise later in section 4.5.

Even with a very informal understanding of progress characterized by regular expressions, we can attempt to motivate the above definitions. To this end we note that **(wltrEps)** captures precisely the notion of progress without actions, where the start predicate is the goal predicate, that **(wltrSeq)** captures the notion of sequencing which corresponds to the functional composition of the predicate transformers of the subexpressions, and that **(wltrAlt)** captures the notion of choice which corresponds to the disjunction of the predicate transformers of the subexpressions.

The form of **(wltrAct)** is suggested by observing that for any action α in $\mathcal{R}_F$ we expect $\mathbf{wltr}.\alpha.q$ to be the weakest predicate that ensures q via α, i.e.,

$$\mathbf{wltr}.\alpha.q \ \mathbf{ensures}_\alpha \ q \qquad \textbf{(E0)}$$

$$Z \ \mathbf{ensures}_\alpha \ q \quad \Rightarrow \quad [Z \Rightarrow \mathbf{wltr}.\alpha.q] \qquad \textbf{(E1)}$$

Manipulating the definition of $\mathbf{ensures}_\alpha$ we observe

$$Z \ \mathbf{ensures}_\alpha \ q$$
$$\equiv \quad \{\text{definition of } \mathbf{ensures}_\alpha \}$$

$$[Z \wedge \neg q \Rightarrow \mathbf{wco}\,.(Z \vee q)] \wedge [Z \wedge \neg q \Rightarrow \mathbf{wp}\,.\alpha.q]$$
$$\equiv \quad \{\text{shunting, predicate calculus}\}$$
$$[Z \Rightarrow ((\mathbf{wco}\,.(Z \vee q) \wedge \mathbf{wp}\,.\alpha.q) \vee q)]$$

In order to characterize the weakest predicate that ensures q via α it therefore seems reasonable to ask for the weakest solution of the following equation:

$$Z : [\![Z \;\Rightarrow\; ((\mathbf{wco}\,.(Z \vee q) \wedge \mathbf{wp}\,.\alpha.q) \vee q)]\!]$$

The righthand side of this equation is monotonic in Z and therefore (by the Knaster-Tarski Theorem) has a weakest solution, which is the same as the weakest solution of the equation

$$Z : [\![Z \;\equiv\; ((\mathbf{wco}\,.(Z \vee q) \wedge \mathbf{wp}\,.\alpha.q) \vee q)]\!]$$

which we denote by $\langle \nu Z :: (\mathbf{wco}\,.(Z \vee q) \wedge \mathbf{wp}\,.\alpha.q) \vee q \rangle$ and which we use as our definition of $\mathbf{wltr}\,.\alpha.q$. We note that by virtue of the above construction and by the relationship between the everywhere operators it follows that $\mathbf{wltr}\,.\alpha.q$ satisfies **(E0)** and **(E1)**.

For **(wltrStar)** we expect that $\mathbf{wltr}\,.U^*.q$ be the weakest predicate that leads to q by some finite sequence of executions of U. Together with the observation that $\mathbf{wltr}\,.U^*.q$ certainly is at least as weak as q, this makes it seem reasonable to ask for a solution of the equation

$$Z : [\![Z \;\equiv\; q \vee \mathbf{wltr}\,.U.Z]\!]$$

Due to the required finiteness of the repetitions we need to consider the strongest solution of the above equation. In the next paragraph, we establish that for any W in $\mathcal{R}_F$ $\mathbf{wltr}\,.W$ is a monotonic predicate transformer, by induction on the structure of W. It follows that the righthand side of the above equation is monotonic in Z, and that therefore (again by the theorem of Knaster-Tarski) the equation has indeed a least fixpoint which we denote by $\langle \mu Z :: q \vee \mathbf{wltr}\,.U.Z \rangle$ and which we use as our definition of $\mathbf{wltr}.U^*.q$.

As some important properties of the **wltr** predicate transformers we establish that each such predicate transformer is monotonic, strict, and weakening:

Lemma 6 (Basic Properties of wltr) *For any W in $\mathcal{R}_F$ and any p, q in $\mathcal{P}_F$:*

$$[\![p \Rightarrow q]\!] \;\Rightarrow\; [\![\mathbf{wltr}.W.p \Rightarrow \mathbf{wltr}.W.q]\!] \qquad \textit{\textbf{(wltrMon)}}$$

$$[\![\mathbf{wltr}.W.\text{false} \;\equiv\; \text{false}]\!] \qquad \textit{\textbf{(wltrStrict)}}$$

$$[\![q \Rightarrow \mathbf{wltr}.W.q]\!] \qquad \textit{\textbf{(wltrWeaken)}}$$

Proof . We prove each property by induction over the structure of W. For **(wltrMon)** we observe the following: $\mathbf{wltr}.\varepsilon$ is the identity function and is therefore obviously monotonic; $\mathbf{wltr}.\alpha$ is defined as a fixpoint of a monotonic equation and is therefore monotonic itself; both $\mathbf{wltr}.(UV)$ and $\mathbf{wltr}.(U+V)$ are defined by monotonic functions of the (by the induction hypothesis) monotonic predicate transformers $\mathbf{wltr}.U$ and $\mathbf{wltr}.V$; and $\mathbf{wltr}.U^*$ is defined as a fixpoint of a monotonic equation using the (by the induction hypothesis) monotonic predicate transformer $\mathbf{wltr}.U$, and is therefore monotonic as well.

For **(wltrStrict)** we show only the cases corresponding to single actions and to repetition, the remaining cases are trivial. We observe for any α in $F.A$:

$$\begin{array}{ll}
& \mathbf{wltr}.\alpha.\text{false} \\
\equiv & \{\textbf{(wltrAct)}\} \\
& \langle \nu Z :: (\mathbf{wco}.(p \vee \text{false}) \wedge \mathbf{wp}.\alpha.\text{false}) \vee \text{false}\rangle \\
\equiv & \{\mathbf{wp}.\alpha \text{ is strict, predicate calculus}\} \\
& \langle \nu Z :: \text{false}\rangle \\
\equiv & \{\text{predicate calculus}\} \\
& \text{false}
\end{array}$$

and for any W in $\mathcal{R}_F$:

$$\mathbf{wltr}.W^*.\text{false}$$

$\equiv$ {**(wltrStar)**}

$\langle \mu Z :: \text{false} \ \vee \ \mathbf{wltr}.U.Z \rangle$

$\equiv$ {induction hypothesis, predicate calculus}

false

Similarly, the proof for **(wltrWeaken)** is straight forward if we use the fact that in the cases corresponding to single actions and repetition any fixpoint of the respective equations is implied by q.

End of Proof.

4.2.2 Junctivity Properties of the wltr Predicate Transformers

We have already established in the previous section that for each W in $\mathcal{R}_F$ the predicate transformer $\mathbf{wltr}.W$ is monotonic. In this section we show that in general $\mathbf{wltr}.W$ does not enjoy other interesting junctivity properties. In particular we demonstrate that $\mathbf{wltr}.W$, where W contains some action from $F.A$, is neither finitely disjunctive nor finitely conjunctive, that it is not or-continuous, and is and-continuous only if the regular expressions W does not contain the * operator (see section 2.1.4 for the definition of junctivity properties).

We start with the results about and-continuity:

Lemma 7 *For any W in $\mathcal{R}_F$ not containing the repetition operator *,* $\mathbf{wltr}.W$ *is and-continuous. Allowing the * operator in general destroys and-continuity.*

Proof . The proof proceeds by induction on the structure of W. We show that all operators but the * operator preserve and-continuity, and that the * operator in general destroys it.

Since $[\![\mathbf{wltr}.\varepsilon.q \equiv q]\!]$ by **(wltrEps)**, $\mathbf{wltr}.\varepsilon$ is trivially and-continuous. By **(wltrAct)**, $\mathbf{wltr}.\alpha.q$ for some α in $F.A$ is the weakest solution of the equation

$$X : [\![X \ \equiv \ \tau.X.q]\!]$$

where

$$[\![\tau.X.Y \;\equiv\; (\mathbf{wco}.(X \vee Y) \wedge \mathbf{wp}.\alpha.Y) \vee Y]\!]$$

From the universal conjunctivity of **wco** and **wp** and the and-continuity of disjunction it follows that τ is and-continuous. Hence, by theorem 2, $\mathbf{wltr}.\alpha$ is and-continuous as well. Function composition and disjunction preserve and-continuity, hence for and-continuous $\mathbf{wltr}.U$ and $\mathbf{wltr}.V$ both $\mathbf{wltr}.UV$ and $\mathbf{wltr}.(U+V)$ are and-continuous as well.
To show that the * operator does not preserve and-continuity we consider the program

program *AndContinuity*
  declare
    var n : *integer*
  assign
    $[\alpha]$ $n := n+1$
end

and define for any natural i the predicate $Q.i$ by $[\![Q.i \equiv n \geq i]\!]$. Clearly, $\langle \forall i,j :: i \leq j \Rightarrow [\![Q.i \Leftarrow Q.j]\!]\rangle$, hence $\{i : i \in \mathbf{N} : Q.i\}$ is linear. For this set of predicates we observe that

$$\begin{array}{ll}
 & \mathbf{wltr}.\alpha^*.\langle \forall i : i \in \mathbf{N} : Q.i\rangle \\
\equiv & \{\text{definition of } Q.i\text{, arithmetic}\} \\
 & \mathbf{wltr}.\alpha^*.\,\text{false} \\
\equiv & \{(\mathbf{wltrStrict})\} \\
 & \text{false}
\end{array}$$

but that

$$\begin{array}{ll}
 & \langle \forall i : i \in \mathbf{N} : \mathbf{wltr}.\alpha^*.(Q.i)\rangle \\
\equiv & \{\text{calculation using } (\mathbf{wltrStar}),\ (\mathbf{wltrAct})\}
\end{array}$$

$\langle\forall i : i \in \mathbf{N} : \text{true}\rangle$

$\equiv$ {predicate calculus}

true

End of Proof.

The remaining results are presented in the following lemma:

Lemma 8 *For any W in $\mathcal{R}_F$ not containing actions from F.A,* **wltr**$.W$ *is both universally disjunctive and universally conjunctive. For any W containing at least one action from F.A,* **wltr**$.W$ *is in general*

(0) *not or-continuous,*

(1) *not finitely disjunctive,*

(2) *not finitely conjunctive.*

Proof . It is easily seen by induction on the structure of W, that $[\![\mathbf{wltr}.W.q \equiv q]\!]$ for any W built from ε, sequencing, alternation, and repetition alone. For such W, **wltr**$.W$ is therefore clearly universally disjunctive and universally conjunctive. For the three negative results we exhibit counterexamples:

ad **(0)**: Consider the program

program *OrContinuity*

declare

var n : *integer*

var $d : \{-1, 1\}$

assign

$[\alpha]$ $\quad n, d := n + d, -1$

$[\beta]$ $\quad n := n + d$

end

which is a slight modification of the program of section 4.1.

For each natural i define $Q.i$ by $[\![Q.i \;\equiv\; n \leq 0 \vee (d = -1 \wedge n \leq i)]\!]$. Clearly, $\langle \forall i, j :: i \leq j \Rightarrow [\![Q.i \Rightarrow Q.j]\!]\rangle$, hence $\{i : i \in \mathbf{N} : Q.i\}$ is linear. Furthermore, it can be shown that $[\![\mathbf{wltr}.\alpha.(Q.i) \equiv Q.(i+1)]\!]$. From this it follows that

$$\begin{array}{ll} & \mathbf{wltr}.\alpha.\langle \exists i : i \in \mathbf{N} : Q.i\rangle \\ \equiv & \{\text{definition of } Q.i\text{, arithmetic}\} \\ & \mathbf{wltr}.\alpha.(n \leq 0 \vee d = -1) \\ \equiv & \{\mathbf{(wltrAct)}\text{, from program}\} \\ & \text{true} \end{array}$$

but that

$$\begin{array}{ll} & \langle \exists i : i \in \mathbf{N} : \mathbf{wltr}.\alpha.(Q.i)\rangle \\ \equiv & \{\mathbf{(wltrAct)}\text{, see above}\} \\ & \langle \exists i : i \in \mathbf{N} : Q.(i+1)\rangle \\ \equiv & \{\text{definition of } Q.i\text{, arithmetic}\} \\ & n \leq 0 \vee d = -1 \end{array}$$

ad **(1)**: Consider the program

program *FiniteDisjunctivity*

declare

var $n : \{0, 1, 2, 3, 4\}$

initially

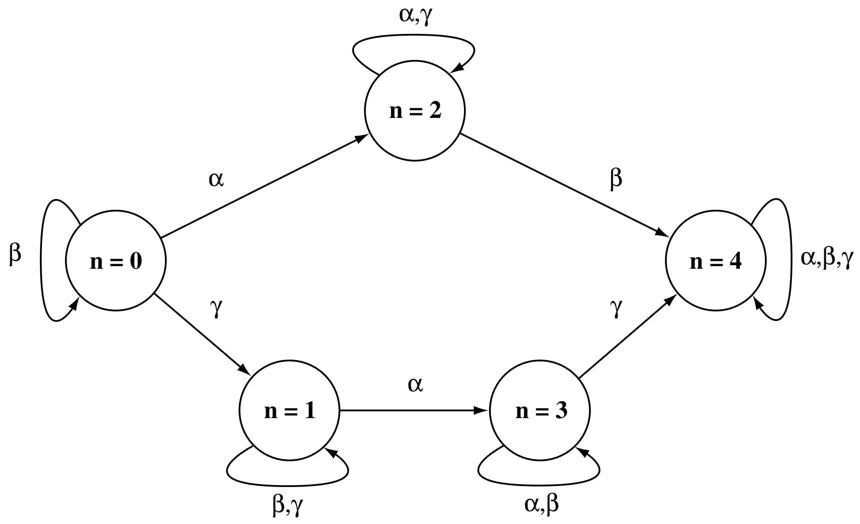
$n = 0$

assign

$[\alpha]$	$n := n + 2$	**if** $n < 2$
$[\beta]$	$n := 4$	**if** $n = 2$
$[\gamma]$	$n := n + 1$	**if** $n = 0 \vee n = 3$

end

which has the following state-transition diagram:



For the predicates p and q defined by $[\![p \;\equiv\; n = 2 \vee n = 4]\!]$ and $[\![q \;\equiv\; n = 3 \vee n = 4]\!]$ we observe that

$$
\begin{array}{ll}
 & \textbf{wltr}.\alpha.(p \vee q) \\
\equiv & \{\text{definition of } p,\, q\} \\
 & \textbf{wltr}.\alpha.(n = 2 \vee n = 3 \vee n = 4) \\
\equiv & \{\text{calculation using } \textbf{(wltrAct)}\} \\
 & \text{true}
\end{array}
$$

but that

$$
\begin{array}{ll}
 & \textbf{wltr}.\alpha.p \;\vee\; \textbf{wltr}.\alpha.q \\
\equiv & \{\text{calculations using } \textbf{(wltrAct)}\} \\
 & (n = 2 \vee n = 4) \;\vee\; (n = 1 \vee n = 3 \vee n = 4) \\
\equiv & \{\text{predicate calculus}\} \\
 & n \neq 0
\end{array}
$$

ad **(2)**: Consider the program

program *FiniteConjunctivity*

  declare

var b : *boolean*

assign

$[\alpha]$ $\quad\quad b := \neg b$

end

and observe for the predicates b and $\neg b$ that

$$\begin{array}{ll} & \mathbf{wltr}.\alpha.(b \wedge \neg b) \\ \equiv & \{\text{predicate calculus}\} \\ & \mathbf{wltr}.\alpha.\,\text{false} \\ \equiv & \{(\mathbf{wltrStrict})\} \\ & \text{false} \end{array}$$

but that

$$\begin{array}{ll} & \mathbf{wltr}.\alpha.b \;\wedge\; \mathbf{wltr}.\alpha.(\neg b) \\ \equiv & \{\text{calculations using } (\mathbf{wltrAct})\} \\ & \text{true} \wedge \text{true} \\ \equiv & \{\text{predicate calculus}\} \\ & \text{true} \end{array}$$

End of Proof.

4.2.3 Relating wltr to wlt

We are now in the position to answer the important question about the relationship between the ordinary leads-to properties of UNITY logic characterized by the **wlt** predicate transformer and our generalized progress properties. In the following we establish the relationship between **wltr** and **wlt**, later in section 4.3.3 we use the result obtained here to state the close connection of the proof systems for ordinary and generalized progress properties.

The relationship between **wltr** and **wlt** is characterized by the following theorem:

Theorem 9 (Relating wltr and wlt) *For any W in $\mathcal{R}_F$, and state predicate q in $\mathcal{P}_F$:*

$$[\![\mathbf{wltr}\,.W.q \;\Rightarrow\; \mathbf{wlt}\,.q]\!] \qquad \textit{(wltrSound)}$$

$$\langle \exists W : W \in \mathcal{R}_F : [\![\mathbf{wlt}\,.q \;\equiv \mathbf{wltr}\,.W.q]\!]\rangle \qquad \textit{(wltrCompl)}$$

The first part of the theorem can be referred to as a *soundness* result since it asserts that states from which a q state is reached by virtue of W are states from which a q state is reached eventually in the sense of ordinary progress. Conversely, the second part can be seen as a *completeness* result since it shows that any state from which a q state is reached eventually is a state from which a q state is reached by virtue of some regular expression W in $\mathcal{R}_F$.

In order to prove these results we recall a few properties of the **wlt** predicate transformer, either taken from [JKR89] or being simple consequences of results found there. First, **wlt** has the following fixpoint characterization:

$$[\![\mathbf{wlt}\,.q \;\equiv\; \langle \mu Z :: q \vee \mathbf{we}\,.Z\rangle]\!] \qquad \textbf{(wlt0)}$$

$$[\![\mathbf{we}\,.q \;\equiv\; \langle \exists \alpha : \alpha \in F.A : \mathbf{stp}\,.\alpha.q\rangle]\!] \qquad \textbf{(wlt1)}$$

$$[\![\mathbf{stp}\,.\alpha.q \;\equiv\; \langle \nu Z :: (\mathbf{wco}\,.Z \wedge \mathbf{wp}\,.\alpha.q) \vee q\rangle]\!] \qquad \textbf{(wlt2)}$$

Furthermore, **wlt** and **stp** enjoy the following properties:

$$[\![q \;\Rightarrow\; \mathbf{wlt}\,.q]\!] \qquad \textbf{(wlt3)}$$

$$[\![\mathbf{stp}\,.\alpha.q \;\Rightarrow\; \mathbf{wlt}\,.q]\!] \qquad \textbf{(wlt4)}$$

$$[\![\mathbf{wlt}\,.(\mathbf{wlt}\,.q) \;\equiv\; \mathbf{wlt}\,.q]\!] \qquad \textbf{(wlt5)}$$

We also need the following lemma relating **stp** and **wltr**:

Lemma 10 *For a program F and any action α in $F.A$:*

$$[\![\mathbf{stp}\,.\alpha \;\equiv\; \mathbf{wltr}\,.\alpha]\!]$$

Proof . $\mathbf{stp}.\alpha.q$ is by **(wlt2)** the weakest solution of the equation $X : [\![X \equiv f.X]\!]$ with f defined by

$$[\![f.X \;\equiv\; (\mathbf{wco}.X \wedge \mathbf{wp}.\alpha.q) \vee q]\!]$$

and $\mathbf{wltr}.\alpha.q$ is by **(wltrAct)** the weakest solution of the equation $X : [\![X \equiv g.X]\!]$ with g defined by

$$[\![g.X \;\equiv\; (\mathbf{wco}.(X \vee q) \wedge \mathbf{wp}.\alpha.q) \vee q]\!]$$

Clearly both f and g are monotonic. Since $\mathbf{wco}$ is monotonic we have $[\![f \Rightarrow g]\!]$, and hence by theorem 3 that $[\![\mathbf{stp}.\alpha \Rightarrow \mathbf{wltr}.\alpha]\!]$.
For the converse, we observe that $\mathbf{wltr}.\alpha$, as a fixpoint of g, satisfies

$$[\![\mathbf{wltr}.\alpha.q \;\equiv\; (\mathbf{wco}.(\mathbf{wltr}.\alpha.q \vee q) \wedge \mathbf{wp}.\alpha.q) \vee q]\!] \qquad \mathbf{(0)}$$

and that $\mathbf{stp}.\alpha$, as the greatest fixpoint of f, satisfies

$$[\![X \equiv f.X]\!] \Rightarrow [\![X \Rightarrow \mathbf{stp}.\alpha.q]\!] \qquad \mathbf{(1)}$$

with which we have for all q:

$$
\begin{array}{ll}
 & [\![\mathbf{wltr}.\alpha.q \;\Rightarrow\; \mathbf{stp}.\alpha.q]\!] \\
\Leftarrow & \{\mathbf{(1)}\} \\
 & [\![\mathbf{wltr}.\alpha.q \;\equiv\; f.(\mathbf{wltr}.\alpha.q)]\!] \\
\equiv & \{[\![\mathbf{wltr}.\alpha.q \equiv q \vee \mathbf{wltr}.\alpha.q]\!] \text{ from } \mathbf{(0)}, \text{ definition of } f\} \\
 & [\![\mathbf{wltr}.\alpha.q \;\equiv\; (\mathbf{wco}.(\mathbf{wltr}.\alpha.q \vee q) \wedge \mathbf{wp}\,\alpha.q) \vee q]\!] \\
\equiv & \{\mathbf{(0)}\} \\
 & \text{true}
\end{array}
$$

End of Proof.

We are now ready to prove the soundness and completeness results.
Proof of theorem 9. For **(wltrSound)** we observe for all q, and W by induction over the structure of W:

case $W = \varepsilon$:

$$\begin{array}{ll} & \mathbf{wltr}.\varepsilon.q \\ \equiv & \{\mathbf{(wltrEps)}\} \\ & q \\ \Rightarrow & \{\mathbf{(wlt3)}\} \\ & \mathbf{wlt}.q \end{array}$$

case $W = \alpha$ for some $\alpha \in F.A$:

$$\begin{array}{ll} & \mathbf{wltr}.\alpha.q \\ \equiv & \{\text{lemma above}\} \\ & \mathbf{stp}.\alpha.q \\ \Rightarrow & \{\mathbf{(wlt4)}\} \\ & \mathbf{wlt}.q \end{array}$$

case $W = UV$:

$$\begin{array}{ll} & \mathbf{wltr}.UV.q \\ \equiv & \{\mathbf{(wltrSeq)}\} \\ & \mathbf{wltr}.U.(\mathbf{wltr}.V.q) \\ \Rightarrow & \{\text{induction hypothesis, twice, } \mathbf{wltr}.U \text{ is monotonic}\} \\ & \mathbf{wlt}.(\mathbf{wlt}.q) \\ \equiv & \{\mathbf{(wlt5)}\} \\ & \mathbf{wlt}.q \end{array}$$

case $W = U + V$:

$$\begin{array}{ll} & \mathbf{wltr}.(U+V).q \\ \equiv & \{\mathbf{(wltrAlt)}\} \\ & \mathbf{wltr}.U.q \vee \mathbf{wltr}.V.q \\ \Rightarrow & \{\text{induction hypothesis, twice, predicate calculus}\} \\ & \mathbf{wlt}.q \end{array}$$

case $W = U^*$:

$$
\begin{array}{ll}
 & [\![\mathbf{wltr}.U^*.q \;\Rightarrow\; \mathbf{wlt}.q]\!] \\
\equiv & \{(\mathbf{wltrStar})\} \\
 & [\![\langle \mu Z :: q \vee \mathbf{wltr}.U.Z\rangle \;\Rightarrow\; \mathbf{wlt}.q]\!] \\
\Leftarrow & \{\text{induction hypothesis, theorem 3, predicate calculus}\} \\
 & [\![\langle \mu Z :: q \vee \mathbf{wlt}.Z\rangle \;\Rightarrow\; \mathbf{wlt}.q]\!] \\
\Leftarrow & \{\text{least fixpoint property}\} \\
 & [\![q \vee \mathbf{wlt}.(\mathbf{wlt}.q) \;\equiv\; \mathbf{wlt}.q]\!] \\
\equiv & \{(\mathbf{wlt3}),\ (\mathbf{wlt5}),\ \text{predicate calculus}\} \\
 & \text{true}
\end{array}
$$

For **(wltrCompl)** it suffices to exhibit a regular expression C in $\mathcal{R}_F$ and to demonstrate that for C the implication $[\![\mathbf{wlt} \Rightarrow \mathbf{wltr}.C]\!]$ holds. Let L be a sequence of all elements of $F.A$ and define C as

$$C \;=\; (\langle +i : L : L.i\rangle)^*.$$

Using **(wltrEps)**, **(wltrAlt)**, and **(wltrStar)** it is easily seen that

$$[\![\mathbf{wltr}.(\langle +i : L : L.i\rangle).q \;\equiv\; \langle \exists \alpha : \alpha \in F.A : \mathbf{wltr}.\alpha.Z\rangle]\!]$$
$$[\![\mathbf{wltr}.C.q \;\equiv\; \langle \mu Z :: q \vee \langle \exists \alpha : \alpha \in F.A : \mathbf{wltr}.\alpha.Z\rangle\rangle]\!]$$

From $[\![\mathbf{wltr}.\alpha \equiv \mathbf{stp}.\alpha]\!]$ (by the lemma above) and **(wlt0)**, and **(wlt1)**, we see that indeed $[\![\mathbf{wltr}.C \equiv \mathbf{wlt}]\!]$.

End of Proof.

4.3 Reasoning about Generalized Progress

In the previous section we have introduced the notion of generalized progress properties of a program F by defining a family of predicate transformers **wltr** characterizing progress by regular expressions. While such a semantics is very useful for investigating such properties and reasoning about them in a general way, it is often more practical to carry out proofs of progress properties of specific programs using a deductive proof system. It is therefore our goal for this section to present such a proof system for generalized progress properties, to exhibit some useful proof rules, and to establish its connection to the previously presented predicate transformer semantics.

We achieve this goal in three steps: in section 4.3.1 we introduce the *generalized leads-to* relation over regular expressions and pairs of state predicates as the strongest relation generated by a set of inference rules; in section 4.3.2 we show how closely the generalized leads-to relation and the **wltr** predicate transformers are related. Finally, we list some useful proof rules in section 4.3.3.

4.3.1 A Deductive System

For a given program F we define in the following a ternary relation $(\lambda W.\lambda p.\lambda q.\, p \overset{W}{\longmapsto} q)$ (pronounced *p leads-to q by W*) over state predicates p and q in $\mathcal{P}_F$ and regular expressions W in $\mathcal{R}_F$ as the strongest relation generated by a set of inference rules. Such a characterization in terms of inference rules is well suited to carrying out proofs of properties of specific programs.

In order to state the proof rules we need to introduce the notion of a *metric* over the reachable state space of F. In the following we use **Ord** to denote the set of ordinal numbers.

Definition 3 (Metric) *A* metric M *for a given program* F *is a family of state predicates* $\{i : i \in \mathbf{Ord} : M.i\}$ *from* $\mathcal{P}_F$*, such that the following two conditions are met:*

$$[\langle \exists i : i \in \mathbf{Ord} : M.i \rangle] \qquad \textbf{\textit{(MetricExh)}}$$
$$\langle \forall i, j : i \in \mathbf{Ord} \land j \in \mathbf{Ord} : i \neq j \Rightarrow [\neg(M.i \land M.j)] \rangle \qquad \textbf{\textit{(MetricDis).}}$$

The first condition states that the predicates in M exhaust the reachable state space of F, the second asserts that any two predicates with different indices are disjoint.

The predicates in M are totally ordered by the ordering relation $\preceq$ (pronounced *precedes*) obtained from lifting the total order relation on the ordinal indices to the predicates:

$$\langle \forall i, j : i \in \mathbf{Ord} \wedge j \in \mathbf{Ord} : i \leq j \;\equiv\; M.i \preceq M.j \rangle$$

After these remarks we are ready to define the generalized leads-to relation in terms of a set of generating inference rules as follows:

Definition 4 (Generalized Leads-To Relation) *For a given program F the relation $(\lambda W.\lambda p.\lambda q.\ p \overset{W}{\longmapsto} q)$ is the smallest subset of $\mathcal{R}_F \times \mathcal{P}_F \times \mathcal{P}_F$ satisfying the following inference rules for all regular expressions U, V in $\mathcal{R}_F$, for all actions α in $F.A$, for all state predicates p, p', q, r, and s in $\mathcal{P}_F$, and for all metrics M for F:*

$$\frac{[p \Rightarrow q]}{p \overset{\varepsilon}{\longmapsto} q} \qquad \textbf{\textit{(PrEps)}}$$

$$\frac{[p \Rightarrow p'], \quad p' \ \mathbf{ensures}_\alpha\ q}{p \overset{\alpha}{\longmapsto} q} \qquad \textbf{\textit{(PrAct)}}$$

$$\frac{p \overset{U}{\longmapsto} r, \quad r \overset{V}{\longmapsto} q}{p \overset{UV}{\longmapsto} q} \qquad \textbf{\textit{(PrSeq)}}$$

$$\frac{p \overset{U}{\longmapsto} q, \quad r \overset{V}{\longmapsto} q}{p \vee r \overset{U+V}{\longmapsto} q} \qquad \textbf{\textit{(PrAlt)}}$$

$$\frac{[p \Rightarrow p'], \langle \forall i : i \in \mathbf{Ord} : p' \wedge M.i \overset{U}{\longmapsto} (p' \wedge \langle \exists j : j < i : M.j \rangle) \vee q \rangle}{p \overset{U^*}{\longmapsto} q} \qquad \textbf{\textit{(PrStar)}}$$

An important observation is that for any W in $\mathcal{R}_F$ there is exactly one inference rule that can be used for establishing that a triple (W, p, q) satisfies the relation. It follows that the above rules are actually equivalences rather than implications[4]. For instance, if $p \overset{U^*}{\longmapsto} q$ holds for a program F, then there exists a metric M and

[4] The lack of such equivalences for the ordinary leads-to relation was identified as the main shortcoming of leads-to in section 3.

a state predicate p' at least as weak as p, such that for all ordinals i, $p' \wedge M.i \overset{U}{\longmapsto} (p' \wedge \langle \exists j : j < i : M.j \rangle) \vee q$ holds,

We can therefore render the above proof rules in the form of the following equivalences:

$$
\begin{array}{llll}
p \overset{\varepsilon}{\longmapsto} q & \equiv & [p \Rightarrow q] & \textbf{(AxEps)} \\
p \overset{\alpha}{\longmapsto} q & \equiv & \langle \exists p' : [p \Rightarrow p'] : p' \ \textbf{ensures}_{\alpha} \ q \rangle & \textbf{(AxAct)} \\
p \overset{UV}{\longmapsto} q & \equiv & \langle \exists r :: (p \overset{U}{\longmapsto} r) \wedge (r \overset{V}{\longmapsto} q) \rangle & \textbf{(AxSeq)} \\
p \overset{U+V}{\longmapsto} q & \equiv & \langle \exists r, s : [r \vee s \equiv p] : (r \overset{U}{\longmapsto} q) \wedge (s \overset{V}{\longmapsto} q) \rangle & \textbf{(AxAlt)} \\
p \overset{U^*}{\longmapsto} q & \equiv & \langle \exists p' : [p \Rightarrow p'] : \langle \exists M : M \text{ is metric} : & \textbf{(AxStar)} \\
 & & \quad \langle \forall i : i \in \mathbf{Ord} : & \\
 & & \quad\quad p' \wedge M.i \overset{U}{\longmapsto} (p' \wedge \langle \exists j : j < i : M.j \rangle) \vee q \rangle \rangle \rangle &
\end{array}
$$

By virtue of these equivalences and by using structural induction over $\mathcal{R}_F$ we can immediately establish that the generalized leads-to relation is well defined:

Theorem 11 (Well-Definedness of $\overset{W}{\longmapsto}$) *For any given program F the equations* ***(AxEps)***, ***(AxAct)***, ***(AxSeq)***, ***(AxAlt)***, *and* ***(AxStar)*** *uniquely define a family of relations $\{W : W \in \mathcal{R}_F : (\lambda p.\lambda q.\ p \overset{W}{\longmapsto} q)\}$.*

Moreover, if a triple (W, p, q) satisfies the generalized leads-to relation, it can be shown to do so by a finite number of applications of the above proof rules (due to the finiteness of the structure of W). Therefore every element (W, p, q) in the relation has a finite proof; from now on we write $F \vdash p \overset{W}{\longmapsto} q$ to denote that (W, p, q) satisfies the generalized leads-to relation.

It is also worth mentioning that the use of ordinals is essential in **(AxStar)**: under unconditional fairness it is generally not possible to bound the number of steps required to achieve progress from any particular start state, it can only be asserted that a finite number of steps suffices. Therefore natural numbers as metric are not sufficient; instead all ordinals of cardinality less than or equal to the cardinality of the size of the state space have to be considered.

4.3.2 A Characterization of Provability

The main result of this section is to establish the very close connection between the generalized leads-to relation just defined and the **wltr** predicate transformers of section 4.2. The result is analogous to the one relating the ordinary leads-to relation to the **wlt** predicate transformer in [JKR89, Kna92] and is stated in the following theorem:

Theorem 12 *For any W in $\mathcal{R}_F$ and state predicates p and q in $\mathcal{P}_F$:*

$$[p \Rightarrow \mathbf{wltr}.W.q] \quad \equiv \quad p \overset{W}{\longmapsto} q$$

Proof . From theorem 11 the generalized leads-to relation is uniquely defined by the equations **(AxEps)**, **(AxAct)**, **(AxSeq)**, **(AxAlt)**, and **(AxStar)**. It is therefore sufficient to show that the relation $\mathbf{R}$ defined as

$$\mathbf{R}.W.p.q \quad \equiv \quad [p \Rightarrow \mathbf{wltr}.W.q]$$

solves the same equations. We establish this by induction over the structure of W by observing for all p and q in $\mathcal{P}_F$:

case $W = \varepsilon$:

$$\begin{array}{ll} & \mathbf{R}.\varepsilon.p.q \\ \equiv & \{\text{definition of } \mathbf{R}\} \\ & [p \Rightarrow \mathbf{wltr}.\varepsilon.q] \\ \equiv & \{(\mathbf{wltrEps})\} \\ & [p \Rightarrow q] \\ \equiv & \{(\mathbf{AxEps})\} \\ & p \overset{\varepsilon}{\longmapsto} q \end{array}$$

case $W = \alpha$ for some $\alpha \in F.A$:

$$\mathbf{R}.\alpha.p.q \quad \equiv \quad p \overset{\alpha}{\longmapsto} q$$

$\equiv$ {definition of $\mathbf{R}$, $\mathbf{(AxAct)}$}

$$[p \Rightarrow \mathbf{wltr}.\alpha.q] \quad \equiv \quad \langle \exists p' : [p \Rightarrow p'] : p' \ \mathbf{ensures}_\alpha \ q \rangle$$

which we prove by mutual implication:

$$
\begin{array}{ll}
& [p \Rightarrow \mathbf{wltr}.\alpha.q] \\
\Rightarrow & \{\text{predicate calculus, } \mathbf{(E0)}, \ \mathbf{wltr}.\alpha.q \text{ is witness for } p'\} \\
& \langle \exists p' : [p \Rightarrow p'] : p' \ \mathbf{ensures}_\alpha \ q \rangle \\
\Rightarrow & \{\mathbf{(E1)}\} \\
& \langle \exists p' : [p \Rightarrow p'] : [p' \Rightarrow \mathbf{wltr}.\alpha.q] \rangle \\
\Rightarrow & \{\text{predicate calculus}\} \\
& [p \Rightarrow \mathbf{wltr}.\alpha.q]
\end{array}
$$

case $W = UV$:

$$
\begin{array}{ll}
& \mathbf{R}.(UV).p.q \\
\equiv & \{\text{definition of } \mathbf{R}\} \\
& [p \Rightarrow \mathbf{wltr}.(UV).q] \\
\equiv & \{\mathbf{(wltrSeq)}\} \\
& [p \Rightarrow \mathbf{wltr}.U.(\mathbf{wltr}.V.q)] \\
\equiv & \{\text{monotonicity of } \mathbf{wltr}.U, \text{ predicate calculus,} \\
& \quad \mathbf{wltr}.V.q \text{ is witness for } r\} \\
& \langle \exists r :: [p \Rightarrow \mathbf{wltr}.U.r] \wedge [r \Rightarrow \mathbf{wltr}.V.q] \rangle \\
\equiv & \{\text{induction hypothesis, twice}\} \\
& \langle \exists r :: (p \overset{U}{\longmapsto} r) \wedge (r \overset{V}{\longmapsto} q) \rangle \\
\equiv & \{\mathbf{(AxSeq)}\} \\
& p \overset{(UV)}{\longmapsto} q
\end{array}
$$

case $W = U + V$:

$$
\begin{array}{ll}
& \mathbf{R}.(U+V).p.q \\
\equiv & \{\text{definition of } \mathbf{R}\} \\
& [p \Rightarrow \mathbf{wltr}.(U+V).q] \\
\equiv & \{\mathbf{(wltrAlt)}\} \\
& [p \Rightarrow \mathbf{wltr}.U.q \vee \mathbf{wltr}.V.q]
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{\text{predicate calculus, } p \wedge \mathbf{wltr}.U.q \text{ is witness for } r, \\
& \quad p \wedge \mathbf{wltr}.V.q \text{ is witness for } s\} \\
& \langle \exists r, s : [r \vee s \equiv p] : [r \Rightarrow \mathbf{wltr}.U.q] \wedge [s \Rightarrow \mathbf{wltr}.V.q] \rangle \\
\equiv & \{\text{induction hypothesis, twice}\} \\
& \langle \exists r, s : [r \vee s \equiv p] : r \overset{U}{\longmapsto} q \wedge s \overset{V}{\longmapsto} q \rangle \\
\equiv & \{\mathbf{(AxAlt)}\} \\
& p \overset{U+V}{\longmapsto} q
\end{array}
$$

case $W = U^*$:

$$
\begin{array}{ll}
& \mathbf{R}.U^*.p.q \quad \equiv \quad p \overset{U^*}{\longmapsto} q \\
\equiv & \{\text{definition of } \mathbf{R}, \mathbf{(AxStar)}\} \\
& [p \Rightarrow \mathbf{wltr}.U^*.q] \quad \equiv \\
& \langle \exists p' : [p \Rightarrow p'] : \langle \exists M : M \text{ is metric} : \\
& \qquad \langle \forall i : i \in \mathbf{Ord} : \\
& \qquad\quad p' \wedge M.i \overset{U}{\longmapsto} (p' \wedge \langle \exists j : j < i : M.j \rangle) \vee q \rangle\rangle \\
\equiv & \{\text{induction hypothesis}\} \\
& [p \Rightarrow \mathbf{wltr}.U^*.q] \quad \equiv \\
& \langle \exists p' : [p \Rightarrow p'] : \langle \exists M : M \text{ is metric} : \\
& \qquad \langle \forall i : i \in \mathbf{Ord} : \\
& \qquad\quad [(p' \wedge M.i) \Rightarrow \mathbf{wltr}.U.((p' \wedge \langle \exists j : j < i : M.j \rangle) \vee q)] \rangle\rangle\rangle
\end{array}
$$

In order to proceed with the proof we introduce some notation: first we use the abbreviation $E.p'.M.i$ for the innermost quantification term of the right-hand side of the last formula:

$$E.p'.M.i \;\equiv\; [(p' \wedge M.i) \Rightarrow \mathbf{wltr}.U.((p' \wedge \langle \exists j : j < i : M.j \rangle) \vee q)]$$

We also introduce the predicate transformer τ by

$$[\![\tau.X \;\equiv\; q \vee \mathbf{wltr}.U.X]\!]$$

Repeated application of τ is defined for any ordinal i in the usual way:

$$[\![\tau^0.X \;\equiv\; X]\!]$$

for step ordinal i: $[\![\tau^i.X \;\equiv\; \tau.(\tau^{i-1}.X)]\!]$

for limit ordinal i: $[\![\tau^i.X \;\equiv\; \langle \exists j : j < i : \tau^j.X\rangle]\!]$

Based on these definitions we state the following properties of τ:

τ is monotonic **(T0)**

$\mathbf{wltr}\,.U^*.q$ is the strongest solution of $X : [\![\tau.X \equiv X]\!]$ **(T1)**

$[\![\mathbf{wltr}\,.U^*.q \;\equiv\; \langle \exists i : i \in \mathbf{Ord} : \tau^i.\,\text{false}\rangle]\!]$ **(T2)**

$[\![\tau.\,\text{false} \;\equiv\; q]\!]$ **(T3)**

$\langle \forall i, j : i \leq j : [\![\tau^i.\,\text{false} \;\Rightarrow\; \tau^j.\,\text{false}]\!]\rangle$ **(T4)**

(T0) follows from $\mathbf{wltr}\,.U$ being monotonic, **(T1)** is a restatement of **(wltrStar)**, **(T2)** is a consequence of **(T1)** and the Knaster-Tarski Theorem, **(T3)** follows from **(wltrStrict)**, and **(T4)** finally follows from **(T0)** and **(T3)**.

We restate our proof obligation using the above definitions:

$$\begin{aligned}&[p \Rightarrow \mathbf{wltr}\,.U^*.q] \quad \equiv \quad (*)\\ &\langle \exists p' :: [p \Rightarrow p'] \wedge \langle \exists M : M \text{ is metric} :\\ &\qquad\qquad \langle \forall i : i \in \mathbf{Ord} : E.p'.M.i\rangle\rangle\rangle\end{aligned}$$

which we prove in the following by mutual implication.

Proof of **(*)**, $\Rightarrow$:

We choose as witnesses for p' and M the following:

$[p' \;\equiv\; \mathbf{wltr}\,.U^*.q]$

$[M.0 \;\equiv\; \neg\,\mathbf{wltr}\,.U^*.q]$

$\langle \forall i : i > 0 : [M.i \;\equiv\; \tau^i.\,\text{false} \;\wedge\; \neg\langle \exists j : j < i : \tau^j.\,\text{false}\rangle]\rangle$

In appendix D.1 we show that M is indeed a metric, and furthermore satisfies the following property for all ordinals i:

$[\langle \exists j : 0 < j \leq i : M.j\rangle \;\equiv\; \tau^i.\,\text{false}]$ **(M0)**

In order to prove the left-to-right implication of **(*)**, it suffices to show that for the choices for p' and M above:

$$\langle \forall i : i \in \mathbf{Ord} : E.p'.M.i \rangle$$

which we establish by transfinite induction over i. For $i = 0$ we observe

$$\begin{array}{ll}
 & E.p'.M.0 \\
\equiv & \{\text{definitions of } M.0,\ p' \text{ and } E\} \\
 & [\mathbf{wltr}.U^*.q \wedge \neg\,\mathbf{wltr}.U^*.q \ \Rightarrow \mathbf{wltr}.U.q] \\
\equiv & \{\text{predicate calculus}\} \\
 & \text{true}
\end{array}$$

For $i > 0$ we have

$$\begin{array}{ll}
 & E.p'.M.i \\
\equiv & \{\text{definitions of } M.i,\ p' \text{ and } E\} \\
 & [\mathbf{wltr}.U^*.q \wedge \tau^i.\text{false} \ \wedge \neg\langle \exists j : j < i : \tau^j.\text{false}\rangle \\
 & \Rightarrow \quad \mathbf{wltr}.U.((\mathbf{wltr}.U^*.q \wedge \langle \exists j : j < i : M.j\rangle) \vee q)] \\
\equiv & \{\text{from } \mathbf{(T2)}\text{: } [\tau^i.\text{false} \Rightarrow \mathbf{wltr}.U^*.q]\} \\
 & [\tau^i.\text{false} \ \wedge \neg\langle \exists j : j < i : \tau^j.\text{false}\rangle \\
 & \Rightarrow \quad \mathbf{wltr}.U.((\mathbf{wltr}.U^*.q \wedge \langle \exists j : j < i : M.j\rangle) \vee q)]
\end{array}$$

For a limit ordinal i the antecedent of the last formula is false, hence this last proof obligation is trivially satisfied.
For a step ordinal i we observe

$$\begin{array}{ll}
 & [\tau^i.\text{false} \ \wedge \neg\langle \exists j : j < i : \tau^j.\text{false}\rangle \\
 & \Rightarrow \quad \mathbf{wltr}.U.((\mathbf{wltr}.U^*.q \wedge \langle \exists j : j < i : M.j\rangle) \vee q)] \\
\Leftarrow & \{\text{left-hand side weakening, splitting the range}\} \\
 & [\tau^i.\text{false} \Rightarrow \mathbf{wltr}.U.((\mathbf{wltr}.U^*.q \wedge (M.0 \vee \langle \exists j : 0 < j < i : M.j\rangle)) \vee q)] \\
\equiv & \{\text{definition of } M.0, \text{ predicate calculus, } i \text{ is step ordinal}\} \\
 & [\tau^i.\text{false} \Rightarrow \mathbf{wltr}.U.((\mathbf{wltr}.U^*.q \wedge \langle \exists j : 0 < j \leq i-1 : M.j\rangle) \vee q)]
\end{array}$$

$\equiv$ {**(M0)**}

$[\tau^i.\,\text{false} \Rightarrow \mathbf{wltr}\,.U.((\mathbf{wltr}\,.U^*.q \wedge \tau^{i-1}.\,\text{false}) \vee q)]$

$\equiv$ {**(T2)**, predicate calculus}

$[\tau^i.\,\text{false} \Rightarrow \mathbf{wltr}\,.U.(\tau^{i-1}.\,\text{false} \ \vee q)]$

$\Leftarrow$ {monotonicity of $\mathbf{wltr}\,.U$, twice, predicate calculus}

$[\tau^i.\,\text{false} \Rightarrow \mathbf{wltr}\,.U.(\tau^{i-1}.\,\text{false}) \ \vee \ \mathbf{wltr}\,.U.q]$

$\Leftarrow$ {**(wrltWeaken)**, predicate calculus}

$[\tau^i.\,\text{false} \Rightarrow \mathbf{wltr}\,.U.(\tau^{i-1}.\,\text{false}) \ \vee \ q]$

$\equiv$ {definition of τ, predicate calculus}

true

Proof of **(*)**, $\Leftarrow$:

Let us abbreviate with B the following equation in p':

$$p' : ([p \Rightarrow p'] \ \wedge \ \langle \exists M : M \text{ is metric} : \langle \forall i : i \in \mathbf{Ord} : E.p'.M.i \rangle \rangle)$$

In order to establish the right-to-left implication of **(*)**, it clearly suffices to show that any solution p' of B also satisfies also satisfies

$$[p' \Rightarrow \mathbf{wltr}\,.U^*.q]$$

i.e., that $\mathbf{wltr}\,.U^*.q$ is the weakest solution of B. We therefore observe for any p' solving B, where M is some corresponding witness metric:

$[p' \Rightarrow \mathbf{wltr}\,.U^*.q]$

$\equiv$ {**(MetricExh)**, **(T2)** and **(T4)**}

$[p' \wedge \langle \exists i : i \in \mathbf{Ord} : M.i \rangle \Rightarrow \langle \exists i : i \in \mathbf{Ord} : \tau^{i+2}.\,\text{false} \rangle]$

$\Leftarrow$ {predicate calculus}

$\langle \forall i : i \in \mathbf{Ord} : [p' \wedge M.i \Rightarrow \tau^{i+2}.\,\text{false}] \rangle$

We establish this last proof obligation by transfinite induction over i. For $i = 0$ we observe:

$p' \wedge M.0$

$$\Rightarrow \quad \{p' \text{ solves } B, \text{ definition of } E\}$$
$$\mathbf{wltr}.U.q$$
$$\Rightarrow \quad \{\text{definition of } \tau\}$$
$$\tau.q$$
$$\equiv \quad \{\mathbf{(T3)}\}$$
$$\tau^2.\,\text{false}$$

For $i > 0$ we have:

$$p' \wedge M.i$$
$$\Rightarrow \quad \{p' \text{ solves } B, \text{ definition of } E\}$$
$$\mathbf{wltr}.U.((p' \wedge \langle \exists j : j < i : M.j \rangle) \vee q)$$
$$\equiv \quad \{\text{predicate calculus}\}$$
$$\mathbf{wltr}.U.(\langle \exists j : j < i : p' \wedge M.j \rangle \vee q)$$
$$\Rightarrow \quad \{\text{induction hypothesis, } \mathbf{wltr}.U \text{ is monotonic}\}$$
$$\mathbf{wltr}.U.(\langle \exists j : j < i : \tau^{j+2}.\,\text{false} \rangle \vee q)$$

For any limit ordinal i we observe

$$\mathbf{wltr}.U.(\langle \exists j : j < i : \tau^{j+2}.\,\text{false} \rangle \vee q)$$
$$\Rightarrow \quad \{i \text{ is limit ordinal}\}$$
$$\mathbf{wltr}.U.(\langle \exists j : j < i : \tau^{j}.\,\text{false} \rangle \vee q)$$
$$\equiv \quad \{\text{definition of } \tau^i \text{ for limit ordinal } i, \mathbf{(T3)}\}$$
$$\mathbf{wltr}.U.(\tau^i.\,\text{false} \vee \tau.\,\text{false})$$
$$\Rightarrow \quad \{\mathbf{(T4)}, \text{ predicate calculus}\}$$
$$\mathbf{wltr}.U.(\tau^i.\,\text{false}) \vee q$$
$$\Rightarrow \quad \{\text{definition of } \tau, \mathbf{(T4)}\}$$
$$\tau^{i+2}.\,\text{false}$$

and for any step ordinal i we observe

$$\mathbf{wltr}.U.(\langle \exists j : j < i : \tau^{j+2}.\,\text{false} \rangle \vee q)$$
$$\equiv \quad \{i \text{ is step ordinal}\}$$
$$\mathbf{wltr}.U.(\langle \exists j : j \leq i-1 : \tau^{j+2}.\,\text{false} \rangle \vee q)$$
$$\equiv \quad \{\mathbf{(T4)}, \mathbf{(T3)}\}$$

$$
\begin{array}{ll}
& \mathbf{wltr}.U.(\tau^{i+1}.\,\text{false} \;\vee\; \tau.\,\text{false}) \\
\Rightarrow & \{\textbf{(T4)}, \text{ predicate calculus}\} \\
& \mathbf{wltr}.U.(\tau^{i+1}.\,\text{false}) \vee q \\
\equiv & \{\text{definition of } \tau\} \\
& \tau^{i+2}.\,\text{false}
\end{array}
$$

which concludes the proof.

End of Proof.

As an immediate consequence of the above theorem we observe for all regular expressions W in $\mathcal{R}_F$ and all state predicates p and q on $\mathcal{P}_F$:

$$
\begin{array}{l}
\mathbf{wltr}.W.q \;\stackrel{W}{\longmapsto}\; q \\
p \;\stackrel{W}{\longmapsto}\; q \;\Rightarrow\; [p \Rightarrow \mathbf{wltr}.W.q]
\end{array}
$$

which together establish that $\mathbf{wltr}.W.q$ as defined by the equations **(wltrEps)**, **(wltrAct)**, **(wltrSeq)**, **(wltrAlt)**, and **(wltrStar)** is the unique weakest solution of the equation $p : (p \;\stackrel{W}{\longmapsto}\; q)$.

4.3.3 Some Derived Proof Rules for Generalized Leads-To

In this section we exhibit some useful laws that increase our repertoire of proof rules and enable us to establish more easily generalized leads-to properties of programs. In the following let F be any program.

Theorem 13 (Derived Proof Rules) *For any V, W in $\mathcal{R}_F$ and any state predicates p, p', q, q', and b in $\mathcal{P}_F$, for any set S and any mappings $f, g : S \to \mathcal{P}_F$:*

$$
\begin{array}{ll}
[p \Rightarrow q] \quad\Rightarrow\quad (p \;\stackrel{W}{\longmapsto}\; q) & \textbf{\textit{(Imply)}} \\
[p' \Rightarrow p] \;\wedge\; (p \;\stackrel{W}{\longmapsto}\; q) \quad\Rightarrow\quad (p' \;\stackrel{W}{\longmapsto}\; q) & \textbf{\textit{(LhsStr)}} \\
(p \;\stackrel{W}{\longmapsto}\; q) \;\wedge\; [q \Rightarrow q'] \quad\Rightarrow\quad (p \;\stackrel{W}{\longmapsto}\; q') & \textbf{\textit{(RhsWeak)}} \\
\langle \forall m : m \in S : f.m \;\stackrel{W}{\longmapsto}\; g.m \rangle \quad\Rightarrow & \textbf{\textit{(GenDisj)}} \\
\qquad (\langle \exists m : m \in S : f.m \rangle \;\stackrel{W}{\longmapsto}\; \langle \exists m : m \in S : g.m \rangle) &
\end{array}
$$

$$(p \overset{W}{\longmapsto} \text{false}) \quad \Rightarrow \quad [\neg p] \qquad \textbf{\textit{(Impossible)}}$$

$$(p \overset{V}{\longmapsto} q \vee b) \ \wedge \ (b \overset{W}{\longmapsto} r) \quad \Rightarrow \quad (p \overset{VW}{\longmapsto} q \vee r) \qquad \textbf{\textit{(Cancel)}}$$

(Imply) states that the generalized leads-to relation as a binary relation over predicates is weaker than implication. By **(LhsStr)** and **(RhsWeak)** the generalized leads-to relation is weakening in its right and strengthening in its left argument. Finally,**(GenDisj)**, **(Impossible)** and **(Cancel)** are generalizations of the general disjunction, impossibility and cancellation theorems of the ordinary leads-to relation (cf. [CM88]).

Proof of Theorem 13. All proof rules are established by using the fundamental connection between the generalized progress relation and the **wltr** predicate transformers (theorem 12) and properties of **wltr**:

For **(Imply)** we observe for all W, p and q:

$$\begin{array}{ll} & p \overset{W}{\longmapsto} q \\ \equiv & \{\text{theorem 12}\} \\ & [p \Rightarrow \textbf{wltr}.W.q] \\ \Leftarrow & \{\textbf{(wltrWeaken)}, \text{ predicate calculus}\} \\ & [p \Rightarrow q] \end{array}$$

We prove **(LhsStr)** and **(RhsWeak)** simultaneously by observing for all W, p, p', q and q':

$$\begin{array}{ll} & p' \overset{W}{\longmapsto} q' \\ \equiv & \{\text{theorem 12}\} \\ & [p' \Rightarrow \textbf{wltr}.W.q'] \\ \Leftarrow & \{\text{antecedent } [p' \Rightarrow p], \text{ predicate calculus}\} \\ & [p \Rightarrow \textbf{wltr}.W.q'] \\ \Leftarrow & \{\text{antecedent } [q \Rightarrow q'], \textbf{(wltrMon)}, \text{ predicate calculus}\} \\ & [p \Rightarrow \textbf{wltr}.W.q] \\ \equiv & \{\text{theorem 12}\} \\ & p \overset{W}{\longmapsto} q \end{array}$$

For **(GenDisj)** we observe for all W, S, f and g, where all quantifications are over elements in S:

$$
\begin{array}{ll}
& \langle \forall m :: f.m \ \overset{W}{\longmapsto}\ g.m \rangle \\
\equiv & \{\text{theorem 12}\} \\
& \langle \forall m :: [f.m \Rightarrow \mathbf{wltr}.W.(g.m)] \rangle \\
\Rightarrow & \{\text{predicate calculus}\} \\
& \langle \forall m :: [f.m \Rightarrow \langle \exists m :: \mathbf{wltr}.W.(g.m) \rangle] \rangle \\
\Rightarrow & \{\text{predicate calculus, } \mathbf{(wltrMon)}\} \\
& [\langle \exists m :: f.m \rangle \Rightarrow \mathbf{wltr}.W.\langle \exists m :: g.m \rangle] \\
\equiv & \{\text{theorem 12}\} \\
& \langle \exists m :: f.m \rangle \ \overset{W}{\longmapsto}\ \langle \exists m :: g.m \rangle
\end{array}
$$

For **(Impossible)** we observe for all W and p:

$$
\begin{array}{ll}
& p \ \overset{W}{\longmapsto}\ \text{false} \\
\equiv & \{\text{theorem 12}\} \\
& [p \Rightarrow \mathbf{wltr}.W.\text{false}] \\
\equiv & \{\mathbf{(wltrStrict)}\} \\
& [p \Rightarrow \text{false}] \\
\equiv & \{\text{predicate calculus}\} \\
& [\neg p]
\end{array}
$$

Finally, for **(Cancel)** we observe for all W, p, q, r, and b:

$$
\begin{array}{ll}
& b \ \overset{W}{\longmapsto}\ r \\
\equiv & \{\text{theorem 12, predicate calculus}\} \\
& [b \Rightarrow \mathbf{wltr}.W.r] \\
\Rightarrow & \{\text{predicate calculus}\} \\
& [q \vee b \ \Rightarrow\ q \vee \mathbf{wltr}.W.r] \\
\Rightarrow & \{\mathbf{(wltrWeaken)}, \mathbf{(wltrMon)}\} \\
& [q \vee b \ \Rightarrow\ \mathbf{wltr}.W.(q \vee r)] \\
\Rightarrow & \{\mathbf{(wltrMon)}, \mathbf{(wltrSeq)}\} \\
& [\mathbf{wltr}.V.(q \vee b) \Rightarrow\ \mathbf{wltr}.(VW).(q \vee r)]
\end{array}
$$

$\Rightarrow$ {from antecedent: $[p \Rightarrow \mathbf{wltr}.V.(q \vee b)]$}

$[p \Rightarrow \mathbf{wltr}.(VW).(q \vee r)]$

$\equiv$ {theorem 12}

$p \overset{VW}{\longmapsto} q \vee r$

End of Proof.

We conclude our listing of properties of the generalized leads-to relation by relating it to the ordinary leads-to relation of UNITY logic. The soundness and completeness result given below is an immediate corollary of theorems 9 and 12 and the following connection between the ordinary leads-to relation and the **wlt** predicate transformer (cf. [JKR89]):

$$[p \Rightarrow \mathbf{wlt}.q] \quad \equiv \quad p \mapsto q$$

Put together we obtain the following

Corollary 14 *For any W in $\mathcal{R}_F$, and state predicates p and q in $\mathcal{P}_F$:*

$$(p \overset{W}{\longmapsto} q) \quad \Rightarrow \quad (p \mapsto q) \qquad \textit{(Sound)}$$
$$(p \mapsto q) \quad \Rightarrow \quad \langle \exists W : W \in \mathcal{R}_F : p \overset{W}{\longmapsto} q \rangle \qquad \textit{(Compl)}$$

In summary, we see that proving $p \overset{W}{\longmapsto} q$ for some W establishes the ordinary $p \mapsto q$ as well, while conversely any ordinary progress property can be proved as a generalized one.

4.4 Progress and Regular Expressions

After having established properties of the predicate transformers $\mathbf{wltr}.W$ for each W in $\mathcal{R}_F$ in section 4.2, we now turn to the task of investigating the structure of the family of such predicate transformers, i.e., to the task of determining the relationship of predicate transformers $\mathbf{wltr}.U$ and $\mathbf{wltr}.V$ for different regular expressions U and V in $\mathcal{R}_F$.

It is our goal to design an equational theory for such predicate transformers that allows us to relate different regular expressions that may be used as hints in the verification of progress properties. Based on such a theory we are able to design methodologies for effectively using regular expressions in progress proofs, for instance by replacing certain regular expressions by others that allow for a more efficient mechanical verification, or by experimenting with different regular expressions in order to check different hypothesis about design knowledge, to gain a clearer understanding of how progress is achieved, and to help in program and specification debugging.

We will meet the goal of designing an equational theory for $\mathcal{R}_F$ by introducing the notion of a *progress algebra* which captures the essence of the algebraic structure of the **wltr** family of predicate transformers. We could have restricted ourselves to analyzing this family directly, but the introduction of a special algebra for doing so provides a layer of abstraction that makes it possible to separate algebraic issues from the details of the fixpoint characterizations of the predicate transformers, and to compare the proposed algebraic structure to other familiar structures like Kleene algebras [Koz90].

The characterization of the algebraic structure of **wltr** is done in two parts: first, in section 4.4.1, we define the notion of progress algebras and establish a few properties of such algebras. Then, in section 4.4.2, we demonstrate that the family **wltr** of predicate transformers can be considered a progress algebra, which establishes the algebraic characterization of **wltr**.

4.4.1 Progress Algebras

So far we have regarded $\mathcal{R}_F$ as a free algebra generated by the actions in $F.A$ and by the operators ε, $\cdot$ (sequencing), $+$ (alternation), and * (repetition). In the following we define a coarser algebraic structure that we call a *progress algebra* by presenting a list of equalities and equational implications which define a congruence relation on regular expressions and thereby an equivalence class structure.

For a given program F we refer to the resulting algebraic structure as $\mathcal{R}_F$ from now on and call it the *progress algebra* for program F. First we define the equational Horn theory for $\mathcal{R}_F$, then we show that $\mathcal{R}_F$ bears many similarities to the well known Kleene algebras and to the algebra of regular events. In the following we use the familiar formalism and terminology of Kleene algebras [Koz90] where appropriate.

We start with the definition of progress algebra in which the binary relation $\leq$ (pronounced *subsumed by*) is defined by $U \leq V \;\equiv\; U + V = V$.

Definition 5 (Progress Algebra) *A progress algebra $\mathcal{K}$ is the free algebra with binary operations $\cdot$ and $+$, unary operation *, and constant ε satisfying the following equations and equational implications for all U, V, and W in $\mathcal{K}$:*

$$U + (V + W) \;=\; (U + V) + W \qquad \textbf{\textit{(PrAlg0)}}$$

$$U + V \;=\; V + U \qquad \textbf{\textit{(PrAlg1)}}$$

$$W + W \;=\; W \qquad \textbf{\textit{(PrAlg2)}}$$

$$U(VW) \;=\; (UV)W \qquad \textbf{\textit{(PrAlg3)}}$$

$$\varepsilon W \;=\; W \qquad \textbf{\textit{(PrAlg4)}}$$

$$W\varepsilon \;=\; W \qquad \textbf{\textit{(PrAlg5)}}$$

$$UV + UW \;\leq\; U(V + W) \qquad \textbf{\textit{(PrAlg6)}}$$

$$(U + V)W \;=\; UW + VW \qquad \textbf{\textit{(PrAlg7)}}$$

$$\varepsilon \;\leq\; W \qquad \textbf{\textit{(PrAlg8)}}$$

$$\varepsilon + WW^* \;\leq\; W^* \qquad \textbf{\textit{(PrAlg9)}}$$

$$\varepsilon + W^*W \;\leq\; W^* \qquad \textbf{\textit{(PrAlg10)}}$$

$$UW \;\leq\; W \quad \Rightarrow \quad U^*W \;\leq\; W \qquad \textbf{\textit{(PrAlg11)}}$$

$$WU \;\leq\; W \quad \Rightarrow \quad WU^* \;\leq\; W \qquad \textbf{\textit{(PrAlg12)}}$$

*A progress algebra satisfying **(PrAlg11)** but not necessarily **(PrAlg12)** is called a* right-handed *progress algebra, and a progress algebra satisfying **(PrAlg12)** but not* *necessarily* ***(PrAlg11)*** *is called a* left-handed *progress algebra.*

Axioms **(PrAlg0)**, **(PrAlg1)** and **(PrAlg2)** characterize the $+$ operator as associative, commutative, and idempotent. The $\cdot$ operator is associative by **(PrAlg3)** and has ε as a unit by **(PrAlg4)** and **(PrAlg5)**. Axioms **(PrAlg6)** and **(PrAlg7)** define how $\cdot$ and $+$ interact, namely that $\cdot$ distributes over $+$ on the right, but not quite on the left. The next axiom **(PrAlg8)** identifies ε as a minimum element. Finally, the properties of the * operator are characterized by axioms **(PrAlg9)** through **(PrAlg12)**.

The motivation for defining an algebraic structure by the above equations and equational implications is twofold: first, we want to stay as close as possible to the axioms of Kleene algebras ([Koz90]), which define a very important and familiar structure that arises in many different areas of computer science including automata theory and program semantics. Doing so allows us to reuse some theorems of Kleene algebras and regular language theory. Second, we want to capture the equational structure of the family **wltr** of predicate transformers as an algebra. This makes it possible to characterize the essence of the structure of the **wltr** predicate transformers in an abstract way.

Comparing progress algebras with Kleene algebras we notice three major differences:

1. Progress algebras lack the equivalent of the $\emptyset$ constant of Kleene algebras. One could consider introducing such a constant by defining $[\![\mathbf{wltr}.\emptyset.q \equiv \text{false}]\!]$, which would actually satisfy the Kleene axioms referring to $\emptyset$. Since such a regular expression does not have a counterpart in either the operational model or the deductive system, we omit it from further consideration.

2. A progress algebra does not have to satisfy the left distributivity of $\cdot$ over $+$. Only the weaker inequality **(PrAlg6)** is required instead.

3. On the other hand any progress algebra satisfies an additional axiom not present in Kleene algebras **(PrAlg8)**.

The first two differences account for the fact, that progress algebras do not have a ring structure. Whereas the omission of a unit element for $+$ is somewhat arbitrary, the lack of left distributivity of $\cdot$ over $+$ is essential. This lack is a well known property of process algebras [Mil89], which, however, are substantially different due to their lack of any law corresponding to **(PrAlg8)**.

In the following we list some important properties of progress algebras that can be derived from the above axioms. The proofs can be found in appendix D.2. First we have the following properties of the subsumption relation $\leq$:

Lemma 15 *In any left-handed or right-handed progress algebra $\mathcal{K}$, the subsumption relation $\leq$ defined by $U \leq V \;\equiv\; U + V = V$ is a partial order. Moreover the sequencing, alternation, and repetition operators are monotonic with respect to $\leq$, i.e., for all U, V, U' and V' in $\mathcal{K}$ with $U \leq V$ and $U' \leq V'$:*

$$UU' \;\leq\; VV' \qquad \textit{\textbf{(PrAlgSeq)}}$$

$$U + U' \;\leq\; V + V' \qquad \textit{\textbf{(PrAlgAlt)}}$$

$$U^* \;\leq\; V^* \qquad \textit{\textbf{(PrAlgStar)}}$$

In spite of the differences from Kleene algebras many theorems of Kleene algebras also hold for progress algebras as stated in the following lemma:

Lemma 16 *In any left-handed or right-handed progress algebra $\mathcal{K}$ for all U, V, and W in $\mathcal{K}$ the following laws hold:*

$$\varepsilon + WW^* \;=\; W^* \qquad \textit{\textbf{(PrAlg13)}}$$

$$\varepsilon + W^*W \;=\; W^* \qquad \textit{\textbf{(PrAlg14)}}$$

$$W^*W^* \;=\; W^* \qquad \textit{\textbf{(PrAlg15)}}$$

$$(W^*)^* \;=\; W^* \qquad \textit{\textbf{(PrAlg16)}}$$

Furthermore, for right-handed $\mathcal{K}$ and all U, V, and W in $\mathcal{K}$:

$$V + UW \leq W \;\Rightarrow\; U^*V \leq W \tag{PrAlg17}$$

and for left-handed $\mathcal{K}$ and all U, V, and W in $\mathcal{K}$:

$$V + WU \leq W \;\Rightarrow\; VU^* \leq W \tag{PrAlg18}$$

(PrAlg13) through **(PrAlg16)** correspond to well-known properties of the * operator of Kleene algebras. Furthermore, as in Kleene algebras [Pra88], **(PrAlg17)** is equivalent to **(PrAlg11)**, and **(PrAlg18)** is equivalent to **(PrAlg12)**.

Finally there are some interesting properties of progress algebras that are not generally true in Kleene algebras:

Lemma 17 *In any left-handed or right-handed progress algebra $\mathcal{K}$, for all $n \in \mathbf{N}$ with $n > 0$ and sequences W in $\mathbf{Z}_n \to \mathcal{K}$, for all permutations π of $\mathbf{Z}_n$, and for all U and V in $\mathcal{K}$ the following laws hold:*

$$UV \;\leq\; \varepsilon \quad\Rightarrow\quad U \;=\; \varepsilon \tag{PrAlg19}$$

$$UV \;\leq\; \varepsilon \quad\Rightarrow\quad V \;=\; \varepsilon \tag{PrAlg20}$$

$$U + V \;\leq\; \varepsilon \quad\Rightarrow\quad U \;=\; \varepsilon \tag{PrAlg21}$$

$$U^* \;\leq\; \varepsilon \quad\equiv\quad U \;=\; \varepsilon \tag{PrAlg22}$$

$$UU^* \;=\; U^* \tag{PrAlg23}$$

$$U^*U \;=\; U^* \tag{PrAlg24}$$

$$\langle +U : W : U \rangle \;\leq\; \langle \cdot U : W \circ \pi : U \rangle \tag{PrAlg25}$$

$$\langle +U : W : U \rangle^* \;=\; \langle \cdot U : W \circ \pi : U \rangle^* \tag{PrAlg26}$$

(PrAlg19) through **(PrAlg22)** characterize ε as an irreducible element, **(PrAlg23)** and **(PrAlg24)** show that arbitrary and positive repetitions are equivalent. **(PrAlg25)** states that the alternation of regular expressions is subsumed by any permuted sequencing of those expressions, and **(PrAlg26)** asserts that the repetitions of an alternation of regular expressions and of any permuted sequencing of those expressions are the same.

In order to show that the equational theory of progress algebras is consistent and to illustrate the connection between Kleene algebras and progress algebras, consider $\mathbf{Reg}_\Sigma$, the *algebra of regular events* over the non-empty alphabet Σ, the elements of which are regular languages over Σ. This is an important instance of a Kleene algebra that we relate to a progress algebra in the following.

Clearly, $\mathbf{Reg}_\Sigma$ is not a progress algebra, since **(PrAlg8)** is not satisfied even if W is restricted to denote only non-empty languages: for any α in Σ, $\varepsilon \leq \alpha$ does not hold, since $\mathcal{L}.(\varepsilon + \alpha) = \{\varepsilon, \alpha\} \neq \{\alpha\} = \mathcal{L}.(\alpha)$. Conversely, a progress algebra can certainly not be embedded in a Kleene algebra because of the lack of equality in **(PrAlg6)**. However, if we interpret $\leq$ as the subsumption order on strings (cf. section 2.1.6), and define equality by $U = V \;\equiv\; U \leq V \wedge V \leq U$, it is easy to check that the resulting structure meets all progress algebra axioms (see appendix D.2.4 for details).

4.4.2 $\mathcal{R}_F$ as Progress Algebra

In the previous section we have introduced the notion of progress algebras and have exhibited many properties of them. It is our goal now to show that the family **wltr** of predicate transformers can be regarded as a progress algebra. This allows us to characterize the algebraic structure $\mathcal{R}_F$ of the **wltr** predicate transformers for any program F.

In order to show that **wltr** is a progress algebra, we have to define the equational theory of **wltr**, to relate the operators $\cdot$, $+$, *, and the constant ε of $\mathcal{R}_F$ to operations on predicate transformers, and finally to show that the equations and equational implications defining progress algebras are met by **wltr**.

The equational theory and the algebraic structure of **wltr** are defined as expected: any W in $\mathcal{R}_F$ denotes the predicate transformer $\mathbf{wltr}.W$ over $\mathcal{P}_S$; the meaning of the constant ε is given by **(wltrEps)** as the identity transformer; the meaning of the operators $\cdot$, $+$, and * is given by **(wltrSeq)**, **(wltrAlt)**, and **(wltrStar)** as functional composition of predicate transformers, disjunction of predicate transformers, and a least fixpoint construction respectively; the meaning of the basic elements α in $F.A$ is given by **(wltrAct)** as the $\mathbf{wltr}.\alpha$ predicate transformer. Finally, equality of regular expressions over $F.A$ (written as $=_F$) is defined as equivalence of the corresponding predicate transformers, i.e., for all U, V in $\mathcal{R}_F$:

$$U =_F V \quad \equiv \quad [\![\mathbf{wltr}.U \equiv \mathbf{wltr}.V]\!] \ .$$

The induced subsumption relation $\leq_F$ on $\mathcal{R}_F$ is then given by

$$U \leq_F V \quad \equiv \quad U + V =_F V \ .$$

It follows that for all U and V in $\mathcal{R}_F$:

$$\begin{array}{ll} & U \leq_F V \\ \equiv & \{\text{definition of } \leq_F\} \\ & U + V =_F V \\ \equiv & \{\textbf{(wltrAlt)}, \text{ definition of } =_F\} \\ & [\![\mathbf{wltr}.U \vee \mathbf{wltr}.V \ \equiv \ \mathbf{wltr}.V]\!] \\ \equiv & \{\text{predicate calculus}\} \\ & [\![\mathbf{wltr}.U \ \Rightarrow \ \mathbf{wltr}.V]\!] \end{array}$$

In other words, the subsumption relation $\leq_F$ on $\mathcal{R}_F$ is exactly the implication of the corresponding predicate transformers. Based on this interpretation we can characterize the algebraic structure of $\mathcal{R}_F$ as follows:

Theorem 18 *For any program F, the algebra $\mathcal{R}_F$ is a right-handed progress algebra.*

Proof . We need to show that the progress algebra axioms **(PrAlg0)** through **(PrAlg11)** are satisfied by $\mathcal{R}_F$.

(PrAlg0), **(PrAlg1)**, and **(PrAlg2)** follow by virtue of **(wltrAlt)** directly from the associativity, commutativity, and idempotency of disjunction. Similarly,

(PrAlg3) follows by virtue of **(wltrSeq)** from the associativity of functional composition.

From **(wltrEps)** we know that $\mathbf{wltr}.\varepsilon$ is the identity transformer, which is the left and right identity element of functional composition. This establishes (using **(wltrSeq)**) both **(PrAlg4)** and **(PrAlg5)**.

For **(PrAlg6)** we observe for all q in $\mathcal{P}_F$:

$$\begin{array}{cl}
 & \mathbf{wltr}.U(V+W).q \\
\equiv & \{\textbf{(wltrSeq)}, \textbf{(wltrAlt)}\} \\
 & \mathbf{wltr}.U.(\mathbf{wltr}.V.q \vee \mathbf{wltr}.W.q) \\
\Leftarrow & \{\mathbf{wltr}.U \text{ is monotonic}\} \\
 & \mathbf{wltr}.U.(\mathbf{wltr}.V.q) \vee \mathbf{wltr}.U.(\mathbf{wltr}.W.q) \\
\equiv & \{\textbf{(wltrSeq)}, \textbf{(wltrAlt)}\} \\
 & \mathbf{wltr}.(UV+UW).q
\end{array}$$

Similarly, for **(PrAlg7)** we observe for all q in $\mathcal{P}_F$:

$$\begin{array}{cl}
 & \mathbf{wltr}.(U+V)W.q \\
\equiv & \{\textbf{(wltrSeq)}, \textbf{(wltrAlt)}\} \\
 & \mathbf{wltr}.U.(\mathbf{wltr}.W.q) \vee \mathbf{wltr}.V.(\mathbf{wltr}.W.q) \\
\equiv & \{\textbf{(wltrSeq)}, \textbf{(wltrAlt)}\} \\
 & \mathbf{wltr}.(UW+VW).q
\end{array}$$

(PrAlg8) follows directly from **(wltrEps)** and **(wltrWeaken)**. For **(PrAlg9)** we see by virtue of **(wltrStar)** that $\mathbf{wltr}.W^*.q$ is a solution of the equation $X : [\![q \vee \mathbf{wltr}.W.X]\!]$, i.e., it satisfies

$$[\![\mathbf{wltr}.W^*.q \;\equiv\; q \vee \mathbf{wltr}.W.(\mathbf{wltr}.W^*.q)]\!]. \qquad \textbf{(P0)}$$

With this we observe for all q in $\mathcal{P}_F$:

$$\begin{array}{cl}
 & \mathbf{wltr}.(\varepsilon + WW^*).q \\
\equiv & \{\textbf{(wltrAlt)}, \textbf{(wltrEps)}, \textbf{(wltrSeq)}\} \\
 & q \vee \mathbf{wltr}.W.(\mathbf{wltr}.W^*.q)
\end{array}$$

$$\equiv \quad \{\textbf{(P0)}\}$$
$$\mathbf{wltr}.W^*.q$$

The proof of **(PrAlg10)** is slightly more difficult. We first observe for all q in $\mathcal{P}_F$:

$$\mathbf{wltr}.(\varepsilon + W^*W).q$$
$$\equiv \quad \{\textbf{(wltrAlt)}, \textbf{(wltrEps)}\}$$
$$q \vee \mathbf{wltr}.W^*W.q$$
$$\equiv \quad \{[\![q \Rightarrow \mathbf{wltr}.W^*W.q]\!] \text{ from } \textbf{(wltrWeaken)}\}$$
$$\mathbf{wltr}.W^*W.q$$
$$\equiv \quad \{\textbf{(wltrSeq)}\}$$
$$\mathbf{wltr}.W^*.(\mathbf{wltr}.W.q)$$

thus leaving us with the proof obligation

$$[\![\mathbf{wltr}.W^*.(\mathbf{wltr}.W.q) \;\equiv\; \mathbf{wltr}.W^*.q]\!], \qquad \textbf{(P1)}$$

which we prove by mutual implication: the implication from right to left is an immediate consequence of **(wltrWeaken)** and the monotonicity of $\mathbf{wltr}.W^*$. For the other direction we introduce two predicate transformers f and g, that characterize, by virtue of **(wltrStar)**, the left-hand and right-hand side of **(P1)** respectively:

$$[\![f.X \;\equiv\; q \vee \mathbf{wltr}.W.X]\!]$$
$$[\![g.X \;\equiv\; \mathbf{wltr}.W.q \vee \mathbf{wltr}.W.X]\!]$$

Clearly, both f and g are monotonic. Furthermore $\mathbf{wltr}.W^*.q$ is the strongest fixpoint of f, and $\mathbf{wltr}.W^*.(\mathbf{wltr}.W.q)$ is the strongest fixpoint of g. Since both sides of **(P1)** are characterized as strongest fixpoints of some predicate transformers, it seems reasonable to attempt to conduct the proof by finding an *intermediate* predicate characterized as fixpoint of some other monotonic predicate transformer h:

$$[\![\mathbf{wltr}.W^*.(\mathbf{wltr}.W.q) \;\Rightarrow\; \mathbf{wltr}.W^*.q]\!]$$

$$\Leftarrow \quad \{\textbf{(wltrStar)}, \text{ properties of suitable } h\}$$
$$[\![\langle \mu Z :: g.Z\rangle \;\Rightarrow\; \langle \mu Z :: h.Z\rangle]\!] \;\wedge\; [\![\langle \mu Z :: h.Z\rangle \;\Rightarrow\; \langle \mu Z :: f.Z\rangle]\!]$$

For h to satisfy the first conjunct it is sufficient, by theorem 3, to require that $[\![g \Rightarrow h]\!]$. Since $\langle \mu Z :: h.Z\rangle$ is the strongest fixpoint of h, the second conjunct is satisfied by h if $\langle \mu Z :: f.Z\rangle$ is a fixpoint of h. Formally we hence require:

$$[\![g \Rightarrow h]\!]$$
$$[\![h.\langle \mu Z :: f.Z\rangle \;\equiv \langle \mu Z :: f.Z\rangle]\!].$$

Any iterated composition of f satisfies the second condition. Unfortunately $[\![g \Leftarrow f]\!]$ holds, but since f is weakening, we try $f \circ f$ for h by observe for all X in $\mathcal{P}_F$:

$$\begin{array}{ll}
 & g.X \\
\equiv & \{\text{definition of } g\} \\
 & \mathbf{wltr}.W.q \vee \mathbf{wltr}.W.X \\
\equiv & \{\textbf{(wltrWeaken)}, \text{ predicate calculus}\} \\
 & q \vee \mathbf{wltr}.W.q \vee \mathbf{wltr}.W.X \\
\Rightarrow & \{\mathbf{wltr}.W \text{ is monotonic}\} \\
 & q \vee \mathbf{wltr}.W.(q \vee X) \\
\Rightarrow & \{\mathbf{wltr}.W \text{ is monotonic, } \textbf{(wltrWeaken)}\} \\
 & q \vee \mathbf{wltr}.W.(q \vee \mathbf{wltr}.W.X) \\
\equiv & \{\text{definition of } f, \text{ twice}\} \\
 & f.(f.X)
\end{array}$$

Hence, choosing $f \circ f$, which is certainly monotonic, for h completes the proof of **(PrAlg10)**.

We finish the proof of theorem 18 by observing that the antecedent of **(PrAlg11)** is

$$[\![\mathbf{wltr}.U.(\mathbf{wltr}.W.q) \;\Rightarrow\; \mathbf{wltr}.W.q]\!], \qquad \textbf{(P2)}$$

whereas the conclusion has the form

$$[\![\mathbf{wltr}.U^*.(\mathbf{wltr}.W.q) \;\Rightarrow\; \mathbf{wltr}.W.q]\!].$$

Since $\mathbf{wltr}.U^*.(\mathbf{wltr}.W.q)$ is, by **(wltrStar)**, the strongest solution of the equation $X : [\![X \;\equiv\; \mathbf{wltr}.W.q \vee \mathbf{wltr}.U.X]\!]$, we have

$$\begin{array}{l} \langle \forall X :: [\![X \;\equiv\; \mathbf{wltr}.W.q \vee \mathbf{wltr}.U.X]\!] \;\Rightarrow \\ \qquad\qquad [\![\mathbf{wltr}.U^*.(\mathbf{wltr}.W.q) \Rightarrow X]\!]\rangle. \end{array} \qquad \textbf{(P3)}$$

We conclude the proof of **(PrAlg11)** by observing for all q in $\mathcal{P}_F$:

$$\begin{array}{ll} & [\![\mathbf{wltr}.U^*.(\mathbf{wltr}.W.q) \;\Rightarrow\; \mathbf{wltr}.W.q]\!] \\ \Leftarrow & \{\textbf{(P3)} \text{ with } X := \mathbf{wltr}.W.q\} \\ & [\![\mathbf{wltr}.W.q \;\equiv\; \mathbf{wltr}.W.q \vee \mathbf{wltr}.U.(\mathbf{wltr}.W.q)]\!] \\ \equiv & \{\text{predicate calculus, } \textbf{(P2)}\} \\ & \text{true} \end{array}$$

End of Proof.

Two remarks about the axioms **(PrAlg6)** and **(PrAlg12)** are in order. First, we note that **(PrAlg6)** cannot be strengthened to equality. By examining the proof for **(PrAlg6)** above we see that for equality we need to prove

$$[\![\mathbf{wltr}.U.(\mathbf{wltr}.V.q \vee \mathbf{wltr}.W.q) \;\equiv\; \mathbf{wltr}.U.(\mathbf{wltr}.V.q) \vee \mathbf{wltr}.U.(\mathbf{wltr}.W.q)]\!]$$

Clearly, we cannot expect this to hold in general, because from lemma 8 we know that $\mathbf{wltr}.U$ is not finitely disjunctive. In fact, the program *FiniteDisjunctivity* of section 4.2.2 serves as a counter example: with $U := \alpha$, $V := \beta$, $W := \gamma$, and $q := (n = 4)$, we have $[\![\mathbf{wltr}.\beta.(n = 4) \equiv n = 2 \vee n = 4]\!]$, $[\![\mathbf{wltr}.\gamma.(n = 4) \equiv n = 3 \vee n = 4]\!]$ and hence by virtue of the calculations in section 4.2.2

$$[\![\mathbf{wltr}.\alpha.(\mathbf{wltr}.\beta.(n = 4) \vee \mathbf{wltr}.\gamma.(n = 4)) \;\equiv\; \text{true}]\!]$$

whereas

$$[\![\mathbf{wltr}.\alpha.(\mathbf{wltr}.\beta.(n=4)) \vee \mathbf{wltr}.\alpha.(\mathbf{wltr}.\gamma.(n=4)) \;\equiv\; n \neq 0]\!]$$

Next, we show that $\mathcal{R}_F$ is not left-handed, i.e., that **(PrAlg12)** is not satisfied. An attempt at a proof of **(PrAlg12)** by transfinite induction over the ordinals up to the ordinal i for which $[\![\mathbf{wltr}.U^*.q \;\equiv\; \tau^i.\,\mathrm{false}]\!]$, where $[\![\tau.X \equiv q \vee \mathbf{wltr}.U.X]\!]$, fails because the inductive step for limit ordinals seems to require $\mathbf{wltr}.W$ to be or-continuous, which it is not in general by theorem 8.

In the following we construct a counterexample from the failed proof. Such a counterexample can be derived from two regular expressions U and V satisfying

$$\neg[\![\mathbf{wltr}.U^* \Rightarrow \langle \exists i : i < \omega : \mathbf{wltr}.U^i \rangle]\!], \tag{O0}$$

i.e., $\mathbf{wltr}.U^*$ is not stronger or equal to the ω - disjunction of all $\mathbf{wltr}.U^i$, but also

$$[\![\mathbf{wltr}.V \equiv \langle \exists i : i < \omega : \mathbf{wltr}.U^i \rangle]\!], \tag{O1}$$

i.e., the ω-disjunction of all $\mathbf{wltr}.U^i$ is expressible[5] in $\mathcal{R}_F$.

We show that such U and V constitute a counterexample by establishing that the antecedent of **(PrAlg12)**, $[\![\mathbf{wltr}.VU \;\Rightarrow\; \mathbf{wltr}.V]\!]$, holds, but that the conclusion $[\![\mathbf{wltr}.VU^* \Rightarrow \mathbf{wltr}.V]\!]$ is not satisfied. For the antecedent we have for all q in $\mathcal{P}_F$

$$\begin{array}{cl}
 & \mathbf{wltr}.VU.q \\
\equiv & \{\textbf{(wltrSeq)}, \textbf{(O1)}\} \\
 & \langle \exists i : i < \omega : \mathbf{wltr}.U^i.(\mathbf{wltr}.U.q) \rangle \\
\equiv & \{\textbf{(wltrWeaken)}, \text{predicate calculus}\} \\
 & \langle \exists i : i < \omega : \mathbf{wltr}.U^i.q \rangle \\
\equiv & \{\textbf{(O1)}\} \\
 & \mathbf{wltr}.V.q
\end{array}$$

A straightforward induction using **(P0)**, **(wltrWeaken)**, and **(wltrSeq)** allows us

[5] We call a predicate transformer τ expressible in $\mathcal{R}_F$ if and only if there is a regular expression V in $\mathcal{R}_F$ such that $[\![\mathbf{wltr}.V \equiv \tau]\!]$.

to establish that $[\![\mathbf{wltr}.U^* \equiv \mathbf{wltr}.U^iU^*]\!]$ for all i in $\mathbf{N}$. For the conclusion we can, therefore, observe for all q in $\mathcal{P}_F$

$$
\begin{array}{ll}
& \mathbf{wltr}.VU^*.q \\
\equiv & \{\mathbf{(wltrSeq)}, \mathbf{(O1)}\} \\
& \langle \exists i : i < \omega : \mathbf{wltr}.U^i.(\mathbf{wltr}.U^*.q)\rangle \\
\equiv & \{\mathbf{(wltrSeq)}, \text{observation above}\} \\
& \langle \exists i : i < \omega : \mathbf{wltr}.U^*.q\rangle \\
\equiv & \{\text{predicate calculus}\} \\
& \mathbf{wltr}.U^*.q
\end{array}
$$

which by **(O0)** and **(O1)** does not imply $\mathbf{wltr}.V.q$.

The following example due to Cohen [Coh96] exhibits such regular expressions U and V, thereby establishing that $\mathcal{R}_F$ is not left-handed. Consider the program *LeftHanded* given by

program *LeftHanded*
  declare
    var n : *natural*
    var b : *boolean*
  assign

$[\alpha]$	$n := n+1$	**if** $\neg b$			
$[\beta]$	$n, b := n-1, \text{true}$	**if** $b \wedge n > 0$	$\sim$	n, true	**if** $\neg b \vee n = 0$
$[\gamma]$	$n, b := 0, \text{true}$	**if** b	$\sim$	n, true	**if** $\neg b$

end

For this program we observe that properties **(O0)** and **(O1)** are satisfied for $U := \beta$ and $V := \gamma$. It can be checked easily that $[\![\mathbf{wltr}.\beta^i.(n = 0) \equiv (n = 0) \vee (b \wedge (n \leq i))]\!]$ for any i in $\mathbf{N}$, $[\![\langle \exists i : i < \omega : \mathbf{wltr}.\beta^i.(n = 0)\rangle \equiv n = 0 \vee b]\!]$, and $[\![\mathbf{wltr}.\gamma.(n = 0) \equiv n = 0 \vee b]\!]$, but that $[\![\mathbf{wltr}.\beta^*.(n = 0) \equiv \text{true}]\!]$.

We conclude our exploration of the algebraic structure of $\mathcal{R}_F$ by combining our results into the main theorem about relating the subsumption relation on progress algebras to implication of the corresponding **wltr** predicate transformers.

From the characterization of $\leq_F$ we recall that that for all U and V in $\mathcal{R}_F$

$$U \leq_F V \;\equiv\; [\![\mathbf{wltr}\,.U \Rightarrow \mathbf{wltr}\,.V]\!]$$

Since, by theorem 18, the family **wltr** of predicate transformers has the structure of a progress algebra, we also know that

$$U \leq V \;\Rightarrow\; U \leq_F V.$$

where the first subsumption is the relation provable in progress algebras, and the second one is the semantic subsumption of elements of $\mathcal{R}_F$. Combining these results we obtain the following:

Theorem 19 *For any program F,* **wltr** *is monotonic with respect to $\leq$ in its first argument, i.e., for all U and V in $\mathcal{R}_F$:*

$$U \leq V \;\Rightarrow\; [\![\mathbf{wltr}\,.U \Rightarrow \mathbf{wltr}\,.V]\!]$$

An immediate consequence of the above theorem and of theorem 12 is the following corollary, which relates the subsumption of regular expressions to the generalized leads-to relation:

Corollary 20 *For any program F and regular expressions U and V in $\mathcal{R}_F$ with $U \leq V$:*

$$(p \overset{U}{\longmapsto} q) \quad\Rightarrow\quad (p \overset{V}{\longmapsto} q).$$

We conclude this section with a brief discussion of the question of how well the **wltr** family of predicate transformers can be characterized algebraically. Theorem 19 establishes that for any given program the implication ordering on the transformers is at least as weak as the subsumption order on the corresponding regular expressions. It is obvious that an exact characterization of the implication order for *individual* programs cannot be achieved by any order on the progress algebra $\mathcal{R}_F$: for a simple program with two (syntactically) identical actions α and β, the predicate transformers $\mathbf{wltr}.\alpha$ and $\mathbf{wltr}.\beta$ are certainly identical, whereas the two regular expressions α and β cannot be related in any progress algebra.

Instead of considering individual programs we might, however, ask about the algebraic structure common to all programs sharing the same action alphabet. More precisely, let A be a finite set of actions, and let $\mathbf{Reg}.A$ be the progress algebra over A. For any regular expressions U and V in $\mathcal{R}.A$ we know that the axiom system for the progress algebra (definition 5) is sound, i.e., that from $U = V$ in $\mathcal{R}.A$ we can conclude $[\![\mathbf{wltr}.U \equiv \mathbf{wltr}.V]\!]$ for any program F with $F.A \supseteq A$. This follows form theorem 19 and the fact that $\mathcal{R}.A$ is a sub-algebra of $\mathcal{R}_F$.

The converse of this observation amounts to the formulation of a *completeness* property of the progress algebra axioms, which can be stated as follows:

$$\langle \forall U, V : U, V \in \mathcal{R}.A : \langle \forall F : F.A \supseteq A : [\![\mathbf{wltr}.U \equiv \mathbf{wltr}.V]\!]\rangle \Rightarrow U = V \rangle$$

It asserts that whenever the predicate transformers corresponding to two regular expressions U and V of some progress algebra $\mathcal{R}.A$ are equivalent *for all programs* F with suitable action set, U and V are provably equivalent in $\mathbf{Reg}.A$. We pose this completeness statement of progress algebras as an open question and remark that a positive answer would establish that the subsumption relation of progress algebras characterizes the *uniform* algebraic structure of the $\mathbf{wltr}$ predicate transformers exactly.

4.5 Progress by Actions

So far we have characterized generalized progress properties by predicate transformers and by a deductive proof system, but are still lacking a formal operational semantics. As mentioned earlier, the main motivation is to provide a link between generalized progress properties characterizing how progress is achieved for a given program, and executions of the program as a sequence of program states and actions, thereby allowing the designer to formulate some operational design knowledge in terms of regular expression hints. In order to allow this link to be exploited during verification, it is important that the operational semantics be as simple and intuitive as possible, and be easily matched with an understanding of possible program executions.

We base our presentation of an operational semantics for generalized progress properties on a certain notion of *games*, inspired by [Dij95], where games are used as a model for the DUALITY calculus. A game takes place between two players, called the *progressor* and the *opponent*, engaging in a series of game rounds. The sequence of the rounds is governed by a strategy chosen by the progressor from a set of possible strategies determined by a regular expression characterizing the game. In a game corresponding to the property $p \overset{W}{\longmapsto} q$, the progressor tries to reach a state satisfying q whenever some state satisfying p has been reached previously, whereas the opponent attempts to prevent this from happening. A program satisfies a property $p \overset{W}{\longmapsto} q$ if the progressor has a winning strategy for reaching q states from p states.

The remainder of this section is organized as follows: in section 4.5.1 we formalize the notions of games and strategies upon which our operational semantics is based. In section 4.5.2 we state our definition of the operational semantics and illustrate it with a few simple examples. Finally, in section 4.5.3 we relate the operational semantics to the deductive system by establishing a soundness and a completeness result.

4.5.1 Games and Strategies

In order to give an operational semantics for the generalized progress properties, we need to formalize the notions of games and strategies for a program F.

A *round* of F is a non-empty, finite run of F, i.e., an element of $(F.A)^+$. If for some α in $F.A$ a round r ends with α, i.e., if $r = x\alpha$ for some $x \in (F.A)^*$, we call r an α-round. A *game* of F is a (finite or infinite) sequence of rounds, i.e., an element of $((F.A)^+)^\infty$. In the following, we denote this set of games of program F by $\mathcal{G}_F$. The concatenation operation $+\!\!+$ on the elements of $\mathcal{G}_F$ is as defined in section 2.1.1.

For every game g we denote by $\bar{g}$ the run obtained by concatenating all rounds of g in order. Formally,

$$\begin{array}{lcll} \bar{\langle\rangle} & = & \langle\rangle & \\ \bar{g} & = & g.0 +\!\!+ \overline{\mathbf{tail}.g} & \text{if} \quad |g| > 0 \end{array}$$

Note that $\bar{g}$ is well defined also for infinite g (cf. [Sto81, Bro93]).

Next we will define the notion of a *strategy* of F for regular expressions W in $\mathcal{R}_F$. The *algebra of strategies* $\mathbf{Str}.A.P$ for a given sort A of actions and a sort P of state predicates, is the free sorted algebra generated by the following five constructors, each listed with its respective type:

$$\begin{array}{ll} \mathbf{eps}: & \mathbf{Str}.A.P \\ \mathbf{act}: & A \to \mathbf{Str}.A.P \\ \mathbf{seq}: & (\mathbf{Str}.A.P \times \mathbf{Str}.A.P) \to \mathbf{Str}.A.P \\ \mathbf{alt}: & (P \times \mathbf{Str}.A.P \times \mathbf{Str}.A.P) \to \mathbf{Str}.A.P \\ \mathbf{star}: & (P \times \mathbf{Str}.A.P) \to \mathbf{Str}.A.P \end{array}$$

The mapping $\mathcal{S}$ from $\mathcal{R}_F$ into $\mathbf{Str}.(F.A).\mathcal{P}_F$ associates with each regular expression W the set of possible strategies $\mathcal{S}.W$ of F. The set $\mathcal{S}.W$ is defined inductively over the structure of W as follows. For all α in $F.A$, and all U, V in $\mathcal{R}_F$:

$$\begin{array}{lcl} \mathcal{S}.\varepsilon & = & \{\mathbf{eps}\} \\ \mathcal{S}.\alpha & = & \{\mathbf{act}.\alpha\} \\ \mathcal{S}.(UV) & = & \{u, v : u \in \mathcal{S}.U \wedge v \in \mathcal{S}.V : \mathbf{seq}.(u, v)\} \end{array}$$

$$
\begin{array}{lcl}
\mathcal{S}.(U+V) & = & \{t,u,v : t \in \mathcal{P}_F \wedge u \in \mathcal{S}.U \wedge v \in \mathcal{S}.V : \mathbf{alt}.(t,u,v)\} \\
\mathcal{S}.U^* & = & \{t,u : t \in \mathcal{P}_F \wedge u \in \mathcal{S}.U : \mathbf{star}.(t,u)\}
\end{array}
$$

A strategy of F for the regular expression W in $\mathcal{R}_F$ is simply an element of $\mathcal{S}.W$. In order to define what it means for a game to satisfy a given strategy, we introduce the relation **sat** over pairs of states and games, and strategies. For a state s, a game g and a strategy w we denote by (s,g) **sat** w that g started in s satisfies w. Formally, **sat** is a binary relation in $(F.S \times \mathcal{G}_F) \times \mathbf{Str}.(F.A).\mathcal{P}_F$, defined as the smallest relation satisfying the following conditions for all states s in $F.S$, all games g in $\mathcal{G}_F$, all actions α in $F.A$, all predicates t in $\mathcal{P}_F$, and all strategies u and v in $\mathbf{Str}.(F.A).\mathcal{P}_F$:

$$
\begin{array}{lll}
(s,g)\ \mathbf{sat}\ \mathbf{eps} & \text{iff} & |g| = 0 \\
(s,g)\ \mathbf{sat}\ \mathbf{act}.\alpha & \text{iff} & |g| = 1 \wedge \langle \exists x : x \in (F.A)^* : \bar{g} = x\alpha \rangle \\
(s,g)\ \mathbf{sat}\ \mathbf{seq}.(u,v) & \text{iff} & \langle \exists e,f : e +\!\!+ f = g : \\
 & & \qquad ((s,e)\ \mathbf{sat}\ u) \wedge \\
 & & \qquad (\mathbf{finite}.\bar{e} \Rightarrow ((\bar{e}.s,f)\ \mathbf{sat}\ v))\rangle \\
(s,g)\ \mathbf{sat}\ \mathbf{alt}.(t,u,v) & \text{iff} & ((s \models t) \Rightarrow ((s,g)\ \mathbf{sat}\ u)) \wedge \\
 & & ((s \not\models t) \Rightarrow ((s,g)\ \mathbf{sat}\ v)) \\
(s,g)\ \mathbf{sat}\ \mathbf{star}.(t,u) & \text{iff} & ((s \models t) \Rightarrow |g| = 0) \wedge \\
 & & ((s \not\models t) \Rightarrow (s,g)\ \mathbf{sat}\ \mathbf{seq}.(u, \mathbf{star}.(t,u)))
\end{array}
$$

The definition of the algebras **Str** and the sets $\mathcal{S}$ follows very closely the structure of the algebra $\mathcal{R}_F$. The only notable addition to the algebraic structure is the occurrence of the predicate arguments in the constructors **alt** and **star**, which we briefly motivate in the following.

For the regular expression ε and for any action α in $\mathcal{R}_F$ there is only one possible strategy, namely **eps** and **act**$.\alpha$ respectively. Also the strategy for a sequence UV is completely determined by the sub-strategies for U and for V, as a game for UV consists of a game for U followed by a game for V. On the other hand, in alternations and repetitions choices have to be made: in the case of an alternation $U + V$ the progressor can decide whether to play a game for U or a game for V, whereas in the case of a repetition U^*, the progressor can decide after each U game whether to terminate the repetition or to continue with more U games. It is this freedom of choice that is captured by the predicate arguments of the **alt** and **star** strategies: for alternations it is the test predicate that determines which sub-strategy to follow, for repetitions it determines termination.

We illustrate the above definitions with an example: consider the program *UpDown* of the previous section and the regular expression $[\text{set}][\text{down}]^*$. For the set of strategies for $[\text{set}][\text{down}]^*$ we obtain

$$\mathcal{S}.[\text{set}][\text{down}]^* \;=\; \{t : t \in \mathcal{P}_{UpDown} : \mathbf{seq}\,.(\mathbf{act}\,.[\text{set}], \mathbf{star}\,.(t, \mathbf{act}\,.[\text{down}]))\}.$$

Choosing for t the predicate $n < 0$, we consider the specific strategy

$$w \;=\; \mathbf{seq}\,.(\mathbf{act}\,.[\text{set}], \mathbf{star}\,.(n < 0, \mathbf{act}\,.[\text{down}])).$$

We also consider the following games (for all k in $\mathbf{N}$):

$$\begin{aligned}
g0 &= \langle\langle[\text{set}], [\text{down}]\rangle\rangle \\
g1 &= \langle\langle[\text{down}], [\text{set}]\rangle\rangle \\
g2.k &= \langle\langle[\text{set}]\rangle, \langle[\text{up}], [\text{down}]\rangle^k\rangle
\end{aligned}$$

With these definitions, the following statements about satisfaction of games hold, where s is an arbitrary reachable state of *UpDown*[6]:

$$\begin{aligned}
((s, g0)\ \mathbf{sat}\ w) &\equiv \text{false} \\
((s, g1)\ \mathbf{sat}\ w) &\equiv (s \models n \leq 0) \\
((s, g2.k)\ \mathbf{sat}\ w) &\equiv (s \models (k > 0 \wedge n = k - 1) \vee (k = 0 \wedge n < 0))
\end{aligned}$$

Game $g0$ does not satisfy w for any start state because it consists of exactly one round, which is not a [set]-round. Game $g1$, on the other hand, consists of a [set]-

[6] Since the initial predicate of *UpDown* is true, all states of program *UpDown* are reachable.

round followed by zero [down]-rounds. In order to satisfy w it is therefore required that the termination predicate $n < 0$ be true at the end of the [set]-round, which determines the set of possible start states as characterized by $n \leq 0$. Similarly, $g2.k$ consists of one [set]-round and k [down]-rounds, which imposes the requirement that at the end of $g2.k$ the termination predicate should be true, but that it should be false at the end of any intermediate round. The reader can check easily that this requirement results in the above characterization of the possible start states.

4.5.2 An Operational Semantics

Based on the formalization of games and strategies we are now ready to formally define the operational semantics of generalized progress properties; i.e., we define what it means for a program F to be a model of such a property:

Definition 6 (Operational Semantics) *For any program F, state predicates p and q in $\mathcal{P}_F$ and regular expression W in $\mathcal{R}_F$, we say F is a model for the generalized progress property $p \overset{W}{\longmapsto} q$, written $F \models p \overset{W}{\longmapsto} q$, and define it by:*

$$
\begin{array}{l}
F \models p \overset{W}{\longmapsto} q \qquad \equiv \\
\qquad\qquad \langle \exists w : w \in \mathcal{S}.W : \\
\qquad\qquad\qquad \langle \forall s, g : s \ \textit{is reachable} : \\
\qquad\qquad\qquad\qquad (s \models p) \wedge ((s, g) \ \mathbf{sat} \ w) \ \Rightarrow \ (s, \bar{g}) \models q \rangle\rangle
\end{array}
$$

According to the definition, F is a model for $p \overset{W}{\longmapsto} q$ if and only if there exists a strategy w with a structure determined by W, such that any game started in a reachable state satisfying p and following the rules of w reaches a state satisfying q after a finite number of actions. The existential quantification in the formula above corresponds to the progressor's ability to choose a particular strategy, whereas the universal quantification reflects the requirement that the strategy be successful regardless of the actions the opponent decides to perform.

We illustrate the operational semantics by considering some special cases for the regular expression W. First, for $W = \alpha$ for some α in $\mathcal{R}_F$, the set $\mathcal{S}.\alpha$ is the singleton $\{\mathbf{act}\,.\alpha\}$. Any game satisfying $\mathbf{act}\,.\alpha$ consists of exactly one α-round. Program F therefore satisfies the generalized progress property $p \overset{\alpha}{\longmapsto} q$ if and only if during any run starting in a reachable state satisfying p and ending with an α action, a state satisfying q is encountered.

Next, we consider the case $W = \alpha^*$ for some α in $\mathcal{R}_F$. Any strategy for α^* is of the form $\mathbf{star}\,.t.(\mathbf{act}\,.\alpha)$ for some state predicate t. Any game satisfying such a strategy is a (finite or infinite) sequence of α-rounds. In order to analyze the operational behavior implied by such a strategy we demonstrate, that $F \models p \overset{\alpha^*}{\longmapsto} q$ if and only if the condition in definition 6 is met by the specific strategy $w = \mathbf{star}\,.q.(\mathbf{act}\,.\alpha)$: clearly, by definition 6, if w satisfies the condition, then $F \models p \overset{\alpha^*}{\longmapsto} q$ holds. Conversely, we need to establish that if the condition is met by any strategy in $\mathcal{S}.\alpha^*$ then it is also met by w. Let v be a strategy establishing $F \models p \overset{\alpha^*}{\longmapsto} q$ and let t be the termination predicate of v. Furthermore let g be any game and s be any state satisfying p. If g is finite then it terminates in a state satisfying q, therefore $(s, \bar{g}) \models q$ holds as required. If g is infinite, we consider the smallest prefix e of g such that $\bar{e}.s \models t$ (or $e = g$ if t is not satisfied after any round of g). Due to the condition for the operational semantics satisfied by v, we have $(s, \bar{e}) \models q$. If e is infinite, we have $e = g$ and therefore $(s, \bar{g}) \models q$. Finally, if e is finite, then $\bar{e}$ is a prefix of $\bar{f}$, from which $(s, \bar{g}) \models q$ follows as well. This establishes that w is a *characterizing strategy* for $p \overset{\alpha^*}{\longmapsto} q$, i.e., a strategy that defines the operational semantics of the given property.

Combining this result with definition 6, we see that program F satisfies the generalized progress property $p \overset{\alpha^*}{\longmapsto} q$ if and only if during any run starting in a reachable state satisfying p and containing infinitely many α actions, a state satisfying q is encountered.

As a last example we have another look at program *UpDown* and demonstrate that it indeed satisfies the property true $\overset{[\text{set}][\text{down}]^*}{\longmapsto}$ $n < 0$. This can be seen by choosing the strategy investigated at the end of section 4.5:

$$w \;=\; \mathbf{seq}.(\mathbf{act}.[\text{set}], \mathbf{star}.(n < 0, \mathbf{act}.[\text{down}])).$$

Any game g started in some state s and satisfying w consists of one [set]-round followed by a sequence of [down]-rounds. Since b is true at the end of the [set]-round, and since it remains true once it becomes true, the values of n at the end of each [down]-round form a strictly decreasing sequence. Because termination of the repetition occurs once $n < 0$ holds, we conclude from the well-foundedness of the naturals that g is finite, and that $\bar{g}.s \models n < 0$ holds. Therefore we have $(s, \bar{g}) \models n < 0$ which establishes that $F \models$ true $\overset{[\text{set}][\text{down}]^*}{\longmapsto}$ $n < 0$.

4.5.3 Soundness and Completeness

In order to relate the deductive system for the generalized leads-to relation of section 4.3 to the operational semantics just defined, we will establish both a soundness and a completeness result. The deductive system is sound with respect to the operational semantics, if and only if any generalized leads-to property proved for a program F is indeed satisfied by F; the deductive system is complete, if and only if any generalized leads-to property that is satisfied by F can actually be proved in the deductive system for F.

To be more precise, we will establish the completeness of the deductive system relative to the expressiveness of the assertion language, in our case the predicate calculus part of the logic. We show that a generalized leads-to property satisfied by a program F can be proved under the assumption that certain sets of states can be characterized by predicates of the assertion language, and that certain predicate transformers and fixpoint operations are expressible in the assertion language as well. This notion of completeness was first investigated by Cook [Coo78], a detailed discussion of some of the issues involved can be found in [Rao95].

We establish the connection between operational semantics and deductive system in the following theorem:

Theorem 21 (Soundness and Completeness) *For any program F and regular expression W in $\mathcal{R}_F$ the deductive system defined for generalized leads-to properties is sound and relatively complete in the sense of Cook:*

$$F \models \quad p \stackrel{W}{\longmapsto} q \qquad \textit{iff} \qquad F \vdash \quad p \stackrel{W}{\longmapsto} q$$

In order to prove the above theorem, the notion of a *canonical strategy* $C.W.q$ for a given regular expression W and predicate q is needed. It has the important property that it serves as a witness strategy in the definition of the operational semantics, i.e., whenever $F \vdash p \stackrel{W}{\longmapsto} q$ holds, we can establish $F \models p \stackrel{W}{\longmapsto} q$ by virtue of $C.W.q$, and conversely, whenever $F \models p \stackrel{W}{\longmapsto} q$ can be established by virtue of some strategy, it also can be established by virtue of $C.W.q$. The definition of $C.W.q$ and the proof of the above theorem can be found in appendix D.3.

4.6 Discussion

Our goals for developing the theory of generalized progress have been threefold: such a theory should (i) provide a new way of establishing ordinary progress properties of programs by allowing the user to explicitly characterize how progress is achieved, (ii) make it possible to take advantage of design knowledge in order to more effectively verify programs, and (iii) increase the efficiency of mechanical verification procedures based on the developed theory.

These goals have been achieved by our proposed theory in the following ways: we can (i) prove ordinary progress properties by our generalized ones due to theorem 9; we have (ii) developed an algebraic theory for treating progress hints formally that can be exploited in the verification process; finally, as will be demonstrated in subsequent chapters, we can (iii) take advantage of the new formalism in the form of improved model checking procedures for generalized progress properties.

In summary, the theory of generalized progress makes it possible to incorporate action-based design knowledge into the interactive verification of concurrent systems. This is accomplished by treating hints about how progress is achieved as formal objects (namely as elements of a progress algebra) and by providing a calculus for reasoning about such hints, for relating them to program executions, and for combining them with state-based reasoning methods (such as proving safety properties).

With the theoretical foundations in place, future work on generalized progress will be centered around two questions: the relationship to other formal approaches for verification of progress properties, and the practical application of the new theory to program verification.

On the theoretical side it will be interesting to explore the relationship of our theory to automata-theoretic approaches [Kur94], to deductive systems based on linear temporal logic [M^+94], or on the propositional mu-calculus [Koz83, Bra93]. It will be worthwhile to investigate to which extent the ideas of formalizing hints and of incorporating design knowledge at various levels of detail could be exploited by these other approaches.

The application of the our theory to the practical verification of concurrent systems is investigated in the following chapter in the context of finite-state model checking. However, it is important to point out, that there is no inherent restriction of the theory of generalized progress properties to neither finite-state systems, or to model checking as a verification technique. In particular the integration with theorem provers [Gol92, OSR93, GM93] and with infinite state-space model checking approaches, as well as the extension to the compositional structure of UNITY logic [Mis] and to compositional verification in general are of great interest.

Chapter 5

Checking Progress Properties

In this chapter we show how we can improve the model checking procedure for UNITY logic, presented in chapter 3, by employing the theory of generalized progress developed in the previous chapter. By incorporating this theory into the model checking procedure we achieve two important advantages: first, the expensive **wlt** computation used in the verification condition for ordinary progress properties can be replaced by verification conditions for generalized progress properties that often are significantly easier to check. Second, a more expressive specification language is made available for the design and verification process thus providing more possibilities for analyzing and debugging programs and their specifications.

In the following, we describe the extensions to the model checking procedure for UNITY in section 5.1 and discuss some heuristics for obtaining regular expression hints in section 5.2. In section 5.3, we illustrate several aspects of the procedure with a non-trivial example, an elevator control program. Section 5.4 concludes the chapter with a discussion of the extended procedure.

5.1 Model Checking for Generalized Progress

The idea for incorporating generalized progress properties into the UNITY model checking procedure is simple: by using the definition of the **wltr** predicate transformers **(wltrEps)**, **(wltrAct)**, **(wltrSeq)**, **(wltrAlt)**, and **(wltrStar)** and theorem 12, we can derive a verification condition for generalized progress properties similar to the one for ordinary progress properties:

$$F \ \models\ p \overset{W}{\longmapsto} q \qquad \text{iff for some invariant } J \text{ of } F\text{:}$$
$$[\![J \wedge p \ \Rightarrow\ \mathbf{wltr}\,.W.(J \wedge q)]\!].$$

As it is the case with the verification conditions for the other properties of UNITY logic, the conjunction with J in the argument of $\mathbf{wltr}\,.W$ can be dropped provided J is an inductive invariant, i.e., satisfies $[\![J \Rightarrow \mathbf{wco}\,.J]\!]$. This is possible because it can be shown that for any inductive invariant J, regular expression W, and predicates p and q

$$[\![(J \wedge p \Rightarrow \mathbf{wltr}\,.W.(J \wedge q)) \ \equiv\ (J \wedge p \Rightarrow \mathbf{wltr}\,.W.q)]\!]$$

holds. Whether or not such a replacement is advantageous depends on the complexity of the representation of the invariant. By virtue of the above equivalence, however, we can use any inductive invariant with which we may be able to reduce the argument and all intermediate results in the computation of $\mathbf{wltr}\,.W.q$ with respect to that invariant.

Advantages of New Model Checking Procedure

The model checking procedure of section 3.3 is simply extended by adding the new verification conditions for generalized progress properties. In particular, the verification of safety properties and the managing of invariants is performed as previously. However, when verifying progress properties the user can experience a threefold improvement over the previous method:

Shortened Verification Time: the evaluation of the verification conditions based on the **wltr** predicate transformers is often simpler than the evaluation of the ordinary progress condition based on **wlt**. This is the case because fixpoint computations can be avoided or at least simplified, since only a subset of the program actions needs to be considered as contributing to the progress of the program. Some examples illustrating the performance gains are discussed in chapter 7.

Finer Specification Detail: with the availability of a new set of progress properties that are more expressive than the traditional **transient** , **ensures** , and $\mapsto$ operators alone, the user can refine the characterization of progress properties in order to either confirm his understanding of the program, or to analyze the program by experimenting with different regular expression hints. The algebraic structure of the progress algebras from section 4.4 is a valuable tool for relating such experiments to one another.

Improved Debugging Information: a failed verification of an ordinary progress property yields a set of violating states from which fair program executions exist that do not reach any goal states; no additional information is given about how a counterexample trace can be obtained. The regular expression hints provide such information that makes it possible to restrict the set of traces among which counterexamples can be found.

Crucial to the applicability of the generalized progress checking is the ability to find suitable regular expression hints, that either help to improve the performance of the verification, or help to analyze or debug the program more effectively. We address this issue in the next section.

5.2 Obtaining Regular Expression Hints

In section 4.1 we have mentioned the two primary ways of obtaining regular expression hints for verifying generalized progress properties: the designer's operational understanding can be captured by a regular expression hint, or such hints can be viewed as an abstract representation or an *outline* of a progress proof. At this point, we want to present two heuristics that can help with obtaining such hints: the first, called *phase-splitting*, takes advantage of the sequential structure of a program; the second, called *action-grouping*, derives hints for progress properties that depend on only a few program actions.

5.2.1 Phase-Splitting

The idea of phase-splitting is based on the transitivity rule for progress, captured by **(AxSeq)**: if it is possible to split the progress towards some goal predicate into subsequent phases separated by *intermediate* sets of states, then a strategy for the desired progress is obtained as the sequence of strategies for the individual phases.

A simple example illustrating this technique is the counter of section 4.1. Progress from true to a negative counter value, i.e., $n < 0$, is achieved in two phases: the first phase establishes that the flag b is set, the second phase then decreases the counter value below 0. Given strategies for the individual phases – [set] for the first, and [down]* for the second phase) – we derive a strategy for the progress property by sequencing, thus obtaining [set][down]* for our example.

Two things are worth mentioning about this technique: although a predicate characterizing the intermediate states between two successive phases must exist, it does not have to be stated explicitly as part of the strategy; moreover, in the context of program design by refinement, refining a phase without changing its boundary predicates corresponds to refining the strategy of that phase without affecting the strategies of other phases. For instance, let us assume that in the above counter program, the flag b was used to model the completion of some process and is to be refined to model this process behavior in a more detailed way. In that case we can obtain a strategy for the refined program simply by concatenating the new strategy for the refinement of the first phase and [down]*, the strategy for the second phase.

Phase-splitting does not improve the performance of progress checking directly: if progress towards some goal exhibits a sequential structure, then ordinary progress checking handles these sequences of phases by repeated iterations, and generalized progress checking cannot decrease the number of these required iterations. However, by virtue of splitting the progress into phases it is often possible to find successful strategies for the individual phases that are much simpler than the iteration over all program actions performed by ordinary progress checking.

5.2.2 Action-Grouping

Actions of a program can be classified in three groups with respect to a given progress property: *contributing* actions actively help in achieving progress and need to be executed in order to reach the set of goal states; *non-interfering* actions have no relevant effect on how progress is achieved; no execution of a non-interfering action can prevent progress; *interfering* actions, on the other hand, not only do not contribute to progress, they can even prevent it when being executed.

An important observation is that in the absence of interfering actions, non-interfering actions can be ignored for achieving progress; i.e., a repetition of contributing actions suffices to reach some goal state. In other words, if there are no interfering actions with respect to the progress property $p \mapsto q$, and if W is the regular expression consisting of the concatenation of all contributing actions, then $p \stackrel{W^*}{\longmapsto} q$ holds. By virtue of **(PrAlg26)** of lemma 17 the same holds for W being the alternation of all contributing actions; because of **(PrAlg25)**, however, using concatenation often results in fewer outer iterations. When using concatenation there might be different orderings of the contributing actions that might affect the complexity and the number of iterations of the check. If there is a sequential dependency among the contributing actions, this should be reflected in the ordering (cf. section 5.3.3 for an example).

For the progress property $b \mapsto n < 0$ of the counter program, actions [set] and [up] are non-interfering, and action [down] is contributing. Hence, [down]* is a successful strategy. But even in the presence of interfering actions, action grouping can be used: in a first step, interfering actions have to be eliminated, then the method can be used as described above. Often the elimination of interfering actions corresponds to an application of the phase-splitting heuristic. As an example we consider again the counter program and the progress property true $\mapsto n < 0$. For this property [down] is contributing, [set] is non-interfering, and [up] is interfering. We can eliminate [up] by falsifying its guard b. Hence, we split the progress property in two parts, true $\mapsto b$ and $b \mapsto n < 0$. The first phase is completed in one [set] step, the second phase is dealt with as described above. Together we obtain the expected regular expression strategy [set][down]* in a mostly mechanical way.

The performance gained when using action grouping compared with ordinary progress checking depends on the ratio of number of contributing actions to the total number of actions: the fewer contributing actions there are, the smaller the number of inner fixpoint computations is in each outer fixpoint iteration. Often the gain is more than linear, as the average complexity of the inner fixpoint computations tends to decrease with fewer contributing actions.

5.3 An Example: An Elevator Control Program

It is the goal of this section to demonstrate the application of the model checking procedure for generalized progress properties to a small but non-trivial example. Such an example is the following elevator control program, which has been motivated by similar programs of this kind discussed in the literature (e.g. [CWB94]).

In the following, we describe the program in section 5.3.1 and state some of its properties in section 5.3.2. Then, in section5.3.3, we derive a regular expression strategy for the main progress property of the program.

5.3.1 The Program Description

The program *Elevator* models an elevator for a building with N floors. There are three variables describing the state of the elevator: *pos* is the number of the floor (in the range from 1 to N) the elevator is currently at; *state* tells whether the elevator is currently moving upwards (*UP*), moving downwards (*DOWN*), or halting (*STOP*); finally, *dir* records the preferred direction of the elevator, -1 for downwards and 1 for upwards, while 0 indicates that there is no preferred direction.

There are also two variables that model the behavior of the users of the elevator: *req* is an array of boolean variables, indexed by the floors. The elevator is requested to go to floor i, $1 \leq i \leq N$, if and only if $req.i$ is true. There is no distinction between whether the elevator is requested to go to some floor by a user waiting on that floor, or by a user in the elevator wanting to go to that floor. Users can only issue a request (i.e., set $req.i$ to true), while the control program can only remove requests (i.e., reset $req.i$ to false). The boolean variable *user* is introduced to model the situation in which from some point onwards for some (or even for all) floors no request is issued any more. Without this variable the unconditional fairness constraint would guarantee that for every floor the elevator is requested infinitely often.

At first glance, the use of both *state* and *dir* to model the elevator seems unnecessary, since 1 and *UP*, as well as -1 and *DOWN* apparently correspond to each other. However, there is a subtle and important difference: while *state* models the physical status of the elevator, *dir* encodes a control strategy. More precisely, the variable *dir* is used to resolve conflicting requests. A conflict arises in a situation in which there are requests both above and below the current position of the elevator. In order to prevent starvation of some floors, the elevator control has to make a fair choice between going upwards and going downwards. For instance, the strategy to always honor requests from below first (which might seem reasonable, since traffic in the first floor is likely to be highest) is not fair: repeated requests on lower floors could prevent requests on higher floors from being served forever.

We employ the following fair strategy for the control program: whenever there is any request, the elevator has a preferred direction given by *dir*. As long as there is a request in the preferred direction, the elevator moves in that direction and services requests. If there is no request in the preferred direction, the preferred direction can be changed to a new value. This strategy is fair, since there can always be only finitely many requests in the preferred direction. After having serviced the last such request the elevator changes its preferred direction and services all pending floors in the new direction.

The elevator program consists of the elevator control part and the user part. The program has $N+7$ actions, 6 for the elevator control, and the remaining for the users. The control actions are: [service] services a request at the current floor; [move] moves the elevator in its preferred direction; [goOn] starts movement of the elevator in the preferred direction; [up] and [down] set the preferred direction to upwards and downwards respectively and start movement of the elevator. The user actions are: [request.i] for each floor i issues a request at floor i, provided the elevator is not there already; [toggle] alternately enables and disables request actions in order to model eventual absence of requests as explained above.

The complete program is listed below. In addition to the variables and actions described above it also contains two transparent variables: *upReq* is a boolean variable indicating whether there is a pending request above the current elevator position; similarly *downReq* indicates such a request below the current position. In the initial state the elevator is halting on floor 1 and has no preferred direction; also, there are no user requests.

program *Elevator*

declare

type $Range = int(1..N)$

var $state$: **enum**$(STOP, UP, DOWN)$

var dir : **int**$(-1..1)$

var $pos : Range$

var $req : Range \rightarrow boolean$

var $user : boolean$

always

$upReq \ = (\exists i : Range, pos < i : req.i)$

$downReq \ = (\exists i : Range, pos > i : req.i)$

initially

$pos = 1$

$(\forall i : Range : \neg req.i)$

$dir = 0$

$state = STOP$

assign

[service] $req.pos, state :=$ false, $STOP$

if $req.pos$

[move] $pos := pos + dir$

if $state \neq STOP \ \wedge \ \neg req.pos$

[goOn] $state := UP$

if $upReq \ \wedge \ dir = 1 \ \wedge \ state = STOP \ \wedge \ \neg req.pos$

~ $DOWN$

if $downReq \ \wedge \ dir = -1 \ \wedge \ state = STOP \ \wedge \ \neg req.pos$

[up] $state, dir := UP, 1$

if $upReq \ \wedge \ (dir = 0 \ \vee \ \neg downReq) \ \wedge \ state = STOP \ \wedge \ \neg req.pos$

[down] $state, dir := DOWN, -1$

$$\textbf{if } downReq \wedge (dir = 0 \vee \neg upReq) \wedge state = STOP \wedge \neg req.pos$$

[halt] $\quad dir := 0$

$$\textbf{if } \neg upReq \wedge \neg downReq \wedge state = STOP \wedge \neg req.pos$$

$([]\ i : Range :$

[request] $\quad req.i := \text{true}$

$$\textbf{if } pos \neq i \wedge user$$

$)$

[toggle] $\quad user := \neg user$

end

5.3.2 Properties of Program *Elevator*

The key property we want to prove of the elevator program is its *eventual service* property: whenever a request is issued on some floor, the elevator will reach that floor eventually and stop there. This is expressed as the following leads-to property in UNITY logic (the variable k is implicitly quantified universally over all floors):

$$req.k \mapsto pos = k \wedge state = STOP \qquad \textbf{(ES)}$$

There are also a few design invariants, which express design knowledge about the elevator control variables *state* and *dir*:

$$\textbf{invariant } state = UP \Rightarrow dir = 1 \qquad \textbf{(I0)}$$

$$\textbf{invariant } state = DOWN \Rightarrow dir = -1 \qquad \textbf{(I1)}$$

$$\textbf{invariant } state = UP \Rightarrow upReq \vee req.pos \qquad \textbf{(I2)}$$

$$\textbf{invariant } state = DOWN \Rightarrow downReq \vee req.pos \qquad \textbf{(I3)}$$

(I0) and **(I1)** state that the elevator can move only in its preferred direction. **(I2)** asserts that the elevator can only move upwards if there is a request above or at the current position (the request was issued above, but the elevator might have moved onto a floor with pending request). **(I3)** asserts the corresponding fact for moving downwards.

All invariants above can be established directly from the program text. They can be checked by the model checking procedure using the verification conditions for invariants with respect to true.

5.3.3 Finding a Strategy for *Elevator*

In order to establish the eventual service property, we want to find a regular expression hint W allowing us to check successfully the generalized progress property

$$req.k \overset{W}{\longmapsto} pos = k \ \wedge \ state = STOP \ .$$

Instead of relying on some operational understanding, we use the heuristics presented in section 5.2 and a notion of abstract top-down proofs in order to obtain a suitable strategy. Our goal is to find such a strategy as directly and with as little effort as possible.

A Simple Derivation Using Heuristics

We start with the action-grouping method and observe that all actions of the user part as well as the [halt] action can be classified as non-interfering with respect to **(ES)**: if the control program works correctly, no user action should be able to interfere with the desired progress; furthermore, as long as the request at floor k has not been serviced, there is at least one request pending, hence [halt] is disabled. The remaining five control actions can be expected to be contributing: the elevator has to move, possibly change direction, and service requests at intermediate floors. This immediately yields the following strategy for establishing **(ES)**:

$$([\text{service}][\text{move}][\text{goOn}][\text{up}][\text{down}])^* \qquad \textbf{(H0)}$$

Checking **(ES)** with this regular expression fails, if we do not take some design invariants into account. A direct check produces a set of violating start states containing, for instance, a state in which $state = DOWN \wedge pos = 1 \wedge \neg req.1$ holds and which violates design invariant **(I3)**. Using **(I3)** we are alerted to the fact that a state satisfying $state = DOWN \wedge dir = 1$ is violating **(ES)**, prompting us to use **(I1)** as well. Continuing the verification process we are asked to supply **(I0)** and **(I2)** as well; of course, we could have added them directly together with **(I1)** and **(I3)** for reasons of symmetry. With these four design invariants in place, the progress check for **(ES)** for the suggested strategy succeeds, proving the program correct and confirming our understanding of the role the program actions play in achieving progress[1].

With a little additional thought we can improve the derived strategy further. There are some sequential dependencies among the contributing actions, which we can reflect in the ordering of the actions in the regular expression: [service] possibly stops the elevator; [goOn], [up], and [down] are effective only if the elevator is stopped, but they also start moving the elevator again; [move] is effective only if the elevator is not stopped. This suggests (among others) as possible orderings either

$$([\text{service}][\text{goOn}][\text{up}][\text{down}][\text{move}])^* \qquad \textbf{(H1)}$$

or

$$([\text{service}][\text{up}][\text{down}][\text{goOn}][\text{move}])^* \qquad \textbf{(H2)}$$

both of which result in fewer fixpoint iterations than **(H0)**[2].

A Detailed Derivation Based on a Progress Proof

Examining the structure of how progress is achieved more carefully, we observe that progress from $req.k$ to $pos = k \wedge state = STOP$ takes place in two phases.

[1] We can explore this idea further by dropping any action from the hint; we then find that the property can no longer be checked successfully. This shows that all five actions are indeed contributing.

[2] **(H2)** performs better than **(H1)**; a subtle sequential dependency between [up] and [goOn] (similarly, between [down] and [goOn]) can be uncovered as part of a more detailed progress proof.

In the first phase the elevator moves to the requested floor, thereby establishing $req.k \ \wedge \ pos = k$; in the second phase the elevator simply stops, which is easily seen to be accomplished by one [service] action.

The first phase, in which most of the progress takes place, cannot be directly split into smaller phases. It is possible, however, to proceed by constructing an abstract proof of the progress property corresponding to the first phase, namely

$$req.k \quad \mapsto \quad req.k \wedge pos = k \qquad \textbf{(ES1)}$$

This proof obligation can be split into three new obligations depending on whether the elevator is initially at, above, or below the requested floor. More precisely we can establish **(ES1)** by using the disjunctivity rule on the three properties

$$req.k \wedge pos = k \quad \mapsto \quad req.k \wedge pos = k \qquad \textbf{(ES10)}$$
$$req.k \wedge pos < k \quad \mapsto \quad req.k \wedge pos = k \qquad \textbf{(ES11)}$$
$$req.k \wedge pos > k \quad \mapsto \quad req.k \wedge pos = k \qquad \textbf{(ES12)}$$

(ES10) is trivial (corresponding to the ε-hint), while **(ES11)** and **(ES12)** are symmetric. If we denote by W_{11} and W_{12} regular expression hints for **(ES11)** and **(ES12)** respectively, and if we denote by $\overline{W}$ for some hint W the hint obtained by replacing every occurrence of [up] with [down] and vice versa, we expect that $W_{12} = \overline{W_{11}}$. Using the rules **(AxAlt)** and **(AxSeq)** we obtain as a suitable hint for **(ES)** the following expression:

$$(\varepsilon + W_{11} + \overline{W_{11}})[\text{service}]$$

Hence, we consider in the following the progress property **(ES11)** and attempt to find a suitable W_{11}. Our next idea is to distinguish between different preferred directions. Since our goal is to reach floor k from some floor below k, we will do so by eventually moving upwards. If the preferred direction is upwards, we can do so directly:

$$req.k \wedge pos < k \wedge dir = 1 \quad \mapsto \quad req.k \wedge pos = k \qquad \textbf{(ES110)}$$

If the preferred direction is downwards, we have to switch it to upwards eventually:

$$req.k \wedge pos < k \wedge dir = -1 \quad \mapsto \quad req.k \wedge pos < k \wedge dir = 1 \qquad \textbf{(ES111)}$$

Finally, if there is no preferred direction, we have to choose one:

$$req.k \wedge pos < k \wedge dir = 0 \quad \mapsto \quad req.k \wedge pos < k \wedge dir \neq 0 \qquad \textbf{(ES112)}$$

Combining **(ES110)**, **(ES111)**, and **(ES112)** using the disjunction and cancelation rules of UNITY logic we can derive **(ES11)**. Let us denote by W_{110}, W_{111}, and W_{112} the yet to be determined regular expression hints for **(ES110)**, **(ES111)**, and **(ES112)**, respectively. Then, the derivation of **(ES11)** corresponds to the following equation for W_{11}:

$$W_{11} \;=\; W_{110} + W_{111}W_{110} + W_{112}(W_{110} + W_{111}W_{110}),$$

which is equivalent in the progress algebra to

$$W_{11} \;=\; W_{112}W_{111}W_{110}\ .$$

We could now continue with the top-down proof and refine the regular expressions further until we reach basic ensures properties and are able to determine all regular expression hints. Alternatively, we can resort to action-grouping and derive some regular expressions directly without having to be concerned with the details of intermediate predicates. It turns out, that both **(ES110)** and **(ES111)** require the induction principle as the next proof step. Hence, we do not lose much structure by using action-grouping directly.

For **(ES110)** we identify [service], [goOn], and [move] as contributing actions with a sequential dependency in the listed order. Similarly, we find for **(ES111)**, that [service], [up], [goOn], and [move] contribute to progress. Finally, for **(ES112)** no grouping is needed since the sequence [service][up] establishes the property. Combining these results we obtain

$$W_{11} \;=\; [\text{service}][\text{up}]([\text{service}][\text{up}][\text{goOn}][\text{move}])^*([\text{service}][\text{goOn}][\text{move}])^*$$

which is equivalent to

$$W_{11} \;=\; ([\text{service}][\text{up}][\text{goOn}][\text{move}])^* \;.$$

Using the definition of $\overline{W_{11}}$ we obtain for the final regular expression hint the following expression

$$(([\text{service}][\text{up}][\text{goOn}][\text{move}])^* \;+\; ([\text{service}][\text{down}][\text{goOn}][\text{move}])^*)[\text{service}] \;.$$

This derivation illustrates two facts: first, a simple method like action-grouping can produce accurate regular expression hints for a top-level progress property, i.e., a progress property for which we do not want to consider intermediate predicates. On the other hand, although the detailed derivation based on the abstract progress proof produced the same top-level result, it also generated many auxiliary properties with associated intermediate predicates and regular expression hints. Deriving such a detailed structure with an abstract proof is certainly more difficult than applying a simple heuristic to obtain some regular expression hint. However, a detailed structure provides a collection of properties, which can be used to analyze the program and to track down errors in case the original property does not hold. Moreover, the detailed derivation is still more abstract than a complete progress proof, particularly since at any level a detailed subproof can be replaced by a regular expression hint obtained by some other method.

5.4 Discussion

In this chapter we have described how the theory of generalized progress can be incorporated into the model checking procedure for UNITY logic. This extension improves the model checking procedure by providing a new way of verifying progress properties. The advantages of the new method are similar to the advantages of the previous procedure for checking safety and basic progress properties: (i) by utilizing a certain form of design knowledge – regular expression hints – the verification task can often be sped up significantly; (ii) the design knowledge has a simple structure and is readily available by virtue of methods and heuristics for obtaining such knowledge from operational considerations and proof outlines; (iii) the extended logic makes it possible to refine progress specifications of programs and thus serves as a valuable tool for analyzing and debugging programs and their specifications.

Several remarks about the model checking procedure of chapter 3 can be extended to the new augmented method: the verification conditions for generalized progress are in principle not restricted to finite state systems and could be combined with infinitary representations and theorem proving systems; moreover, other logics like LTL and Fair-CTL could be extended in a similar way in order to take advantage of at least some aspects of the new properties; however, several features specific to UNITY – such as the unconditional fairness, identification of actions, and the structure of the deductive system – have been exploited in order to define these new properties.

A very appealing characteristic of the new method for verifying progress properties is the flexibility with which it can be used in the design and verification process: at one extreme, a completely automated verification based on the fixpoint characterization of the leads-to operator can be attempted; at the other extreme, one could start with a detailed proof outline and verify the correctness of the individual proof steps. The most promising application of the method, however, lies somewhere in the middle: some design knowledge will be directly available and can be used at little or no extra cost due to the simplicity of the formalism in which the knowledge can be stated; additional design knowledge can be provided in response to failed verification attempts or when analyzing the program.

It was not to be expected to find a method with which the verification of progress properties under fairness could be reduced to the evaluation of some simple and local checking conditions as it is the case for safety properties. However, the new method is a step in the right direction by allowing us to simplify the verification conditions to be evaluated, while at the same time providing us with a more expressive logic as a means to reason about our programs.

Chapter 6

The UNITY Verifier System

In the previous chapters we have demonstrated how the temporal logic and programming notation of UNITY can be extended and exploited so to obtain efficient model checking algorithms. In particular, we have argued that interactive verification based on UNITY logic makes it possible to incorporate design knowledge into the verification process, either by providing or establishing sufficiently strong invariants, or by supplying progress hints in the form of regular expressions. Moreover, the interactive model checking based on UNITY logic takes advantage of the asynchronous computation model of UNITY and of the design information supplied in order to simplify or even eliminate some computations required for verifying properties. In this chapter, we describe how the previously developed theory and the proposed algorithms are implemented as part of the *UNITY Verifier System*, abbreviated as UV System, an interactive symbolic model checker for finite state UNITY programs and propositional UNITY properties.

The design and implementation of an interactive verification system for UNITY (and our extensions of UNITY logic) is motivated by three goals: first, we need to substantiate our claim that the verification of certain concurrent systems can be aided by taking advantage of the simplicity and structure of UNITY logic, and by employing a methodology for making design knowledge a formal ingredient of the verification process; second, we want to provide a useful tool for designers who routinely use UNITY for designing, analyzing, or modeling concurrent systems, which can help them in performing certain tedious and error-prone verification tasks; and third, we strive to construct a system that can serve as a platform for further research on verification techniques and methodologies.

The above goals influenced the overall design of the UV system significantly. In order to demonstrate the feasibility and advantages of the UNITY model checking approach and of the interactive use of it, a system has to be built that can be run on sizable and interesting examples, and that can be compared to other model checker implementations. The desire to build a useful tool calls for an intuitive user interface that allows for easy interactive access to the functionality provided by the model checker, while presenting information about verifier invocations in a manageable way. Furthermore, the intention of using the system as a basis for future research requires its construction in a modular and extensible fashion.

In the remainder of this chapter we describe how these design goals are met by our implementation of the UV System[1]. In section 6.1 we give an overview of the system architecture. Important features of the UV language, in which the programs and properties that are to be submitted to the verifier are written, are discussed in section 6.2. We present some aspects of the user interface of the UV System in section 6.3 and explain how the system is used for interactive program verification. We conclude this chapter with a summary of the implementation and a discussion of extensions to the system that are currently under way or are planned for the near future.

For up-to-date documentation of the UV system and for a more detailed description of various implementation issues, the reader is invited to visit the UV system home page on the world wide web at the following URL:

`http://www.cs.utexas.edu/users/markus/uv2/welcome.html`

[1] The system description is based on the current version, 2.3.3 , of the UV System.

6.1 The UV System Architecture

In order to meet the requirements of modularity, extensibility, performance, and ease of use, the architecture of the UV System has been designed to support the following features:

Object-oriented workspace: entities that are relevant for a verification task, such as programs and properties, are available to the user for direct inspection and manipulation.

Separation of user interface from core functionality: the operations of the verifier are independent from their representation in the user interface; different interfaces can be provided for different tasks or user preferences, experimenting with the interface does not require extensive changes to the system core.

Scriptability: modifying the user interface, performing repeated tasks or series of experiments, or recording statistics in an automated fashion becomes possible without having to recompile the entire system.

Adequate Input Language: a strongly typed language with user definable expressions and data types close to the original UNITY notation makes it possible to conveniently state programs and properties.

Efficient Symbolic Representation: expressions and formulae used to represent various parts of the programs and properties under consideration are encoded and represented by ordered binary decision diagrams (OBDDs, [Bry86]), and can, therefore, often be stored and manipulated very efficiently.

These features are implemented in a system architecture that consists of two separate layers: the *UV kernel* provides the core functionality of the system, while a *GUI* (Graphical User Interface) allows the user to access the kernel in a convenient manner. The architecture is based on the Tcl/Tk system ([Ous94]), which makes it possible to separate kernel and user interface, to make the core functionality of the system available as a set of Tcl commands, and to interactively execute and combine these commands as required by a specific verification methodology or user interface. Moreover, the use of Tcl/Tk as a GUI definition language significantly reduces both code size and development time needed to build a user interface (cf. section 6.4).

The user can interact with the two layers in different ways: the kernel is accessible as the `uvwish` shell, which is an extension of the standard Tcl/Tk windowing shell `wish`. In addition to all the usual wish commands, several UV specific commands have been added to provide the UV functionality. The user can interact directly with the command line interface of the uvwish shell. On the other hand, a GUI is entirely written as Tcl/Tk scripts on top of uvwish. There is a standard GUI provided with the UV system distribution, but it can be modified or replaced by another one without affecting the kernel. Every interaction with the kernel through a GUI can be performed with the uvwish shell alone, although often in a far less convenient way. On the other hand, not all operations of the kernel are necessarily accessible through a GUI. The overall system structure of the UV system is illustrated in figure 6.1. As can be seen the uvwish shell consists of three parts:

The Tcl/Tk Library and Parser provide all the common wish commands and the system initialization, as well as the main event loop for the entire system.

UV Tcl Commands access the special functionality and data representations of the UV system core through a well defined script command interface. Typical commands are uv_init for initializing the UV workspace, and uv_parse for parsing a string as a UV input.

The UV Workspace holds all the data objects generated and manipulated during a verification session such as formulae, programs, and properties. It contains

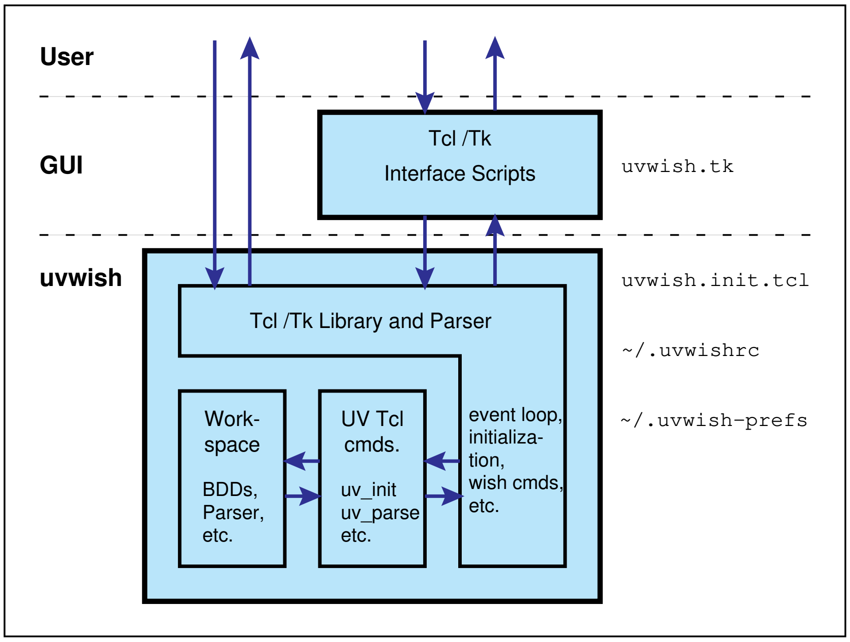


Figure 6.1: The UV System Architecture

an OBDD package for the symbolic representation of boolean functions and many auxiliary data structures. It also implements the actual system operations such as UV parsing and model checking.

While the functionality of the UV system is determined by the UV kernel (consisting of the workspace, the Tcl commands, and the Tcl/Tk interface), its appearance is mostly defined by several Tcl/Tk scripts that define how the UV system is executed:

The loading script `uvwish.init.tcl` is responsible for processing command line options, maintaining the preference file, and starting up the UV kernel.

The preference script `.uvwish-prefs` located in the user's home directory keeps track of user preferences and system settings. It is maintained by the loading script and is not intended to be directly edited by the user.

The interface script `uvwish.tk` defines the graphical user interface. It is executed as last part of the UV system start-up process.

The user script `.uvwishrc` , also located in the user's home directory, is a user-supplied resource script that can contain additional setup or customization commands. These commands are executed after the user interface has been set up.

The UV system obtains its modular and extensible structure mainly from the separation of the user interface from the system kernel, and from the modularization of commands providing the kernel functionality. Moreover, certain implementation-level techniques have been used to keep the overall structure clean and manageable: first, the object-oriented features of C++ as the implementation language are used extensively to ensure the design principles of data encapsulation and separation of functionality (model checking algorithms) from representation (symbolic representation as well as user interface). Second, the parser and the lexical analyzer for the UV input language (cf. section 6.2) are generated from sets of augmented grammar rules, allowing for greater flexibility in making modifications to the input language. Finally the OBDD package and its memory management (including a non-incremental version of a treadmill garbage collector [Bak92]) have been developed as part of the system in order to facilitate a seamless interaction with the other system components while retaining the possibility of modifying the system as part of future research. Meeting the goals of seamless interaction and future extensibility would have become more difficult when using an existing package for symbolic representations.

In the remaining part of this section we discuss briefly two important parts of the UV kernel: the workspace and its implementation in section 6.1.1, as well as the OBDD package in section 6.1.2. Aspects of the GUI and the Tcl interface are presented later in section 6.3, while an overview of the modules of the UV source code can be found in appendix C.

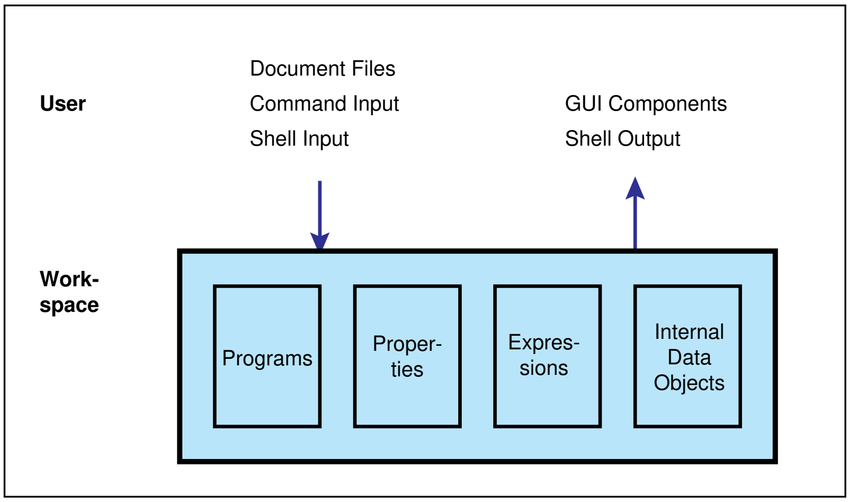


Figure 6.2: The UV Workspace

6.1.1 The UV Workspace

During a verification session the user manipulates objects that are relevant to the verification task at hand. Such objects include UNITY programs, properties, and expressions characterizing relations and sets of program states. These objects (as well as other internally used objects such as symbol tables or status information blocks) are maintained by the UV system in what can be conceptualized as a workspace, a structured collection of relevant objects. An illustration of the workspace structure is shown in figure 6.2.

There are two ways in which objects are introduced into the UV workspace: either they are constructed as a result of a translation from an external textual form according to the UV input language into an internal representation suitable for storage and manipulation, or they are created by performing certain operations with already existing objects. Building a program object from a textual description of a UNITY program is an example of object construction, whereas extracting an expression characterizing a counter-example from a property after a failed model checker invocation is a typical way of creating an object from an already existing one.

For effective and convenient interactive operation of the UV system, the internal representations of the relevant objects in the workspace need to be directly accessible through the user interface. This is accomplished by maintaining external names for those objects as well as mappings that allow the user to access objects by their external names. These external names typically consist of a prefix that characterizes the kind of an object (for instance `#expr` for expressions) followed by a unique number for that category. Certain objects can also be referred to by user-provided names.

The most important entities a user deals with during a verification session are the following:

documents and user input externally represent workspace objects such as programs and properties,

programs are the system models to be analyzed and verified,

properties are specifications to be checked for certain programs,

expressions are, in general, formulae over the state space of certain programs characterizing sets of states, relations, invariants or parts of properties,

invariants are expressions that play a special role in the verification of UNITY programs.

regular expressions are formal hints used in checking progress properties.

In the following we briefly characterize these entities and describe typical operations performed with them.

Documents and User Input

In order to introduce UNITY programs, properties, and expressions into the UV system the user provides an ASCII text description of these objects. The syntax of this external description is governed by the UV language. Input strings can be read from document files or can be provided interactively through some user input window. Document files typically contain descriptions of programs and properties to be checked, whereas interactive input is mostly used to evaluate interesting expressions or to experiment with additional properties.

Each input string provided by the user is processed by the UV Parser, which performs syntax and type checking and generates an internal representation for correctly described objects. In case syntax or type errors are encountered, a suitable error message is returned and the location of the error in the input string is reported. Parsing and introducing new objects into the workspace is done incrementally as newly described objects are added to the workspace without altering or removing existing ones.

Programs

Programs define the models (state transition systems) for which model checking is to be performed. In particular, any program defines a state space of all possible values of its variables, a set of initial states, and a labeled transition relation determining which state can be reached by executing a program statement in a given state. Furthermore there are invariants associated with every program, corresponding to sets of states that are closed under the execution of the program.

A successfully parsed program is entered into the workspace as part of the *program table* containing all currently defined programs. The user can display status information of all programs contained in the program table, and can invoke operations on the programs such as computing the strongest invariant.

Properties

Properties are the specification formulae that need to be checked for programs. Each property is associated with one program, of which it expresses a certain behavior.

A successfully parsed property is entered into the workspace as part of the *property table* containing all currently defined properties. The user can display status information of all properties contained in the property table, can invoke operations on the properties, such as model checking the associated program for the given property, or can remove properties from the workspace.

Expressions

Expressions represent typed values over some state space, typically the state space of a particular program or the global state space. Boolean expressions over the state space of a program often denote sets of program states; similarly, boolean expressions over the Cartesian product of a program state space with itself denote transition relations.

Expressions are introduced into the workspace either directly through parser invocation on some user input, or as a result of certain operations on other expressions, programs, or properties. A successfully parsed or created expression is entered into the workspace as part of the *expression table* which contains all currently defined expressions. The user can display information about expressions contained in the expression table, and can, furthermore, refer to them by using external expression names when parsing subsequent input.

Invariants

Invariants are boolean expressions over the state space of a program that play an important role in the model checking of properties of that program. Finding suitably strong invariants is critical to efficient interactive model checking of UNITY programs.

The UV system maintains at most three invariants for each program: the *type invariant* expressing that every program variable takes on values only from the domain determined by its type, the *current invariant* which is the conjunction of all invariants established for the program during a verification session, and, optionally, the *strongest invariant* characterizing the set of reachable states of the program.

Regular Expressions

Regular expressions are part of generalized progress properties introduced in chapter 4. They play an important role in utilizing action-based design knowledge in the model checking of progress properties.

Currently, the UV system maintains regular expressions only as part of generalized progress properties. Additional operations, such as keeping regular expressions as separate objects and performing algebraic manipulations on them will be provided in future system revisions.

6.1.2 Symbolic Representation Using OBDDs

Ordered binary decision diagrams are long known to be an efficient symbolic representation for boolean functions encoding sets of states and transition relations (cf. 2.3.2). We chose OBDDs as symbolic representation mostly because of the success of BDD based verification methods and implementations. It is, however, important to note that the model checking procedures for UNITY, presented in chapters 3 and 5, are independent of any particular symbolic or even explicit representation. For us, OBDDs make it possible to verify a wide range of programs, but it might still be the case that other representations (such as explicit state enumeration, or certain representations based on predicate calculus for infinite state systems) are better suited for some classes of programs.

Ideally, it should be possible to keep a strong separation of the symbolic representation and the actual verifier operations. That makes it possible to incorporate improved representations, or even to use multiple representations for different verification tasks or inputs. When work on the UV system was begun in 1992, an OBDD package was developed as a separate C++ class with the goal of supporting this separation by accessing the symbolic representation from the model checker and parser only through a well defined interface (cf. appendix C). Although by now there are complete OBDD packages and libraries available (for instance, David Long's OBDD package developed at CMU), we continued to use our own OBDD package for two main reasons: first, the intimate understanding of the package allows us to experiment with different ideas and implementations; second, as part of a one-person prototype development it is more important to concentrate on the algorithmic contributions than to attempt to incorporate all new ideas for the improvement of symbolic representations. Nonetheless, it was important to provide an efficient implementation of the low-level symbolic representation structures and manipulation operations documented in the literature. The techniques and methods currently implemented include the use of reduced OBDDs as described in [BBR90], of a combined *and-exists* operation in computing relational products ([McM93], of quantification ordering in synchronous transitions similar to those described in [BCM91], of restriction [CM90], and generalized cofactoring [TSL$^+$90].

In addition to provisions for taking advantage of the monotonicity of predicate transformers in early termination of fixpoint computations, the current implementation also uses a special second level cache for memoizing certain and-exists computations in addition to the standard *if-then-else* (ITE) cache ([BBR90]. We observed, in particular, that in the presence of relatively small deterministic multiple assignments, typical of many UNITY programs, a hit-rate of above 30% was achieved for the and-exists cache, resulting in a significant gain in performance. The and-exists caching technique has been used in other systems as well [Fil94], where somewhat smaller performance improvements on large examples have been observed.

6.2 The UV Input Language

A user of the UV system describes most entities that play a role in a verification session by using the UV input language. UNITY programs, UNITY properties, and expressions involving program variables and predicates are typical entities, which are entered into the UV workspace after being parsed and compiled into an internal OBDD based form suitable for further processing. Therefore, the UV input language defines the set of possible user inputs, and provides the user with a convenient way of expressing programs, properties, and expressions. These goals are met by designing the UV input language to be close to the original UNITY notation while having the following features:

Finiteness: in order to ensure finiteness of all state spaces, the UV input language allows only finite data types.

Strong Typing: a strong type system makes it possible to derive information about invariants from the program text.

Complete Set of Properties: all properties of UNITY logic are implemented including the generalized progress properties introduced in chapter 4.

Statement Labels: in extension to the traditional UNITY notation, all statements have unique labels by which they can be identified in debugging and formalizing progress hints.

In the following, we briefly describe some aspects of the UV input language. In particular, we discuss parse units, types and type checking, expressions, programs, and properties. In our presentation we show some rules of the grammar for the UV input language. A complete formal description of the grammar for the UV input language and of the UV type system can be found in appendix A.

6.2.1 Name Space and Scoping

The UV system has two name spaces: one for variable, constant, type, program, and field names (henceforth referred to as object names), the other for action names. The structures of the name spaces determine the visibility of names in a system description: associated with each action and each object is a *scope*, defined as parts of the description in the UV input language from which the action or object can be referred to by its name. Both action and object name space have a tree structure with the global scope as their roots and sub-scopes as their children. Sub-scopes for both objects and actions are introduced with every program definition, additional sub-scopes for objects (field names) are created for each record type.

Every object and action is visible in the scope where it is declared and recursively in all children of that scope, subject to the restriction that names in sub-scopes (lower scopes) hide the same names in outer scopes. Names declared in the same scope of the same name space have to be unique, e.g. , it is illegal to declare two enumeration types with a shared enumeration constant name in the same scope, since the enumeration constants are declared in the same scope.

The action name space has an empty global scope and a local scope for each program containing exactly the statement labels of that program.

The object name space has a global scope containing certain predefined names, such as the type `boolean` and the constants `true` and `false`, as well as all object names declared by the user globally (e.g. , program names are globally declared constants of program type). On the other hand, object names declared inside programs (or field names of record types) have local scope, i.e., are only visible from within the program, or from within expressions and properties that are placed into the program context.

6.2.2 Types

The benefits of using type systems in programming are well known. In addition to the ability of recognizing certain program errors at compile-time, a suitable type system provides the user with invariants that can be derived from the program text by simple syntactic operations. These invariants come at no cost for the programmer since there are no proof obligations to establish them.

We, therefore, adopt a simple strong type system of finite types for the UV input language. In this type system every expression occurring in a program or in a property has a statically well defined type. In the following we describe the available types together with the operations on them, and sketch the rules for correctly typing programs and properties.

The UV Type System

The type system of the UV input language consists of nine different kinds of types. The simple types are `boolean`, `number`, `cyclic`, `bits`, `int` (finite range integers), `enum` (enumeration), and `program`; the structured types are record and mapping. The simple types are all finite, and the structured types are finite types constructed from simpler finite types.

Among the simple types, the number and program types are restricted in the sense that the user cannot declare variables of these types: any program can be thought of as the definition of a constant (the program name) of type program, and any expression made up from only number literals and arithmetic operators is given a number type. There are no operations defined on the program type, but program constants are used in specifying the context of properties.

The boolean type has two predefined constants `true` and `false` and the usual boolean operators. Finite enumeration types are given by ordered list of unique enumeration constants; these constants define the possible values of the type as well as a linear order (from smallest to greatest element) on it. Additionally, the

UV type system provides three finite-range integer types, that differ slightly in the way they relate to the actual set of integers, i.e., in the way the arithmetic operators `+` and `-` are defined (the comparison relations are similarly defined for all integer types, as the relation on the corresponding integers):

Cyclic types: `cyclic(n)` represents the integers modulo the positive natural `n`, i.e., the cyclic group of `n` elements. Inequality relations are to be used carefully, since they do not obey monotonicity laws (e.g. , in `cyclic(5)`, `2<4` but not `2+2<4+2`).

Bits types: `bits(n)` represents `n`-bit numbers for which arithmetic is performed modulo 2^{n}. They are similar to the `cyclic` types, but support additional bit-level operations[2].

Interval types: `int(m..n)`: represents the intervals of integers from `m` up to and including `n`. The type of an expression involving interval type variables is the syntactic minimal interval type suitable for the result; for instance, if `x` has type `int(2..7)`, the expression `x-3` has type `int(-1..4)`. No modulo arithmetic is performed.

The two kinds of structured types are the record type and the mapping type, the latter of which can be thought of as a generalization of array types. A record type is defined by a non-empty set of pairs of *field-names* and previously defined types. A mapping type is determined by two previously defined types, an index type and an element type. A variable of a mapping type is a (finite) mapping from the index type to the element type. Arrays in languages like C++ or PASCAL are special cases of mappings in which the index type is some integer or range type. A few examples of type declarations are:

```
type Range = int(1..8);
```

[2] not implemented currently

```
type State = enum{idle, trying, critical};
type Process = {id: cyclic(16); state: State};
var process: Process;
var network: Range -> Process;
type Monitor = Process -> enum{never, once, often};
```

Note that the index type of a mapping type can be a structured type (see `Monitor` in the example above). Accessing components of a record type or elements of an index type is expressed with the function application operator `.` as in `process.state` or `network.(i+1)` .

Subtypes

The notions of *subtypes* and *type coercion* play an important role in determining whether a given program, property, or expression is typed correctly: only correctly typed input is compiled into the internal representation, and any (sub-)expression of a correctly typed input has a unique, statically determined type.

Informally, a type S is a subtype of type T if every element of S can be regarded as an element of T. For instance, extending a record type R by a new component yields a subtype of R, since any element of the new type can be regarded as an element of R (in which the value of the new component is ignored).

The subtype relation on integer types is, at the first glance, somewhat unusual: `number` types are subtypes of all other integer types, `cyclic` and `bits` types are only subtypes of themselves, and `int` types are subtypes of any `int` type. Types `cyclic` and `bits` types are kept incompatible with other integer types because they have a different algebraic structure. Compatibility of arbitrary `int` types was chosen in order to support the informal semantics of such types as providing a *window* into the infinite set of integers. In particular, this compatibility is a requirement for being able to handle assignments of the form `x := x+1` where `x` has interval type `int(m..n)`; the right-hand side has type `int(m+1..n+1)` which needs to be a subtype of `int(m..n)` in order to comply with the type rule for assignments. Instead of disallowing such an assignment altogether, it is given the semantics that the value of the right-hand side is some arbitrary value in the target range from `m` to `n` in case the value of `x+1` falls outside this range. An implementation is free to choose that value arbitrarily.

The rationale behind this semantics is the following: an assignment such as the above is inappropriate for a finite state program, since it can generate an infinite program behavior. The only reasonable occurrence of such an assignment should be guarded by some predicate `b` that effectively restricts the growth of the value of `x`. The intention should be that in an actual program execution the statement is never enabled in a state in which the right-hand side produces an out-of-range value. In other words, by including an assignment such as `x := x + 1 if b` in a program, the user actually generates the proof obligation `invariant b ==> x<n`, which should be checked as a required property of the program.

The subtyping relation for structured types is defined recursively as follows: a record type `R` is a subtype of a record type `S`, if and only if every field-name of `S` occurs in `R` and for each field-name `f` of `S` the type associated with `f` in `R` is a subtype of the type associated with `f` in `S`. A mapping type `M` defined as `A->B` is a subtype of a mapping type `N` defined as `C->D`, if and only if `B` is a subtype of `D` and `C` is a subtype of `A`.

Type Checking of Expressions

An expression is well-typed if the UV type checking algorithm successfully labels all subexpressions with their types. The type checking algorithm proceeds by labeling subexpressions with their types in a bottom-up fashion starting with literals and variables, while requiring that the types of any subexpressions occurring as arguments to some operator be compatible with that operator.

6.2.3 Parse Units

The input to the UV parser (more precisely, to the `uv_parse` command of the `uvwish` shell) consists of a sequence of input units, each of which either declares some types or variables, or describes a program or a property, or defines an expression.

All units that have been parsed correctly and have passed the type checking successfully are compiled into an internal OBDD-based representation. In particular, programs are internally represented as a set of statements, each of which is symbolically represented by a boolean function characterizing the transition relation of the statement. Similarly, the internal representation of a property includes the OBDDs representing the expressions from which the property is built, and – in the case of generalized progress properties– also a representation of the regular expression that is part of the property. Expressions are compiled into vectors of OBDDs, the size of which is determined statically by the type of the corresponding expression.

Programs

A program consists of a program name and four sections: the `declare`-section declares local types and variables and, thereby, defines the local state space of the program; the `always`-section defines transparent variables ([CM88]) that serve mostly as abbreviations to render the program easier to read and understand; the `initially`-section defines the set of initial states of the program, and the `assign`-section defines the state transition in terms of asynchronous program actions. Several examples of programs are given in chapter 7.

A program is well-typed if the following conditions are met: all of its expressions are well-typed, the types of all expressions in the *always* section are subtypes of the declared types of the corresponding transparent variables, all expressions in the *initially* section are of type boolean, all expressions appearing as guards in statements in the *assign* section are of type boolean, and the types of all expressions appearing on the right-hand side of the assignment operator are subtypes of the corresponding types of the left-hand side variables, record components, or mapping components.

Properties and Expressions

A property is defined in the context of exactly one program, which is named in a context declaration preceding the property. As a consequence, all global symbols and all local symbols defined in the associated program can be referred to in the property. Some examples of properties of a program named `sample` are:

```
in sample: n=3 unless m>0;
in sample: true --> n=3 by [alpha]*[beta];
```

A property is well-typed if and only if all of its expressions are well-typed, and – unless the property is a **constant** property – are of type boolean.

6.3 The User Interface

Interaction with the UV system can take place at two different levels: the Tcl interface provides access to the core functionality of the UV workspace, whereas a GUI makes some of that functionality accessible in a more convenient way. In the following, we describe these two interfaces in some more detail: first, we describe the *standard* GUI that is provided with the UV system distribution; then, we briefly summarize the underlying Tcl interface.

6.3.1 The Standard Graphical User Interfaces

When interacting with the UV system through its graphical interface the user typically deals with the following parts:

The Command Window gives access to system commands, displays system messages, and allows the user to invoke the parser.

Document Windows are created for each document file in use and allow the user to edit, save and parse documents.

The Program Table Window lists all current programs and allows the user to examine and perform operations on them.

The Property Table Window lists all current properties and allows the user to examine and perform operations on them.

The Expression Table Window lists information about all current expressions.

Property Information Windows display detailed status information about selected properties.

In the following we show typical instances of these interface components and briefly explain how they can be used.

The Command Window

The command window consists of the following four parts arranged from top to bottom; see figure 6.3 for an example.

The Menu Bar provides access to various system operations.

Command Buttons invoke operations on the command window.

The System Message Area displays system messages such as parser error messages or status messages.

The User Input Area allows the user to interactively parse and process UV input.

The menu bar contains a pull-down menu labeled `File` providing operations for creating new documents (`New`), opening an already existing document (`Open...`), and for terminating a verification session (`Quit`). Other menus for accessing system

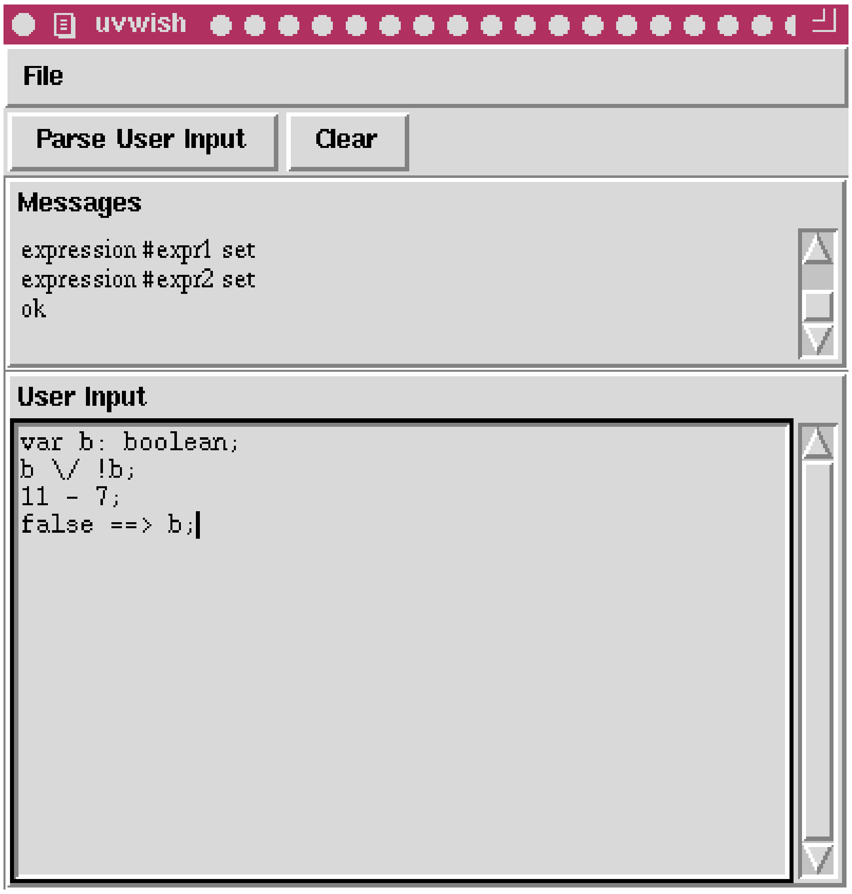


Figure 6.3: A Command Window

level information, setting preferences, or invoking tools like formulae browsers will be added in future releases.

There are two command buttons, one labeled `Parse User Input` for running the UV parser on the entire content of the user input area of the command window, the other labeled `Clear` for clearing both the message area and the user input area of the command window. Status and error messages produced by the parser are displayed in the message area of the command window, and the error position is indicated in the user input area by highlighting the token that caused the error.

Document Windows

A document window is used for viewing and editing input to the UV system, and for filing and parsing operations on such input documents. A document window consists of the following three parts listed as they appear in the layout of the window:

Command Buttons allow the user to perform filing and parsing operations on the document.

The Document Message Area displays messages that result from performing operations on the document.

The Document Content Area is an editable text area that can be used for browsing, writing, or modifying the document content.

An example of a typical document window is shown in figure 6.4.

A document window contains four command buttons that cause the following operations to be performed:

`Parse` runs the UV parser on the entire document content of the document window. Status and error messages produced by the parser are displayed in the message area of the document window, and the error position is indicated in the document content area by highlighting of the token that caused the error.

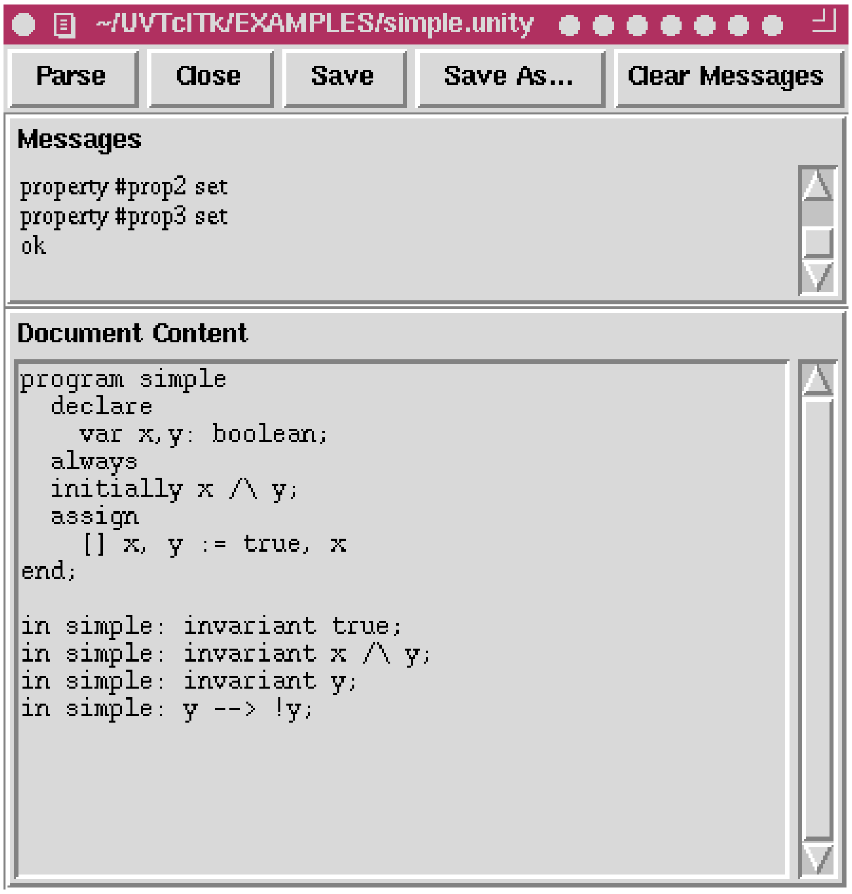


Figure 6.4: A Document Window

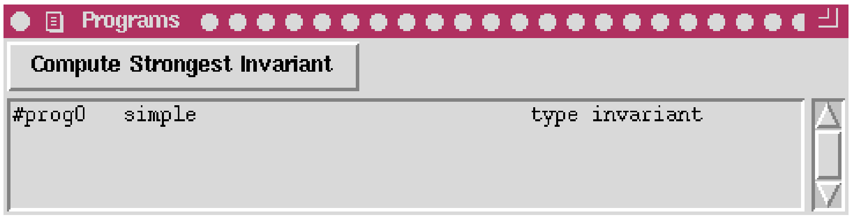


Figure 6.5: A Program Table Window

`Close` closes the document window. Changes made to the document content that have not been saved are lost.

`Save` saves the document to a file. If the document was opened from a file, changes are saved to that file. If the document was newly created, the user is prompted for a new filename under which the document is to be saved.

`Save As...` saves the document to a file, but always prompts the user for a filename under which the document is to be saved.

`Clear Messages` clears the message area of the document window.

The Program Table Window

All programs currently in use are listed in the program table window. The window consists of a command button labeled `Compute Strongest Invariant` and a scrollable list of all programs currently in the UV workspace. An example of a typical program table window is shown in figure 6.5.

For each program one line is displayed containing the program ID, the program name, and information about which invariant has been computed for the program so far. Program lines can be selected with the usual click and drag techniques: clicking the command button causes the strongest invariant to be computed for all selected programs. Although the UV kernel implements different algorithms for computing the strongest invariant (such as forward chaining and iterative squaring), the command button invokes the algorithm that has been found to be the most

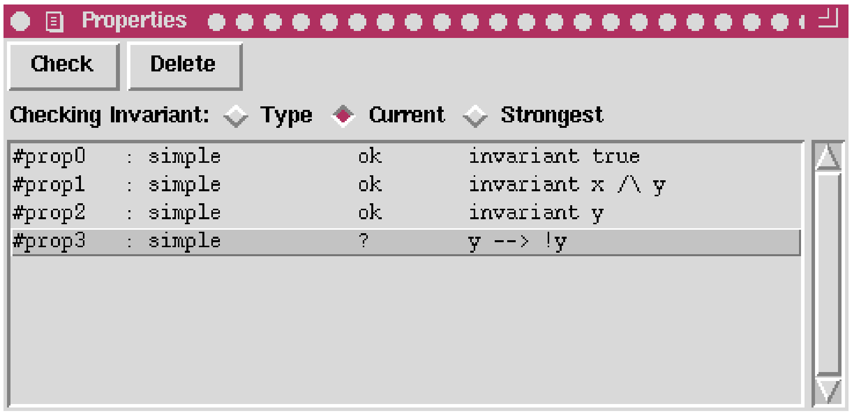


Figure 6.6: A Property Table Window

efficient for most programs. This algorithm is called *frontier forward chaining*; its name is due to the fact that in successive iterations of the forward exploration of the state space successors of those states are determined that have been added only in the previous iteration.

The Property Table Window

All properties currently in use are listed in the property table window. The window consists of the following three parts given in the order in which they appear in the layout of the window:

Command Buttons allow the user to perform operations on selected properties.

Invariant Radio Buttons are used to specify which invariant is to be used in checking properties.

The List of Properties is a scrollable list of all properties currently in the UV workspace.

An example of a typical property table window is shown in figure 6.6.

A property table window contains two command buttons: the first, labeled `Check`, invokes the model checker on all currently selected properties; the second, labeled `Delete`, removes the selected properties from the workspace.

The radio buttons are used to select whether a model checker invocation should use the type invariant, the current invariant, or the strongest invariant of the respective programs for which properties are checked.

For each property in the list of properties, one line is displayed containing summary information about the property, namely the property ID, the name of the program associated with the property, the checking status of the property, and the beginning of the textual representation of the property. The checking status is one of the following four:

`new` indicates that no checking attempt has been made for the property.

`?` indicates that the property has not yet been proved to hold in the associated program, but that there might be a suitably strong invariant with respect to which the property could still be proved.

`ok` indicates that the property has been proved to hold in the associated program.

`fail` indicates that the property has been proved not to hold in the associated program by checking it with respect to the strongest invariant of the program.

Property lines are selected for subsequent operations with the usual click and drag techniques. Double-clicking on a property line with the first mouse button brings up a property information window for this property containing more detailed information about the checking status.

The Expression Table Window

All expressions currently in use are listed in the expression table window. The window consists of a list of entries with one line for each expression. An example of an expression table window is shown in figure 6.7.

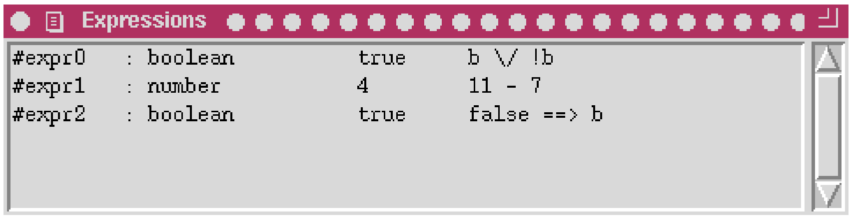


Figure 6.7: An Expression Table Window

For each expression in the UV workspace, one line is displayed characterizing the expression. Each such line consists of the expression ID, the type of the expression, the value of the expression (if the expression represents a constant, otherwise a number in square brackets showing the number of OBDD nodes required in the symbolic representation of the boolean vector encoding of the expression), and the beginning of the textual representation of the expression.

Property Information Windows

For each property listed in the property table, window a property information window can be displayed by double-clicking on the entry of the property in the property table list.

The property information window displays more detailed information about the property and is useful for analyzing model checker output and for debugging properties and their associated programs. The information presented in a property information window includes the name of the program associated with the property, a textual representation of the property, and a list of status items, each providing information about a certain aspect of the checking status of the property. An example of a property information window is shown in the figure 6.8.

Each status item is labeled with an identifying category. Some status items have additional information associated with them in the form of some expression. If such an expression is present, a button labeled `Add Expr` appears at the right of the corresponding status item line. Pressing that button causes the expression to

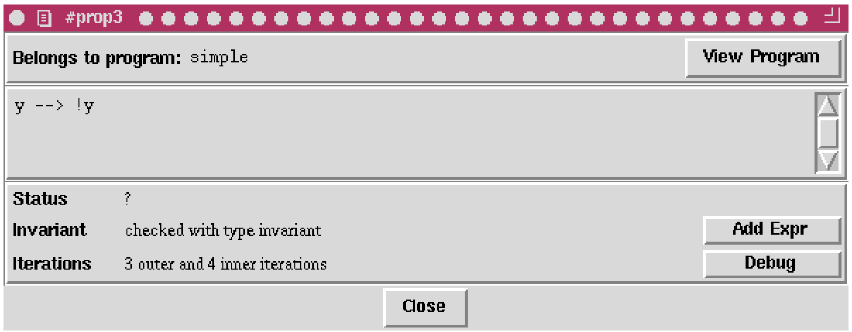


Figure 6.8: A Property Information Window

be added to the expression table window, thereby being made available for further investigation.

The following table lists all status items by their categories and describes the information available through them:

`Status` shows the checking status of the property as displayed in the property table window.

`Invariant` identifies the kind of invariant used for checking the property. This invariant is made available by pressing the `Add Expr` button.

`Implication` states the result of the implication check required for **invariant** and **co** properties. A predicate characterizing all states (within the checking invariant) falsifying the implication is available by pressing the `Add Expr` button.

`Safety` states the result of the safety check required for **co**, **unless**, **stable**, **invariant**, **constant**, and **ensures** properties. In particular, it names a program statement violating the condition (if there is any). If there is such a violating statement then a predicate characterizing all states (within the checking invariant) from which an execution of named statement violates the safety condition is available by pressing the `Add Expr` button.

`Transition` states the result of the helpful transition check required for **transient** and **ensures** properties. In particular, it names a helpful program statement (if there is any).

`Value` shows a value for the expression of a **constant** property for which the stability condition is violated (if there is such a value). If there is a value that is changed by some statement, then such a value is made available by pressing the `Add Expr` button.

`Iterations` presents the number of outer (least) and inner (greatest) fixpoint iterations in the checking of leads-to and generalized leads-to properties. A predicate characterizing all states (within the checking invariant) for which there is a fair program execution not satisfying the leads-to property is made available by pressing the `Add Expr` button.

Furthermore, all status items characterizing counter-examples (i.e., the items labeled `Implication`, `Safety`, `Value`, and `Iterations`) have another button attached to them, which is labeled `Debug`. Pressing this button brings up a new window with a witness for the violation characterized by the corresponding status item. Pressing the `Close` button closes the property information window.

6.3.2 The Tcl Interface

The Tcl interface of the UV system consists of a set of commands and some special Tcl variables added to the standard windowing shell `wish`. The entire functionality is available through these commands and variables: in particular, the standard GUI presented in the previous section is implemented completely as a set of Tcl/Tk scripts.

The commands specific to the UV system as part of the `uvwish` shell are listed briefly in the following table; a detailed description of these commands can be found in appendix B:

`uv_check` invokes the model checker.

`uv_expr` accesses information about expressions.

`uv_info` accesses information about the UV system.

`uv_init` initializes the UV system.

`uv_option` sets and displays system options and parameters.

`uv_parse` parses a string of the UV language and adds the defined entities to the UV workspace.

`uv_prog` accesses information about programs.

`uv_prop` accesses information about properties.

`uv_si` computes strongest invariants.

Currently, there are three Tcl variables that can be used to control the behavior of some UV commands, or communicate information between executing UV commands and other active Tcl/Tk scripts:

`uvAbort` is a boolean variable that when set to 1 causes the potentially time-consuming UV commands `uv_check` and `uv_si` to be aborted.

`uvProgress` is an integer variable that counts the number of major iterations during `uv_check` invocations for leads-to properties and during `uv_si` invocations.

`uvSubProgress` is an integer variable that is set to the percentage of statements checked in `uv_check` invocations for safety and basic progress properties, and in each major iteration of a `uv_check` invocation for a leads-to property.

These variables are used in the standard GUI, for instance, in order to display computation progress bars indicating the status of a lengthy UV command execution (such as computing a strongest invariant or checking a property of a complex program). This is possible because the potentially time-consuming commands `uv_check` and `uv_si` periodically process pending events, thereby updating the user interface display and checking the variables mentioned above.

6.4 Implementation Summary and Extensions

The UV system in its current form is the result of a research and implementation effort that began in 1992 as an experimental OBDD based model checker for UNITY programs, which was written in Scheme [RC86], and ran on a Macintosh computer. Since then, many significant improvements to the symbolic representation have been made, the system has been written in C++ under UNIX and the X Windows system, and the theory of incorporating design knowledge has been developed. The first version of the UV system was made publicly available in December 1994 [Kal94, Kal95b]. After a redesign and clean-up of the system architecture that had grown over a period of two years, the second and current version of the UV system was released in October 1995. Besides the new structure and the interface based on the Tcl/Tk package, version 2 improved the efficiency of the symbolic representation and the expressiveness of the UV input language. We expect the UV system to continue to be a platform for conducting and evaluating further research in verification of concurrent programs in the near future.

The current version, 2.3.3 , of the UV system is based on Tcl 7.4 and Tk 4.0, and has been successfully tested on SunOS 4.1.3 and Solaris 2.4. It was developed using the GNU suite of development tools, including the parser generator *bison* and the lexical scanner generator *flex*. The UV system source consists of about 36,000 lines of commented code. About 24,500 lines are written in C++ , 9,000 lines are C++ code generated by *bison*, *flex* and the *genclass* type generator, and the remaining 2,500 lines are Tcl/Tk code defining the user interface. Overall, 32% of the code deals with parsing and compilation of UV input into the internal OBDD representation, 26% is needed to implement the symbolic OBDD representation and the symbol table handling, about 23% is used on the model checking algorithms and related data structures, 12% is spent on implementing the Tcl interface, and only about 7% of the code makes up the standard GUI. It is worth mentioning that this confirms one of the greatest alleged advantages of using a scripting language like Tcl/Tk for building user interfaces, namely that the code size and the time required to build, modify, and maintain such an interface is significantly lower than traditional user interface designs, in which the interface code is typically written in C++ using some interface toolkit. In the first version of the UV system, we had written an interface based on the Motif[Hel92] toolkit, that not only took up more than 30% of the entire code, but also was more difficult to modify and adjust to the changing requirements of a system that is used as an experimental platform for research.

In spite of its current size and its established usefulness, there are some implementation aspects of the UV system that need to be improved. They can be divided into the following three groups:

Symbolic Representation: several well established techniques for improving the OBDD performance need to be implemented, such as dynamic variable reordering [Rud93] and partitioned OBDD representations [HD93, Jai96];

UV Language: quantified statements and formulae need to be introduced in order to improve the ease of expression of the input language; support for regular expression hints as separate objects and for performing algebraic operations on them could simplify working with progress properties;

User Interface: a better presentation of debugging information in the form of counter-example traces, and the addition of formula browsers could facilitate the user's task of finding and debugging program or specification errors.

In addition to these implementation-oriented extensions, future research concerning compositional reasoning will prompt the need for support of advanced structuring and reasoning methods, which are expected to utilize the modular and extensible structure of the current version of the UV system.

Chapter 7

Experimental Results

In this chapter we present some empirical results in applying the UV system to some practical problems. Each example has been chosen with the intention to emphasize a particular aspect of the UV system and the model checking procedures it is based on.

In section 7.1 we present a two-process mutual exclusion protocol that has been one of the first applications of the UV system, which detected errors in a manual correctness proof. A resource allocation protocol based on a dining philosopher algorithm demonstrates the increased efficiency of local safety checking in section 7.2. The counter example of chapter 4 has been included in section 7.3 to document the savings obtained by generalized progress checking. The remaining two examples, Milner's cycler in section 7.4 and the elevator control program of chapter 5 in section 7.5, are more elaborate examples that illustrate the advantages of using generalized progress properties.

Each example is presented with the program and relevant properties written using the UV input language. However, for the sake of conciseness of the listings some language features not implemented in the current revision 2.3.3 of the UV system are used. In particular, quantified assignments and formulae are used extensively. We refer to the language including these additional syntactic features as the *extended UV input language.*

All examples were run on a SPARC-20 workstation with about 20 MB of main memory allocated to the model checker.

7.1 Two-Process Mutual Exclusion

The following two-process mutual exclusion algorithm is taken from [Mis90a], where it is derived by refinement from a set of properties. Important properties of the algorithm, like mutual exclusion and absence of starvation, are established along with the manual. Although these properties can be established, some of the intermediate invariant proofs in the paper were indeed not correct. Some of these errors had been discovered independently [DF90], but they were all discovered automatically by the UV system.

The mutual exclusion algorithm for two processes U and V is based on an encoding of a *shared queue* by three boolean variables `u`, `v`, and `p`. Processes enter their IDs at the end of the queue if they are requesting to enter their critical sections; the ID at the head of the queue belongs to the process permitted to enter its critical section; upon leaving its critical section a process removes its ID from the head of the queue. The variables `u` and `v` indicate whether the IDs of U or V are in the queue, respectively; variable `p` indicates which process ID is at the head of the queue: if it is the ID of V then `p` is true, if it is the ID of U then it is false (its value is immaterial if the queue is empty).

The encoding of the mutual exclusion algorithm as a UNITY program is given in the following listing. We have added two boolean variables `hu` and `hv` to model the possibility that either process can remain in its non-critical section forever (without these variables each process would request to enter its critical section eventually due to the unconditional fairness constraint).

```
program Mutex
  declare
    type PC = enum(noncritical, requesting,
                   trying, critical, exiting);
    var m, n: PC;
    var u, v, p: boolean;
    var hu, hv: boolean;

  always
  initially
    !u;
    !v;
    m = noncritical;
    n = noncritical;
  assign

    // first process (u)
    [u0]    hu    := !hu
    [u1]    u, m := true, requesting    if hu /\ m = noncritical
    [u2]    p, m := v, trying           if m = requesting
    [u3]    m    := critical            if !p /\ m = trying
    [u4]    u, m := false, exiting      if m = critical
    [u5]    p, m := true, noncritical   if m = exiting

    // second process (v)
    [v0]    hv    := !hv
    [v1]    v, n := true, requesting    if hv /\ n = noncritical
    [v2]    p, n := !u, trying          if n = requesting
```

```
    [v3]    n    := critical           if p /\ n = trying
    [v4]    v, n := false, exiting     if n = critical
    [v5]    p, n := false, noncritical if n = exiting

end;

// invariants
in Mutex: invariant u == (m >= requesting /\ m <= critical);
in Mutex: invariant v == (n >= requesting /\ n <= critical);

// invalid invariants
in Mutex: invariant m = critical \/ m = exiting ==> !p;
in Mutex: invariant n = critical \/ n = exiting ==> p;
in Mutex: invariant (u == (m >= requesting /\ m <= critical))
                    /\ (m = critical \/ m = exiting ==> !p);
in Mutex: invariant (v == (n >= requesting /\ n <= critical))
                    /\ (n = critical \/ n = exiting ==> p);

// corrected invariants
in Mutex: invariant (u == (m >= requesting /\ m <= critical))
                    /\ (m = critical ==> !p);
in Mutex: invariant (v == (n >= requesting /\ n <= critical))
                    /\ (n = critical ==> p);

// misc. properties
in Mutex: m = trying unless m = critical;
in Mutex: m = requesting --> (p == v) /\ m = trying;
in Mutex: m = critical --> p;
```

```
// mutual exclusion
in Mutex: invariant !(m = critical /\ n = critical);

// absence of starvation
in Mutex: m = requesting --> m = critical;
```

Due to the small size of program *Mutex* the strongest invariant and all valid properties can be checked almost instantaneously. Two facts, however, are worth noting: first, the four invariants in the sections labeled *invariants* and *corrected invariants* are sufficient to establish all the other valid properties. Second, the number of inner iterations necessary for checking the absence of starvation property can be reduced from 116 to 22 by using the regular expression hint $[u2]([v2][v3][v4][v5])[u3]$, which captures the transition of process u from `requesting` to `critical` with the possibly necessary pass of process v through its critical section.

7.2 Resource Allocation: Dining Philosophers

The dining philosophers program presented here is based on the distributed dining philosophers algorithm found in [CM88]. It implements a ring topology in which two neighboring processes share a fork. Each process is in one of three states, *thinking*, *hungry*, or *eating*. A process can transit from *thinking* to *hungry* at any time, can move from *hungry* to *eating* only if neither of its neighboring processes is in its *eating* state, and moves from *eating* to *thinking* after finite time. While the transitions from *thinking* to *hungry* and from *eating* to *thinking* are under control of the processes, a scheduler has to determine when processes can transit from *hungry* to *eating*. An important property that need to be maintained by the scheduler is the *mutual exclusion* property, stating that two neighboring processes are not eating at the same time.

In order to resolve conflicts arising when two neighboring processes are ready to transit from *hungry* to *eating*, the algorithm for the scheduler maintains a partial order among the processes in which processes with higher priority are chosen over processes with lower one. To guarantee that no process is permanently discriminated against, the partial order needs to be dynamic and fair over time. This is accomplished my maintaining a directed acyclic graph over the topology of the processes, in which an edge from process u to process v indicates that u has higher priority than v. A process chosen to enter its *eating* state decreases its priority by changing all incident edges to point away from it.

The algorithm implements the directed acyclic graph in a distributed fashion in which every pair of neighboring processes communicates via three variables, `fork`, `clean`, and `rf`. These variables encode both the ordering between the processes and their respective states. For a detailed description of this encoding and the derivation of the algorithm, the reader is referred to [CM88].

In the following we show the program and several relevant properties written in the extended UV input language. The syntactic parameter `N` is instantiated to yield ring topologies of varying sizes. Syntactic features used in the presentation of this program that are not present in the current implementation of the UV input language include the quantification of expressions and assignments, the use of array literals (such as the one in the definition of `other`), and the if-then-else operator `<| |>`.

```
program dining

  declare
    type State = enum(thinking, hungry, eating);
    type Neighbors = enum(left, right);
    type Index = cyclic(N);
```

```
  var ready : boolean;    // used for modeling the nondeterministic
                          // behavior of a thinking process

  var dine : Index -> State;
  var clean : Index -> boolean;
  var fork : Index -> Neighbors;
  var rf : Index -> Neighbors;

always
  other : Neighbor -> Neighbor =
      (-> n: Neighbor |: left <| n = right |> right);
  mayEat : Index -> boolean =
  (-> i: Index |: (fork.i = left /\ (clean.i \/ rf.i = right)) /\
                    (fork.(i-1) = right /\ (clean.(i-1) \/
                                              rf.(i-1) = left)));
  sendReq : Index -> Neighbors -> boolean =
      (-> i: Index |:
      (-> n: Neighbors |: fork.i = n /\ rf.i = other.n /\
                            dine.i = hungry));
  sendFork : Index -> Neighbors -> boolean =
  (-> i: Index |:
      (-> n |: Neighbors: fork.i = n /\ !clean.i /\
                            rf.0 = n /\ !(dine.i = eating)));

initially
  (/\ i: Index |: dine.i = thinking);
  (/\ i: Index |: !clean.i);
```

```
    (/\ i: Index |: fork.i != rf.i);
    (/\ i: Index |: fork.i = left <| i != N-1 |> right);

  assign
    [toggle]    ready := !ready

    ([] i: Index |:
    [th]    dine.i := hungry if dine.i = thinking /\ ready)

    ([] i: Index |:
    [et]    dine.i := thinking if dine.i = eating)

    ([] i: Index |:
        [he]    dine.i, clean.i, clean.(i-1) := eating, false, false
            if (dine.i = hungry) /\ mayEat.i)
    ([] i: Index |:
    ([] n: Neighbor |:
        [r] rf.i := n if sendReq.i.(other.n)))

    ([] i: Index |:
    ([] n: Neighbor |:
        [f] fork.i, clean.i := n, true if sendFork.i.(other.n)))

end;

// auxiliary invariants
in dining: (/\ i: Index |:
                  invariant (dine.i = eating) ==>
```

```
                           (fork.i = left) /\ !clean.i);
in dining: (/\ i: Index |:
                  invariant (dine.i = eating) ==>
                            (fork.i-1 = right) /\ !clean.i-1);

// mutual exclusion property
in dining: (/\ i, j: Index | i != j:
                invariant !(dine.i = eating /\ dine.j = eating));
```

In order to establish the mutual exclusion of the two neighboring processes 0 and 1, we have to check the following property:

```
in dining: invariant !(dine.0 = eating /\ dine.1 = eating);
```

The attempt to compute the strongest invariant is not successful for all but small values of `N`. For instance, the program for a ring with 10 processes has 71 assignment statements, requires 51 state bits for a syntactic state space of $1.27 \cdot 10^{14}$ states which has a diameter of 104 (i.e., the number of iterations needed in the computation of the strongest invariant). Computing the strongest invariant takes about 45 minutes and produces a OBDD representation consisting of 844 nodes. The subsequent check of the invariant property takes only a few milliseconds.

We contrast this with the interactive approach in which the user supplies design knowledge in the form of invariants. From the design of the algorithm, it is immediately clear that a process can only be in its eating state if it holds both forks and both the forks are dirty. In particular, the following weaker invariants can be asserted for processes 0 and 1:

```
in dining: invariant (dine.0 = eating) ==>
                     (fork.0 = left) /\ !clean.0;
in dining: invariant (dine.1 = eating) ==>
                     (fork.0 = right) /\ !clean.0;
```

The above invariants can be established directly with the type invariant, the mutual exclusion property can then be checked successfully with respect to the current invariant. For 10 processes all three checks together take only a few milliseconds. Checking mutual exclusion for all 10 processes requires 20 auxiliary invariants, all of which can be checked in about 0.5 seconds. For 20 processes with 101 state bits and $8 \cdot 10^{24}$ states, checking of the 20 mutual exclusion properties and the 40 auxiliary invariants establishes mutual exclusion in about 2 seconds. An important point is that the auxiliary design invariants express a straightforward fact about the design of the algorithm and are therefore readily available in a situation in which the verification is not entirely separated from the program design.

7.3 Progress by Regular Expressions: A Counter

In chapter 4 the counter program *UpDown* was used as an illustrative example to introduce the concept of generalized progress properties and was discussed there in detail. Here, we present some performance measurements that demonstrate the improved efficiency of using generalized progress properties as compared to ordinary leads-to properties.

In the following listing of the program and the two progress properties the syntactic parameter `N` needs to be instantiated for different counter sizes:

```
program UpDown
  declare
    var b: boolean;
    var x: int(0..N-1);
  always
  initially
  assign
    [set]      b := true
```

```
    [up]        x := x+1 if !b /\ x < N-1
    [down]      x := x-1 if x > 0
end;

in UpDown: true --> x=0;
in UpDown: true --> x=0 by [set][down]*;
```

In the following table we summarize performance measurements for different counter sizes. The two properties, for which model checking is compared, are the ordinary leads-to property $\text{true} \mapsto x = 0$ (indicated by $\mapsto$ in the table), and the generalized leads-to property $\text{true} \overset{[set][down]*}{\longmapsto} x = 0$ (indicated by r- $\mapsto$). Three measurements are listed: *iterations* states the number of inner fixpoint iterations needed to complete the check, *ops* states the number of OBDD node lookup requests in thousands, and *time* shows the execution time in seconds. All model checker invocation establish the respective property as correct and use only the automatically generated type invariant.

N		10	20	50	100	200	500	1000	10000
itera-	$\mapsto$	107	320	1551	5608	21222	128000	506006	n/a
tions	r- $\mapsto$	41	81	201	401	801	2001	4001	40001
ops	$\mapsto$	2.5	11	100	548	2810	22283	88933	n/a
in 10^3	r- $\mapsto$	0.8	2.1	8.7	18	39	119	243	3844
time	$\mapsto$	0.3	0.3	1.1	4.8	27.3	227.9	1028.4	n/a
in s	r- $\mapsto$	0.2	0.3	0.3	0.4	0.5	1.1	2.2	38.0

As can be seen from this table, the number of iterations for the ordinary leads-to check is quadratic in `N`, whereas it is only linear for the generalized leads-to check.

7.4 Scheduling: Milner's Cycler

The scheduling problem known as *Milner's Cycler* ([Mil89]) has been described in section 2.2.1; it also has been used to illustrate the UNITY model checking procedure in section 3.4. In the following, we present some performance measurements for checking different properties.

Milner's Cycler has been used as a benchmark in the literature in order to compare different verification methods and systems. Preliminary results in [Kal95a] demonstrated the advantages of using design invariants for checking safety properties. The following listing shows the program *Cycler* together with several relevant properties written in the extended UV input language, where the syntactic parameter `N` needs to be instantiated to the actual ring size.

```
program Cycler

declare
  type Index = cyclic(N);
  type PC = enum(start, sync, choose, bc, cb);

  var a: Index;          // a holds the index of the last process
                         // that performed its a-action
  var i: Index;          // specification variable
  var cyc: Index -> PC;  // process states

initially
  a = N - 1;
  cyc.0 = start;
  (/\ i: I | i != 0: cyc.i = bc);
```

```
assign

  ([] i: Index |:
    [st] a, cyc.i := i, sync    if cyc.i = start
  )

  ([] i: Index |:
    [ch] cyc.i := bc            if cyc.i = choose
  )

  ([] i: Index |:
    [cb] cyc.i := start         if cyc.i = cb
  )

  ([] i: Index |:
    [sb] cyc.i, cyc.(i+1) := choose, start
                    if cyc.i = sync /\ cyc.(i+1) = bc
  )

  ([] i: Index |:
    [sc] cyc.i, cyc.(i+1) := choose, cb
                    if cyc.i = sync /\ cyc.(i+1) = choose
  )

end;

// design invariants
in Cycler: invariant a = i ==>
```

```
        cyc.i = sync \/ cyc.(i+1) = start \/ cyc.(i+1) = cb;
in Cycler:
invariant (/\ i: Index |: ((cyc.i = cb \/ cyc.i = start \/
                            cyc.i = sync) ==>
            (/\ j: Index | i != j: (cyc.j = choose \/ cyc.j = bc))));

// safety property characterizing cyclic behavior
in Cycler: a = i co a = i \/ a = i+1;

// progress properties asserting absence of deadlock
in Cycler: a = i --> a = i+1;
in Cycler: a = i --> a = i+1 by
            ([sc.i][cb.(i+1)][sb.i])[st.(i+1)];

// specific versions of the above progress properties for i = 1
in Cycler: a = 1 --> a = 2;
in Cycler: a = 1 --> a = 2 by ([sc.1][cb.2][sb.1])[st.2];
```

The design invariants are obtained from the design of the processes, or, alternatively, from inspection of the state transition diagram as follows. If the a-action of process i has been taken most recently (modeled by the predicate $a = i$), then either process i is in state `sync`, or it has synchronized with process $i+1$ which has not yet performed its a-action, i.e., process $i+1$ is in state `cb` or in state `start`. This is formulated as the first design invariant above.

The second design invariant is derived from the following observation: process states can be divided into two classes, one containing `cb`, `start`, and `sync`, the other containing `bc` and `choose`. The two program statements effecting a process synchronization, namely `[sb]` and `[sc]`, have the property that the two participating processes have states in different classes, and each participating process changes its class as a result of the synchronization. Together with the fact, that initially exactly one process, namely process 0, has a state in the first class, it follows, that there is always at most one process with in a state from that first class. This is formulated in the second design invariant.

In the following table we compare the checking of the safety property $a = i$ **co** $a = i \vee a = i + 1$ by computing the strongest invariant (indicated by *si*) and by using the design invariants (indicated by *inv*). The following table lists the number of OBDD node lookup requests in thousands under *ops*, the checking time in seconds under *time*, as well as the size of the syntactic state space under *states* and the maximum distance of any state from the start state under *diameter* for various sizes of `N`.

N		4	8	12	16	20
states		$1.00 \cdot 10^4$	$2.50 \cdot 10^7$	$3.52 \cdot 10^{10}$	$3.91 \cdot 10^{13}$	$3.81 \cdot 10^{16}$
diameter		20	44	68	92	116
ops	si	24	313	2479	9980	32990
in 10^3	inv	5	21	54	86	170
time	si	0.4	2.2	16.6	74.0	296.5
in s	inv	0.3	0.5	0.7	1.1	1.9

Next, we check the progress property, $a = i \;\mapsto\; a = i + 1$. Clearly, progress from $a = i$ to $a = i+1$ is achieved in two phases: first, process $i+1$ has to reach its `start` state, then $a = i + 1$ is established by virtue of the `[st.`$(i + 1)$`]` action. In the first phase synchronization between processes i and $i+1$ needs to take place, achieved by either action `[sb.`i`]` or action `[sc.`i`]`. Action `[sb.`i`]` moves process i+1 to its `start` state, whereas after action `[sc.`i`]` an execution of action `[cb.`$(i + 1)$`]` is necessary in order to complete the first phase. Since the choice between synchronization via `[sb.`i`]` or `[sc.`i`]` is not determined by the current state (for instance, if process i is in its `sync` state and process $i + 1$ is in its `choose` state, then executions with either synchronization action are possible), we choose as regular expression for the first phase the sequential composition [sc.i][cb.$(i+1)$][sb.i], which results in the regular expression for the progress property as shown above.

The following table compares the ordinary progress check (indicated by $\mapsto$) for the property $a = 1 \mapsto a = 2$ with the generalized one (indicated by r-$\mapsto$) using the regular expression hint derived above. As in the previous examples, the lines labeled with *iterations* show the total number of inner fixpoint iterations, the ones labeled with *ops* show the number of OBDD node lookup requests in thousands, and the ones labeled with *time* show the checking time in seconds:

N		4	8	12	16	20
itera-	$\mapsto$	86	174	268	370	480
tions	r-$\mapsto$	12	12	12	12	12
ops	$\mapsto$	22	287	2030	8917	29334
in 10^3	r-$\mapsto$	7	24	52	87	145
time	$\mapsto$	0.4	2.3	16.5	87.7	369.8
in s	r-$\mapsto$	0.3	0.5	0.8	1.3	1.8

In this example the effect of using a generalized progress property is particularly impressive. Since the regular expression hint does not contain a *-operator, the verification condition evaluation is reduced to a few simple fixpoint computations of depth 1. We are thereby able to take advantage of the local nature of the achievement of progress, something we could not have done with ordinary leads-to properties. We could have attempted to establish the progress property by a series of **ensures** properties; in that case, however, we would have been required to come up with suitable intermediate predicates, a task which the verification check using generalized progress properties performs implicitly.

7.5 An Elevator Control Program

The elevator control program has been discussed in detail in section 5.3. Here we present the program source in the extended UV language and report some of the performance results for checking progress properties.

The following listing shows the program *Elevator*, four design invariants, and the leads-to properties we are interested in. The syntactic parameter `N` needs to be instantiated to obtain programs for various number of floors:

```
program Elevator

declare
  type Range = int(1..N);
  var state: enum(STOP, UP, DOWN); // state of elevator
  var dir  : int(-1..1);           // direction of movement
  var pos  : Range;                // current floor
  var req  : Range -> boolean;     // array of requested floors
  var user : boolean;              // user enable flag

always
  upReq    : boolean = (\/ i: Range | pos < i: req.i);
                                   // indicates request above
  downReq  : boolean = (\/ i: Range | pos > i: req.i);
                                   // indicates request below

initially
  pos = 1 /\ dir = 0 /\ state = STOP;
  (/\ i: Range |: !req.i);
```

```
assign
  [service] req.pos, state := false, STOP
            if req.pos

  [move]    pos := pos + dir
            if state != STOP /\ !req.pos

  [goOn]    state := UP
            if upReq /\ dir = 1 /\ state = STOP /\ !req.pos
      ~          DOWN
            if downReq /\ dir = -1 /\ state = STOP /\ !req.pos

  [turnUp]  state, dir := UP, 1
            if upReq /\ (dir = 0 \/ !downReq)
                 /\ state = STOP /\ !req.pos

  [turnDown]    state, dir := DOWN, -1
            if downReq /\ (dir = 0 \/ !upReq)
                   /\ state = STOP /\ !req.pos

  [halt]    dir := 0
            if !upReq /\ !downReq /\ state = STOP /\ !req.pos

  ([] i: Range |:
    [request]   req.i := true
                if pos != i /\ user)

  [toggle]  user := !user
```

```
end;

// auxiliary invariants

in Elevator: invariant state = UP ==> dir = 1;
in Elevator: state = DOWN ==> dir = -1;
in Elevator: invariant state = UP ==> upReq \/ req.pos;
in Elevator: invariant state = DOWN ==> downReq \/ req.pos;

// quantified leads-to properties

in Elevator: (/\ i: Range |:
                    req.i --> pos = i /\ state = STOP);
in Elevator: (/\ i: Range |:
                    req.i --> pos = i /\ state = STOP by
                     ([service][turnUp][turnDown][goOn][move])*);

// specific leads-to properties for i = 3

in Elevator: req.3 --> pos = 3 /\ state = STOP;
in Elevator: req.3 --> pos = 3 /\ state = STOP by
                 ([service][turnUp][turnDown][goOn][move])*;
```

The strategy for the generalized leads-to properties has been derived and motivated in section 5.3. We compare the verification of the specific leads-to properties, first using the ordinary leads-to, then using the regular expression strategy. The results are summarized in the following table, where *iterations* states the number of inner fixpoint iterations needed to complete the check, *ops* states the number of OBDD node lookup requests in millions, and *time* shows the execution time in seconds. The properties have been checked with respect to the current invariant, being the conjunction of the four auxiliary design invariants. All checks establish the respective properties to hold for program *Elevator*. The ordinary leads-to property is indicated by $\mapsto$, the generalized by r-$\mapsto$:

N		20	50	100
states		$3.77 \cdot 10^8$	$1.01 \cdot 10^{18}$	$2.28 \cdot 10^{33}$
itera-	$\mapsto$	2264	13248	51576
tions	r-$\mapsto$	658	1798	3698
ops	$\mapsto$	2.25	18.3	659
in 10^6	r-$\mapsto$	1.17	7.6	95
time	$\mapsto$	10.8	280	>10000
in s	r-$\mapsto$	5.3	83	1150

Using the regular expression hint reduces the number of required fixpoint iterations significantly. The actual checking time does not decrease by the same factor, because the fewer inner iterations are expected to correspond on average to more complicated formulae, since they are more likely to contribute to progress (some iterations in the ordinary progress computation are more likely to terminate very quickly since they do not contribute to the desired progress). Also, some implementation specific issue may play a role; for instance, the hit ratio of the *and-exists* cache was significantly worse for the generalized progress checking than for the ordinary one. A different chaching strategy or cache organization could lead to even better results.

It should be pointed out again that the observed results confirm that the increase in performance observed when using regular expression hints is closely related to the ratio of the number of actions that contribute to progress versus the total number of actions. Even in a program like the above that does not have a behavior that can be described in sequential phases, the possibility of restricting one's attention to the actions that are helpful for the property under consideration produces significant performance improvements.

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.

Appendix A

The UV Input Language

In this appendix we present the complete grammar of the UV input language of version 2.3.3 of the UV system. First we introduce our grammar notation in section A.1, and describe lexical conventions in section A.2. Then we show the grammar rules in section A.3. We conclude this appendix in section A.4 with a brief presentation of language extensions that will be implemented in future revisions of the UV system.

A.1 Grammar Notation

The grammar for the UV input language is presented in an extended Backus-Naur Form (EBNF). There are three EBNF *meta symbols*, namely ::= for separating left and right-hand side of grammar rules, | for separating alternative right-hand sides of a grammar rule, and $(\ )^*$ to denote any finite repetition of the items enclosed in parentheses.

Non-terminals of the grammar are represented as strings enclosed in angle brackets, as in *<name>* , while *terminals* are represented using typewriter font, as in `initially` or `=!=` .

A.2 Lexical Conventions

The UV input language is case sensitive. The input is made up from a sequence of tokens and whitespace (i.e., blanks, tabs, and newline characters as well as comments). Whitespace is significant only for separating tokens, beyond that it is ignored.

Tokens are either terminals as appearing in the grammar rules below, or strings of characters representing the <*number*> , the <*name*> , or the <*external*> non-terminals.

A comment starts with the characters "//" (without the quotation marks) and ends with the first subsequent newline character.

A number as generated by the non-terminal <*number*> is a string of one or more digits, restricted by some implementation dependent limitations on the number size. It is guaranteed that 15 bit numbers (i.e., numbers from 0 to 32767) are supported.

A name as generated by the non-terminal <*name*> is any string of one or more characters made up from letters and digits starting with a letter. Names are case-sensitive and the following keywords of the UV language are not available for names: `assign`, `by`, `co`, `const`, `constant`, `declare`, `end`, `ensures`, `if`, `in`, `initially`, `invariant`, `program`, `stable`, `transient`, `type`, `unless`, and `var`.

Names are used for denoting variables, programs, types, statement labels, and record fields. Examples of names are `hello`, `isDone`, and `t34y0`.

An external name as generated by the non-terminal <*external*> consists of the string `#expr` followed (without any whitespace in between) by a string of one or more digits. The maximum number of allowed digits is implementation dependent, but at least four digits are guaranteed. External names are used for denoting expressions that have been generated interactively during a session with the UV system. Examples of external names are `#expr76` and `#expr003`.

For ease of reference we list in the following the remaining tokens of the UV input language: `:=`, `(`, `)`, `[`, `]`, `;`, `:`, `,`, `..`, and `-->`.

A.3 Grammar Rules

The following presentation is divided by the syntactic categories of the grammar.

A.3.1 Expressions

Expressions are generated by the non-terminal <*expr*> . The following table lists all operators in increasing order of their binding power (operators in the same group share the same binding power, in which case the operators associate to the left – if allowed by the typing rules):

`==` (boolean equivalence), `=!=` (boolean antiequivalence, exclusive or)
`==>` (boolean implication), `<==` (boolean follows-from)
`/\` (boolean conjunction), `\/` (boolean disjunction)
`!` (boolean negation)
`=` `!=` `>` `>=` `<` `<=` (nonboolean relational operators)
`+` (addition), `-` (subtraction)
`+` (unary plus), `-` (unary negation)
`.` (record field access, mapping indexing)

<*expr*>	::=	<*name*>
	|	<*external*>
	|	<*number*>
	|	`(` <*expr*> `)`
	|	<*expr*> `.` <*name*>
	|	<*expr*> `.` <*expr*>
	|	`+` <*expr*>
	|	`-` <*expr*>
	|	<*expr*> `+` <*expr*>
	|	<*expr*> `-` <*expr*>
	|	<*expr*> `=` <*expr*>

$\mid \quad <\mathit{expr}> \quad \texttt{!=} \quad <\mathit{expr}>$

$\mid \quad <\mathit{expr}> \quad > \quad <\mathit{expr}>$

$\mid \quad <\mathit{expr}> \quad < \quad <\mathit{expr}>$

$\mid \quad <\mathit{expr}> \quad >\texttt{=} \quad <\mathit{expr}>$

$\mid \quad <\mathit{expr}> \quad <\texttt{=} \quad <\mathit{expr}>$

$\mid \quad \texttt{true}$

$\mid \quad \texttt{false}$

$\mid \quad \texttt{!} \quad <\mathit{expr}>$

$\mid \quad <\mathit{expr}> \quad \texttt{/\textbackslash} \quad <\mathit{expr}>$

$\mid \quad <\mathit{expr}> \quad \texttt{\textbackslash/} \quad <\mathit{expr}>$

$\mid \quad <\mathit{expr}> \quad \texttt{==}> \quad <\mathit{expr}>$

$\mid \quad <\mathit{expr}> \quad <\texttt{==} \quad <\mathit{expr}>$

$\mid \quad <\mathit{expr}> \quad \texttt{==} \quad <\mathit{expr}>$

$\mid \quad <\mathit{expr}> \quad \texttt{=!=} \quad <\mathit{expr}>$

$<\mathit{exprList}> \quad ::= \quad <\mathit{expr}> \quad (\texttt{,} \quad <\mathit{expr}>)^*$

A.3.2 Types

$<\mathit{type}> \quad ::= \quad <\mathit{name}>$

$\mid \quad \texttt{boolean}$

$\mid \quad \texttt{int(} \quad <\mathit{expr}> \quad \texttt{..} \quad <\mathit{expr}> \quad \texttt{)}$

$\mid \quad \texttt{cyclic(} \quad <\mathit{expr}> \quad \texttt{)}$

$\mid \quad \texttt{bits(} \quad <\mathit{expr}> \quad \texttt{)}$

$\mid \quad \texttt{enum(} \quad <\mathit{nameList}> \quad \texttt{)}$

$\mid \quad \{ \quad <\mathit{compList}> \quad \}$

$\mid \quad <\mathit{type}> \quad \texttt{-}> \quad <\mathit{type}>$

$<\mathit{nameList}> \quad ::= \quad <\mathit{name}> \quad (\texttt{,} \quad <\mathit{name}>)^*$

<*compList*> ::= <*compItem*> (`,` <*compItem*>)*

<*compItem*> ::= <*name*> `:` <*type*>

A.3.3 Programs

<*program*> ::= `program` <*name*>
<*declare*>
<*always*>
<*initial*>
<*assign*>
`end`

<*declare*> ::= `declare` (<*declItem*> `;`)*

<*declItem*> ::= `var` <*declSymbols*> `:` <*type*>
| `type` <*declSymbols*> `=` <*type*>

<*declSymbols*> ::= <*name*> (`,` <*name*>)*

<*always*> ::= `always` (<*defItem*> `;`)*

<*defItem*> ::= <*name*> `:` <*type*> `=` <*expr*>

<*initial*> ::= `initially` (<*expr*> `;`)*

<*assign*> ::= `assign` (<*statement*>)*

$<statement>$	$::=$	$<statLabel>$ $<simpleStat>$
$<statLabel>$	$::=$	`[]`
	$\mid$	`[` $<name>$ `]`
$<simpleStat>$	$::=$	$<assignment>$ (`||` $<assignment>)^*$
$<assignment>$	$::=$	$<lhs>$ `:=` $<rhs>$
$<lhs>$	$::=$	$<var>$ (`,` $<lvalue>)^*$
$<lvalue>$	$::=$	$<name>$
	$\mid$	$<lvalue>$ `.` $<name>$
	$\mid$	$<lvalue>$ `.` $<expr>$
$<rhs>$	$::=$	$<exprList>$
	$\mid$	$<condRhs>$ (`~` $<condRhs>)^*$
$<condRhs>$	$::=$	$<exprList>$ `if` $<expr>$

A.3.4 Properties

$<property>$	$::=$	`constant` $<expr>$
	$\mid$	`invariant` $<expr>$
	$\mid$	`stable` $<expr>$
	$\mid$	`transient` $<expr>$
	$\mid$	$<expr>$ `co` $<expr>$
	$\mid$	$<expr>$ `ensures` $<expr>$
	$\mid$	$<expr>$ `unless` $<expr>$

	|	*<expr>* `-->` *<expr>*
	|	*<expr>* `-->` *<expr>* `by` *<regexp>*
<regexp>	::=	`[]`
	|	`[` *<name>* `]`
	|	`(` *<regexp>* `)`
	|	*<regexp>* `+` *<regexp>*
	|	*<regexp>* *<regexp>*
	|	*<regexp>* `*`

A.3.5 Parse Units

<input>	::=	(*<unit>*)*
<unit>	::=	
	|	*<scopedUnit>* ;
	|	`in` *<name>* : *<scopedUnit>* ;
	|	*<program>* ;
	|	*<declItem>* ;
<scopedUnit>	::=	*<expr>*
	|	*<property>* ;

A.4 Future Extensions to the Input Language

Two important features of the UV input language not implemented in version 2.3.3 are *quantified formulae, statements, and assignments* and *variable ordering directives.*

In order to introduce quantification to the language we need to add a quantifier construct:

<quant> ::= *<quantDummies>* `:` *<type>* `|` *<quantRange>* `:`

<quantDummies> ::= *<nameList>*

<quantRange> ::=
| *<expr>*

The range of a quantification starting with such a quantifier construct consists of a type for the dummy variables and an optional boolean expression restricting the values of the dummies to those satisfying the expression.

Quantified statements are introduced by modifying the grammar rule for the statement non-terminal *<statement>* as follows:

<statement> ::= *<statLabel>* *<simpleStat>*
| *<quantStat>*

<quantStat> ::= `(` `[]` *<quant>* (*<statement>*)* `)`

Similarly, quantified assignments are introduced by modifying the assignment non-terminal *<assignment>* :

<assignment> ::= *<lhs>* `:=` *<rhs>*
| `(` `||` *<quant>* *<simpleStat>* `)`

Finally, quantified expressions are obtained by adding the following alternative to the rule for the expression non-terminal *<expr>* :

<expr> ::= `(` *<op>* *<quant>* *<expr>* `)`

where *<op>* stands for any commutative and associative binary operator with a unit element.

As far as directives for influencing the variable ordering for the OBDD representation of program variables are concerned it might be argued that such information should be separated from the actual program text, as it constitutes information not so much about the program but about how to make it manageable for a specific symbolic representation. Therefore, we plan to include commands for managing variable orderings as separate entities that can be loaded, modified, browsed, and saved. However, until this will be implemented, a simple mechanism that gives the user some additional control over the variable ordering seems desirable.

Such a mechanism is the one that has been implemented in an earlier revision (1.19) of the UV system, in which the user can cause certain variables to be interleaved by tagging them with the `%interleaved` directive. This mechanism is included by extending the grammar rule for the declare section as follows:

<declare>	::=	`declare` (*<ditem>* `;`)*
<declItem>	::=	`var` *<declSymbols>* `:` *<type>* *<interl>*
	|	`type` *<declSymbols>* `=` *<type>*
<declSymbols>	::=	*<name>* (`,` *<name>*)*
<interl>	::=	
	|	`%interleaved`
	|	`%interleaved` *<name>*

The directive `%interleaved` after a list of variables causes all variables in the list to have interleaved BDD indices. The directive `%interleaved` followed by a variable name after a list of variables causes all the variables in the list to have BDD indices that are interleaved not only for the list of variables, but also for the named variable (and all variables the named variable is interleaved with). *<name>* in the last variation must be a previously declared variable name (including the immediately preceding list of declared variables), there is no restriction on its type (in particular it does not need to match the type of the declared variables).

Appendix B

The Tcl Interface of UV

The overall structure of the UV System has been described in section 6 as consisting of a kernel providing the functionality of the UV workspace and of a graphical user interface making the core functions of the kernel accessible in a convenient fashion. This separation is possible due to the use of an interface layer between kernel and user interface: the kernel makes its operations and data structures available through a number of Tcl commands, the user interface is built as a collection of Tcl/Tk scripts that rely solely on the commands provided by the kernel.

This appendix serves as a reference section for the Tcl commands implemented in the current version of the UV system (2.3.3). Each command is described in its own entry with information about command syntax, parameters and return values, and general usage description.

For displaying the command syntax, the following conventions are used: `typewriter font` is used to show text that is to be typed literally, *italics* denote variables or options that need to be replaced with some suitable text. Moreover the common meta-symbols [] for optional occurrence, {} for grouping, | for separating alternatives, and $^+$ for indicating one or more occurrences of the preceding item are used.

There are also a few additional commands and command options that will be made available in future releases of the UV system. Such commands include `uv_symbol` for accessing symbol table information about identifiers, `uv_order` for accessing and modifying the OBDD variable ordering of variable encodings, and finally `uv_regexp` for accessing information about regular expressions used as hints in generalized leads-to properties.

B.1 `uv_check`

Invoke the model checker on a property.

Synopsis

`uv_check` *n* [*inv*]

n	ID number of property to check,
inv	option indicating which invariant is to be used invariant to use for checking; one of `-type`, `-current`, and `-strongest`.

Result

A two-element list of the form *invoke status* is returned, where *invoke* indicates, whether the checker was actually invoked:

`0`	if checker was not invoked (e.g. because checking result could be determined from invariants alone),
`1`	if checker was invoked,

and *status* indicates the checking status of the property:

`ok`	if property has been proved correct,
`?`	if checking status is not known yet,
`fail`	if property has been proved incorrect.

Description

The model checker is invoked on the property with ID `#prop`*n* using the invariant indicated by *inv* as follows:

`-type`	use type invariant

`-current` use current invariant (default)

`-strongest` use strongest invariant

Special Considerations

The UV workspace must have been initialized prior to executing `uv_check`.

B.2 uv_expr

Access information about an expression.

Synopsis

`uv_expr` *n* [`-context` | `-def` | `-sat` | `-size` | `-type` | `-value`]

n	ID number of expression to access information about

Result

An exception is raised if *n* is not an integer or if there is no expression in the workspace with ID *n*. Otherwise a value is returned depending on the selected option:

no option	The empty string is returned.
`-context`	The context of the expression is returned as a list of scope names denoting the path from the global scope (corresponding to the empty list) to the scope of the expression
`-def`	The source text defining the expression is returned.
`-sat`	An exception is raised if the expression is not of type boolean or is not satisfiable; otherwise a satisfying variable assignment is returned in the form of a list of scope assignment lists. Each scope assignment list is ordered from outmost (global) scope to innermost (local) scope, and each scope list consists of the scope name ({} for the global scope) followed by lists of two elements, the first being a variable name and the second being its assigned value.

For instance, the satisfying variable assignment for the expression `x=3 /\ b` where `x` is a program variable of program `P` and `b` is a global variable, is represented as `{{} {b true}} {P {x 3}}`.

`-size` The number of OBDD nodes used in internally representing the expression is returned.

`-type` The type of the expression is returned.

`-value` The value of the expression is returned: if the expression is a constant, that constant value is returned; otherwise all variables the expression depends on are displayed in a list of scope lists, where scope lists are ordered from outmost (global) scope to innermost (local) scope, and where each scope list consists of the scope name ({} for the global scope) followed by all variables of that scope the expression depends on.

For instance, the value of an expression of program `P` depending on the program variable `x` and the global variable `b` is represented as `{{} b} {P x}`.

Description

Information about an expression in the UV workspace is returned. An expression is referred to by the ID number that is part of the expression identifier generated by the parser or by certain expression setting commands at the time the expression was entered into the workspace. Without an option `uv_expr` can be used to check whether an expression with a given ID number is present in the UV workspace. Specific information about an expression can be obtained by using one of the listed options as shown above.

Special Considerations

The UV workspace must have been initialized prior to executing `uv_expr`.

B.3 uv_info

Access information about the UV system and the UV workspace.

Synopsis

uv_info {-flags | -memory | -version}

Result

Depending on the chosen option the following information is returned:

-flags	A list of compile-time flags is returned, indicating how the UV system was built. Currently the only supported flag is the DEBUG flag indicating whether the UV system was compiled in debug mode.
-memory	Information about the current memory usage is returned as a list of five lists containing (in that order) information about the OBDD heap, the OBDD hash table, the ITE cache, the AEN cache, and the function cache. If a certain cache is not enabled (cf. the uv_option command), the corresponding list is empty. The information in each list is structured as follows: OBDD heap: total number of OBDD nodes, number of used OBDD nodes, number of allocated OBDD nodes; OBDD hash table: total number of slots, number of free slots, number of entries, maximum, average, and variance of length of entry chains, number of find operations, percentage of hits, average chaining length; ITE cache, AEN cache, function cache: total number of slots, number of free slots, number of find operations, percentage of hits.

`-version`	the revision number of the UV system presented in the form $n.m.p$ where n is the major revision number, m is the minor revision number, and p is the patch level.

Description

Information about the UV system and about the UV workspace is returned.

Special Considerations

The UV workspace must have been initialized prior to executing `uv_info` with the `-memory` option.

B.4 `uv_init`

Initialize the UV workspace.

Synopsis

`uv_init` [`-mem` n]

n Natural number in the range from 6 to 24, specifying the amount of memory allocated (see below). The default value is 16.

Result

none, or message string if workspace has already been initialized

Description

The UV workspace is initialized by allocating a OBDD heap of 2^n nodes, a OBDD hash table of 2^{n-2} entries, and proportionally sized cache tables. The initial global symbol table is set up as well.

Special Considerations

`UV_init` needs to be called once at the beginning of a session. Subsequent calls have no effect.

B.5 `uv_option`

Set system options or return their current status.

Synopsis

`uv_option` {`-bdd` | `-reduce`} [*value*]

value	the value to which the specified option is set.

Result

If no *value* parameter is given, the current value of the specified option is returned.

Description

Without a *value* parameter the current status of the specified system option is returned. If a *value* parameter is given, then the specified option is set to the indicated value. The values for the different options and their meaning are as follows:

The `-bdd` option enables certain OBDD-level caches; its value is the sum of any of the following:

`1`	use a general function cache (default),
`2`	use a separate AEN cache.

The `-reduce` option controls the use of the reduction operation on OBDDs; its value is the sum of any of the following:

`1`	use during strongest invariant computations (default),
`2`	use during wlt-computations.

B.6 uv_parse

Parse input string according to UV language grammar and update UV workspace accordingly.

Synopsis

`uv_parse` *var* $\{string\}^+$

var	Tcl variable receiving list of parse results
string	input string

Result

No value is returned if input is parsed successfully, otherwise a list of the form *pos len msg* is returned, where *pos* is the character offset into the concatenation of input strings of the first character of the offending token, *len* is the number of characters of the offending token, and *message* is an error message string describing the nature of the parse error. (The first character of the input has offset 1.)

The Tcl variable *var* contains a list of result items, one for each successfully parsed input unit, of the following form:

#expr*n*	for expressions: expression ID,
#prog*n*	for programs: programs ID,
#prop*n*	for properties: property ID,
symbollist	for declarations: list of declared symbols,

where the numbers n are uniquely assigned for each input unit in each category.

Description

The parser for the UV language is invoked on the concatenation of all string arguments. The input string can represent any number of input units like declarations, expressions, programs, or properties. Each successfully parsed unit is compiled into the UV workspace, and a suitable result item identifying the unit is appended to the Tcl variable *var* (see results above).

Parsing is terminated upon the first encountered error, in which case some error information is returned. Possible errors include syntax errors, type errors, and compilation errors (e.g. exhausted resources).

Special Considerations

The UV workspace must have been initialized prior to executing `uv_parse`.

B.7 uv_prog

Access information about a program.

Synopsis

```
uv_prog  n
uv_prog  n  -def
uv_prog  n  -invariant  [inv]
uv_prog  n  -name
uv_prog  n  -statement  [s]
```

n	the ID number of the program to be accessed
inv	an option indicating which invariant of the program is to be accessed; one of `type`, `current`, or `strongest`
s	the number of the statement the label of which is to be returned (counting form 0).

Result

An exception is raised if n is not an integer or if there is no program in the workspace with ID number n. Otherwise a value is returned depending on the selected option:

no option	The empty string is returned.
`-def`	The source text defining the program is returned.
`-invariant`	Without a *inv* parameter a characterization of the invariant of the program is returned in the form of a two element list, each element being one of `type`, `current`, or `strongest`. The first element is the weakest characterization of the invariant or the program, the second is its strongest characterization (`type` is weaker than `current`, which is weaker than `strongest`).

For instance, the result `type strongest` means that the strongest invariant has been computed and is the same as the type invariant.

If the *inv* parameter is given, the indicated invariant is entered into the workspace as an expression and an expression identifier of the form `#expr`n is returned, where n is the unique ID number which the new expression can be referenced by. An exception is raised if the strongest invariant is requested without having been computed.

`-name` The program name is returned.

`-statement` Without an s parameter the number of statements of the program is returned; otherwise the label of statement number s is returned.

Description

Information about a program in the UV workspace is returned. A program is referred to by the ID number that is part of the program identifier generated by the parser at the time the program was entered into the workspace. Without an option `uv_prog` can be used to check whether a program with a given ID number is present in the UV workspace. Specific information about a program can be obtained by using one of the listed options as shown above.

Special Considerations

The UV workspace must have been initialized prior to executing `uv_prog`.

B.8 uv_prop

Access information about a property or delete a property.

Synopsis

`uv_prop` n
`uv_prop` n `-arg` $[a]$
`uv_prop` n `-def`
`uv_prop` n `-delete`
`uv_prop` n `-info` $[key]$
`uv_prop` n `-infoexpr` key
`uv_prop` n `-op`
`uv_prop` n `-prog`
`uv_prop` n `-status`

n	the ID number of the property to be accessed
a	number indicating which argument is to be accessed (counting from 0),
key	key indicating which checking information is to be returned; one of `implication`, `invariant`, `iterations`, `safety`, `transition`, or `value`.

Result

An exception is raised if n is not an integer or if there is no property in the workspace with ID number n. Otherwise a value is returned depending on the selected option:

no option	The empty string is returned.
`-arg`	Without an a parameter the number of predicate arguments of the property is returned; otherwise argument number a (counting from 0) is entered into the workspace and a unique expression identifier is returned.

`-def` The source text defining the property is returned.

`-delete` The empty string is returned and the property is removed from the workspace.

`-info` Without a *key* parameter, a list of all key values for which there is checking information for the property is returned. If a *key* parameter is given, the following specific information about the checking status of the property is returned (if present) depending on the value of *key*:

`implication`: 1 if and only if the implication part of the property is true, 0 otherwise;

`invariant`: characterization of the checking invariant as a list of two items each having one of the values `type`, `current`, or `strongest`; cf. the `-invariant` option of the `uv_prog` command for a description of the format;

`iterations`: list of numbers of major and minor iterations;

`safety`: label of a violating statement (or empty string);

`transition`: label of a helpful statement (or empty string);

`value`: 1 if and only if there is value violating constant condition, 0 otherwise.

`-infoexpr` Depending on the *key* parameter the following expressions are entered into the workspace and an expression identifier is returned (provided expression information is available):

`invariant`: checking invariant,

`implication`: negation of implication check, `safety`: negation of safety check of a violating statement (or empty string if there is no violation),

`transition`: empty string (no expression entered),

`value`: value for which constant condition is violated (or empty string if there is no violation),

`iterations`: negation of leads-to check.

Note that the negated check expressions for the keys `implication`, `iterations`, and `safety` characterize the states that do not satisfy the required property.

`-op` The operator of the property is returned.

`-prog` The program identifier of the program to which the property belongs is returned,

`-status` The checking status of the property is returned; it is one of `ok`, `?`, `fail`, or `new`.

Description

Information about a property in the UV workspace is returned. A property is referred to by the ID number that is part of the property identifier generated by the parser at the time the property was entered into the workspace. Without an option `uv_prop` can be used to check whether a property with a given ID number is present in the UV workspace. Specific information about a property can be obtained by using one of the listed options as shown above.

Special Considerations

The UV workspace must have been initialized prior to executing `uv_prop`.

B.9 `uv_si`

Compute strongest invariant of program.

Synopsis

`uv_si` n [*opt*]

n	ID number of program for which strongest invariant is to be computed,
opt	option indicating which operation is to be performed; one of `-clear`, `-forward`, `-frontier`, and `-square`.

Result

Return the number of iterations performed for computing the strongest invariant for options `-forward`, `-frontier`, and `square`, or the empty string for option `-clear`. Raise an exception if workspace does not contain a program with ID number n.

Description

Compute the strongest invariant of the program with ID number n in the UV workspace using an algorithm determined by the option *opt* as follows (if the strongest invariant has been computed previously, it is simply recalled from a special cache):

`-clear`	do not compute the strongest invariant but forget it in case it has been computed previously; useful for comparing strongest invariant computations using different algorithms,
`-forward`	use the standard forward chaining algorithm to explore the reachable state space,

`-frontier`	use a modification of the forward chaining algorithms in which only successors of states are considered that are at the frontier of the state space exploration, i.e.that have been added to the set of reachable states in the previous iteration (default),
`-square`	use an iterative squaring algorithm.

Special Considerations

The UV workspace must have been initialized prior to executing `uv_si`.

Appendix C

The UV System Source Structure

The UV System consists in its current revision, 2.3.3 , of about 35000 lines of C++ and Tcl/Tk code. Here we give an overview of the structure of the UV sources with a brief description of each file. In our presentation we follow the hierarchical structure of the UV source directory. For every file we show its size in bytes and a short description of its content. We start with the UV root directory:

Filename: `UV/`	Size	Description
`uvwish.make`	3396	Makefile for UV system
`uvwish.cc`	2240	Contains *main* procedure; starts extended Tcl/Tk shell `uvwish`
`UVVersion.h`	27	Current revision number of UV project
`Global.h`	1422	Some global definitions
`UVPackage.h` `UVPackage.cc`	18629 6021	Interfaces of Tcl commands and implementation of UV workspace

Filename: `UV/`	Size	Description
`Failure.h` `Failure.cc`	1311 1534	Procedures for dealing with serious and catastrophic system failures
`Options.h` `Options.cc`	1667 1085	Data structure for dealing with global system options
`int.defs.h` `WSExpression.defs.h`	2696 2818	Auxiliary declarations for integers and workspace expressions; generated by GNU `genclass`

`int.WSExpression.AVLMap.h`
`int.WSExpression.AVLMap.cc`
`int.WSExpression.Map.h`
`int.WSExpression.Map.cc`
	4797
	14410
	3626
	2494
		AVL map from integer indices to workspace expressions; generated by GNU `genclass`

`uvwish.init.tcl`	6537	Tcl/Tk startup script; handles command line arguments and the preference file
`uvwish.tk`	1183	Tcl/Tk script defining user interface
`uvwish.message`	1045	Startup message about user notification

There are five subdirectories, which we describe in the following. The `BDD` subdirectory contains the files implementing the OBDD package and the symbol table:

Filename: `UV/BDD/`	Size	Description
`BDDManager.h` `BDDManager.cc`	6260 6254	Management and coordination of BDD operations
`BDD.h` `BDD.cc`	21661 42262	BDD data structures and operations
`BDDVector.h` `BDDVector.cc`	13932 12073	Data structures and operations for vectors of BDDs

Filename: `UV/BDD/`	Size	Description
`BDDMemory.h` `BDDMemory.cc`	10844 14826	BDD memory management
`Cachetable.h` `Cachetable.cc`	8997 12551	Implementation of various caches and hash tables
`Symboltable.h` `Symboltable.cc`	58588 89479	Data structures and methods for symbol tables
`BDDMapping.h` `BDDMapping.cc`	1991 1990	Data structures and algorithms for mapping program variables to BDD indices

The `MODELCHECKER` subdirectory contains the files implementing the UNITY model checking algorithm and internal representations of properties and programs:

Filename: `UV/MODELCHECKER/`	Size	Description
`Program.h` `Program.cc`	10026 20597	Program related data structures and algorithms, e.g. , strongest invariant computation
`ProgramTable.h` `ProgramTable.cc`	5368 5890	Management of program collections
`Statement.h` `Statement.cc`	8316 16331	Statement related data structures and algorithms
`Property.h` `Property.cc`	10682 29443	Property related data structures and algorithms, e.g. , verification condition evaluation
`PropertyTable.h` `PropertyTable.cc`	3401 3459	Management of property collections
`PropertyStatus.h` `PropertyStatus.cc`	13503 19025	Data structures and methods for maintaining checking status of properties

Filename: `UV/MODELCHECKER/`	Size	Description
`Regexp.h` `Regexp.cc`	2913 6245	Data structures and model checking algorithms for regular expressions

`int.defs.h`
`BDDIndex.defs.h`
`Program.defs.h`
`PropertyPtr.defs.h`
	2696
	2770
	2761
	2831
		Auxiliary declarations for integers, BDD indices, programs, and pointers to properties; generated by GNU `genclass`

`BDDIndex.AVLSet.h`
`BDDIndex.AVLSet.cc`
`BDDIndex.Set.h`
`BDDIndex.Set.cc`
	4677
	18976
	3528
	3533
		AVL set of BDD indices; generated by GNU `genclass`

`int.Program.AVLMap.h`
`int.Program.AVLMap.cc`
`int.Program.Map.h`
`int.Program.Map.cc`
	4497
	14135
	3521
	2464
		AVL map from integer indices to programs; generated by GNU `genclass`

`int.PropertyPtr.AVLMap.h`
`int.PropertyPtr.AVLMap.cc`
`int.PropertyPtr.Map.h`
`int.PropertyPtr.Map.cc`
	4753
	14355
	3606
	2488
		AVL map from integer indices to pointers to properties; generated by GNU `genclass`

The `PARSER` subdirectory contains the files describing the scanner and parser for the UV input language and implementing the BDD compiler:

Filename: `UV/PARSER/`	Size	Description
`UVLanguage.l` `UVLanguage.y`	6106 50471	Scanner and parser definitions for flex and bison
`ParseTree.h` `ParseTree.cc`	48720 116287	Data structures and methods for parse tree management, type checking, and BDD compilation

Filename: UV/PARSER/	Size	Description
ParseInfo.h ParseInfo.cc	4238 1900	Data structure for communication between parser and workspace
Display.h Display.cc	13428 19550	Data structures and methods for external representation of data items
FontInfo.h BDDIndex.defs.h BitAssignment.defs.h BitAssRep.defs.h	1408 2770 2829 2785	Font codes for special symbols Auxiliary declarations BDD indices, and internal data structures for representing bit assignments; generated by GNU `genclass`
BDDIndex.BitAssRep.AVLMap.h BDDIndex.BitAssRep.AVLMap.cc BDDIndex.BitAssRep.Map.h BDDIndex.BitAssRep.Map.cc	4927 14555 3651 2521	AVL map from BDD indices to bit assignment representations; generated by GNU `genclass`
BitAssignment.List.h BitAssignment.List.cc	8762 22333	List of bit assignments; generated by GNU `genclass`

The `TCL` subdirectory contains the files implementing the Tcl commands for providing the UV system functionality via the `uvwish` shell (cf. B). The headers for all these files are contained in `UV/UVPackage.h`:

Filename: UV/TCL/	Size	Description
UVCheck.cc	8141	Implementation of uv_check command
UVExpr.cc	5881	Implementation of uv_expr command
UVInfo.cc	7351	Implementation of uv_info command

Filename: `UV/TCL/`	Size	Description
`UVInit.cc`	2125	Implementation of uv_init command
`UVOption.cc`	3436	Implementation of uv_option command
`UVParse.cc`	13560	Implementation of uv_parse command
`UVProg.cc`	6439	Implementation of uv_prop command
`UVProp.cc`	16107	Implementation of uv_prog command
`UVSI.cc`	4350	Implementation of uv_si command

The `SCRIPTS` subdirectory contains the Tcl/Tk scripts defining the various parts of the graphical user interface:

Filename: `UV/SCRIPTS/`	Size	Description
`Command.tcl`	4672	Script for command window
`Debugger.tcl`	3539	Script for debugger window
`Document.tcl`	6191	Script for document windows
`Expression.tcl`	2028	Script for expression table window
`FSBox.tcl`	43219	Script for file selection dialog
`Program.tcl`	6098	Script for program table window
`Property.tcl`	18687	Script for property table window
`Utilities.tcl`	2356	Utility scripts

Appendix D

Additional Proofs

In this appendix we provide the remaining proofs of chapter 4.

D.1 Properties of Metric M in Theorem 12

For the predicate transformer τ and the collection M of predicates defined by

$$\llbracket \tau.X \;\equiv\; q \vee \mathbf{wltr}\,.U.X \rrbracket$$
$$[M.0 \;\equiv\; \neg\,\mathbf{wltr}\,.U^*.q]$$
$$\langle \forall i : i > 0 : [M.i \;\equiv\; \tau^i.\text{false} \;\wedge\; \neg\langle \exists j : j < i : \tau^j.\text{false}\rangle]\rangle$$

we need to show the following:

$$[\langle \exists j : 0 < j \leq i : M.j\rangle \;\equiv\; \tau^i.\text{false}] \qquad \textbf{(M0)}$$
$$[\langle \exists i : i \in \mathbf{Ord} : M.i\rangle] \qquad \textbf{(M1)}$$
$$\langle \forall i, j : i \in \mathbf{Ord} \wedge j \in \mathbf{Ord} : i \neq j \Rightarrow [\neg M.i \vee \neg M.j]\rangle \qquad \textbf{(M2)}$$

(M1) and **(M2)** establish that M is a metric, **(M0)** is an auxiliary result used both in the proof of theorem 12 and in the proof of **(M1)**.

Proof . We first recall two properties of τ from section 4.2.1:

$$[\![\mathbf{wltr}.U^*.q \ \equiv\ \langle \exists i : i \in \mathbf{Ord} : \tau^i.\text{false}\rangle]\!] \qquad \textbf{(T2)}$$

$$\langle \forall i, j : i \le j : [\![\tau^i.\text{false} \ \Rightarrow\ \tau^j.\text{false}]\!]\rangle \qquad \textbf{(T4)}$$

The proof of **(M0)** is by transfinite induction over i. For $i = 0$ the proof obligation is satisfied trivially. For any step ordinal i we observe

$$\begin{array}{ll}
& \langle \exists j : 0 < j \le i : M.j\rangle \\
\equiv & \{\text{splitting the range, } i \text{ is step ordinal, induction hypothesis}\} \\
& M.i \ \vee\ \tau^{i-1}.\text{false} \\
\equiv & \{\text{definition of } M.i\} \\
& (\tau^i.\text{false} \ \wedge\ \neg\langle \exists j : j < i : \tau^j.\text{false}\rangle) \ \vee\ \tau^{i-1}.\text{false} \\
\equiv & \{\text{predicate calculus}\} \\
& \tau^i.\text{false} \ \vee\ \tau^{i-1}.\text{false} \\
\equiv & \{\textbf{(T4)}, \text{ predicate calculus}\} \\
& \tau^i.\text{false}
\end{array}$$

For any limit ordinal i we observe

$$\begin{array}{ll}
& \langle \exists j : 0 < j \le i : M.j\rangle \\
\equiv & \{[M.i \equiv \text{false}] \text{ since } i \text{ is limit ordinal}\} \\
& \langle \exists j : 0 < j < i : M.j\rangle \\
\equiv & \{\text{predicate calculus}\} \\
& \langle \exists k, j : 0 < j \le k < i : M.j\rangle \\
\equiv & \{\text{predicate calculus}\} \\
& \langle \exists k : k < i : \langle \exists j : 0 < j \le k : M.j\rangle\rangle \\
\equiv & \{\text{induction hypothesis}\} \\
& \langle \exists k : k < i : \tau^k.\text{false}\rangle \\
\equiv & \{i \text{ is limit ordinal}\} \\
& \tau^i.\text{false}
\end{array}$$

For **(M1)** we observe

$$\begin{array}{ll}
& \langle \exists i : i \in \mathbf{Ord} : M.i\rangle \\
\equiv & \{\text{splitting the range}\}
\end{array}$$

$$
\begin{array}{ll}
& M.0 \ \vee \ \langle \exists i : i > 0 : M.i \rangle \\
\equiv & \{\text{predicate calculus}\} \\
& M.0 \ \vee \ \langle \exists i, j : 0 < j \leq i : M.j \rangle \\
\equiv & \{\text{predicate calculus}\} \\
& M.0 \ \vee \ \langle \exists i : i \in \mathbf{Ord} : \langle \exists j : 0 < j \leq i : M.j \rangle \rangle \\
\equiv & \{\mathbf{(M0)}\} \\
& M.0 \ \vee \ \langle \exists i : i \in \mathbf{Ord} : \tau^i.\,\text{false} \rangle \\
\equiv & \{\text{definition of } M.0,\ \mathbf{(T2)}\} \\
& \neg\, \mathbf{wltr}.U^*.q \ \vee \ \mathbf{wltr}.U^*.q \\
\equiv & \{\text{predicate calculus}\} \\
& \text{true}
\end{array}
$$

Finally, for **(M2)** we observe for any ordinal i with $i > 0$:

$$
\begin{array}{ll}
& M.i \wedge M.0 \\
\Rightarrow & \{\text{definition of } M.i, \text{ predicate calculus}\} \\
& \tau^i.\,\text{false} \wedge M.0 \\
\Rightarrow & \{\text{definition of } M.0,\ \mathbf{(T2)}\} \\
& \mathbf{wltr}.U^*.q \ \wedge \ \neg\, \mathbf{wltr}.U^*.q \\
\equiv & \{\text{predicate calculus}\} \\
& \text{false}
\end{array}
$$

and for any ordinals i, j with $i > j$:

$$
\begin{array}{ll}
& M.i \wedge M.j \\
\Rightarrow & \{\text{definition of } M, \text{ predicate calculus}\} \\
& \neg \langle \exists k : k < i : \tau^k.\,\text{false} \rangle \ \wedge \ \tau^j.\,\text{false} \\
\Rightarrow & \{\text{predicate calculus},\ i > j\} \\
& \neg \tau^j.\,\text{false} \ \wedge \tau^j.\,\text{false} \\
\equiv & \{\text{predicate calculus}\} \\
& \text{false}
\end{array}
$$

End of Proof.

D.2 Properties of Progress Algebras

In some of the following proofs of properties of progress algebras we use a small extension of the proof format introduced in section 2.1.3: when transforming algebraic expressions we allow $=$, $\leq$, and $\geq$ as operators relating subsequent lines of a proof in the same way as we use $\equiv$, $\Rightarrow$, and $\Leftarrow$ when relating lines of a proof consisting of transformations of logical expressions.

D.2.1 Proof of Lemma 15

Lemma 15 *In any left-handed or right-handed progress algebra $\mathcal{K}$, the subsumption relation $\leq$ defined by $U \leq V \;\equiv\; U + V = V$ is a partial order. Moreover the sequencing, alternation, and repetition operators are monotonic with respect to $\leq$, i.e., for all U, V, U' and V' in $\mathcal{K}$ with $U \leq V$ and $U' \leq V'$:*

$$UU' \leq VV' \qquad \textbf{\textit{(PrAlgSeq)}}$$

$$U + U' \leq V + V' \qquad \textbf{\textit{(PrAlgAlt)}}$$

$$U^* \leq V^* \qquad \textbf{\textit{(PrAlgStar)}}$$

Proof . Reflexivity, antisymmetry and transitivity of $\leq$ are easily shown using **(PrAlg0)** through **(PrAlg2)**[1]. Furthermore, we observe for all U, V, U' and V' in $\mathcal{K}$ with $U \leq V$ and $U' \leq V'$:

$$\begin{array}{ll} & U + U' \leq V + V' \\ \equiv & \{\text{definition of } \leq\} \\ & (U + U') + (V + V') = V + V' \\ \equiv & \{\textbf{(PrAlg0)}, \textbf{(PrAlg1)}\} \\ & (U + V) + (U' + V') = V + V' \\ \equiv & \{U + V = V,\ U' + V' = V' \text{ from assumption}\} \\ & \text{true} \end{array}$$

[1] This is an instance of the *Little Theory* in [DS90], stating that a relation $\leq$ defined in terms of a binary operator $+$ is reflexive if $+$ is idempotent, is antisymmetric if $+$ is commutative, and is transitive if $+$ is associative.

$$
\begin{array}{ll}
& UU' \;\leq\; VV' \\
\equiv & \{U + V = V,\ U' + V' = V' \text{ from assumption}\} \\
& UU' \;\leq\; (U + V)(U' + V') \\
\equiv & \{\textbf{(PrAlg7)}\} \\
& UU' \;\leq\; U(U' + V') + V(U' + V') \\
\Leftarrow & \{\textbf{(PrAlg6)},\ \textbf{(PrAlg0)}\} \\
& UU' \;\leq\; UU' + UV' + V(U' + V') \\
\equiv & \{\text{definition of } \leq\} \\
& UU' + (UU' + UV' + V(U' + V')) \;=\; UU' + UV' + V(U' + V') \\
\equiv & \{\textbf{(PrAlg0)},\ \textbf{(PrAlg2)}\} \\
& \text{true}
\end{array}
$$

For the * operator we first assume $\mathcal{K}$ to be right-handed. The case of $\mathcal{K}$ being left-handed is dealt with similarly.

$$
\begin{array}{ll}
& U^* \;\leq\; V^* \\
\Leftarrow & \{\text{transitivity of } \leq, \text{ preparing for } \textbf{(PrAlg11)}\} \\
& U^* \leq U^*V^* \;\wedge\; U^*V^* \leq V^* \\
\Leftarrow & \{\textbf{(PrAlgSeq)},\ \textbf{(PrAlg4)}\} \\
& \varepsilon \leq V^* \;\wedge\; U^*V^* \leq V^* \\
\equiv & \{\textbf{(PrAlg8)}, \text{ predicate calculus}\} \\
& U^*V^* \leq V^* \\
\Leftarrow & \{\textbf{(PrAlg11)}\} \\
& UV^* \leq V^* \\
\Leftarrow & \{\textbf{(PrAlg9)}\} \\
& UV^* \leq \varepsilon + VV^* \\
\equiv & \{\textbf{(PrAlg8)},\ \textbf{(PrAlg2)}; \text{ definition of } \leq,\ \textbf{(PrAlgAlt)}\} \\
& UV^* \leq VV^* \\
\Leftarrow & \{\textbf{(PrAlgSeq)}\} \\
& U \leq V \\
\equiv & \{\text{assumption}\} \\
& \text{true}
\end{array}
$$

End of Proof.

D.2.2 Proof of Lemma 16

Lemma 16 *In any left-handed or right-handed progress algebra $\mathcal{K}$, for all U, V, and W in $\mathcal{K}$ the following laws hold:*

$$\varepsilon + WW^* = W^* \qquad \textit{\textbf{(PrAlg13)}}$$

$$\varepsilon + W^*W = W^* \qquad \textit{\textbf{(PrAlg14)}}$$

$$W^*W^* = W^* \qquad \textit{\textbf{(PrAlg15)}}$$

$$(W^*)^* = W^* \qquad \textit{\textbf{(PrAlg16)}}$$

Furthermore, for right-handed $\mathcal{K}$ and all U, V, and W in $\mathcal{K}$:

$$V + UW \leq W \;\Rightarrow\; U^*V \leq W \qquad \textit{\textbf{(PrAlg17)}}$$

and for left-handed $\mathcal{K}$ and all U, V, and W in $\mathcal{K}$:

$$V + WU \leq W \;\Rightarrow\; VU^* \leq W \qquad \textit{\textbf{(PrAlg18)}}$$

Proof . One direction of **(PrAlg13)** and **(PrAlg14)** is given by **(PrAlg9)** and **(PrAlg10)**, respectively. The other direction follows easily from **(PrAlg8)**, **(PrAlgSeq)**, and the definition of $\leq$. For **(PrAlg15)** we observe for a right-handed progress algebra

$$\begin{array}{ll} & W^*W^* \leq W^* \\ \Leftarrow & \{\textbf{(PrAlg11)}\} \\ & WW^* \leq W^* \\ \equiv & \{\textbf{(PrAlg13)}\} \\ & WW^* \leq \varepsilon + WW^* \\ \equiv & \{\textbf{(PrAlg2)}, \text{definition of } \leq\} \\ & \text{true} \end{array}$$

and similarly for a left-handed progress algebra. We prove **(PrAlg16)** by mutual implication: for one direction we observe for a right-handed progress algebra

$$
\begin{aligned}
& (W^*)^* \ \leq \ W^* \\
\equiv \quad & \{\textbf{(PrAlg14)}\} \\
& \varepsilon + (W^*)^* W^* \ \leq \ W^* \\
\Leftarrow \quad & \{\textbf{(PrAlg2)}, \textbf{(PrAlgAlt)}, \textbf{(PrAlg8)}\} \\
& (W^*)^* W^* \ \leq \ W^* \\
\Leftarrow \quad & \{\textbf{(PrAlg11)}\} \\
& W^* W^* \ \leq \ W^* \\
\equiv \quad & \{\textbf{(PrAlg15)}\} \\
& \text{true}
\end{aligned}
$$

and similarly for a left-handed progress algebra. For the other direction we have

$$
\begin{aligned}
& W^* \ \leq \ (W^*)^* \\
\Leftarrow \quad & \{\textbf{(PrAlgStar)}\} \\
& W \ \leq \ W^* \\
\equiv \quad & \{\textbf{(PrAlg13)}\} \\
& W \ \leq \ \varepsilon + W W^* \\
\Leftarrow \quad & \{\textbf{(PrAlg8)}, \textbf{(PrAlgSeq)}, \textbf{(PrAlg2)}\} \\
& \text{true}
\end{aligned}
$$

Finally, we establish **(PrAlg17)** for a right-handed progress algebra; the proof of **(PrAlg18)** for a left-handed progress algebra is similar.

$$
\begin{aligned}
& V + UW \ \leq \ W \\
\Rightarrow \quad & \{\textbf{(PrAlg8)}, \textbf{(PrAlgAlt)}\} \\
& V \leq W \ \wedge \ UW \leq W \\
\Rightarrow \quad & \{\textbf{(PrAlg11)}\} \\
& V \leq W \ \wedge \ U^* W \leq W \\
\Rightarrow \quad & \{\textbf{(PrAlgSeq)}\} \\
& U^* V \leq W
\end{aligned}
$$

End of Proof.

D.2.3 Proof of Lemma 17

Lemma 17 *In any left-handed or right-handed progress algebra $\mathcal{K}$, for all $n \in \mathbf{N}$ with $n > 0$ and sequences W in $\mathbf{Z}_n \to \mathcal{K}$, for all permutations π of $\mathbf{Z}_n$, and for all U and V in $\mathcal{K}$ the following laws hold:*

$$UV \leq \varepsilon \quad \Rightarrow \quad U = \varepsilon \tag{PrAlg19}$$

$$UV \leq \varepsilon \quad \Rightarrow \quad V = \varepsilon \tag{PrAlg20}$$

$$U + V \leq \varepsilon \quad \Rightarrow \quad U = \varepsilon \tag{PrAlg21}$$

$$U^* \leq \varepsilon \quad \equiv \quad U = \varepsilon \tag{PrAlg22}$$

$$UU^* = U^* \tag{PrAlg23}$$

$$U^*U = U^* \tag{PrAlg24}$$

$$\langle +U : W : U \rangle \leq \langle \cdot U : W \circ \pi : U \rangle \tag{PrAlg25}$$

$$\langle +U : W : U \rangle^* = \langle \cdot U : W \circ \pi : U \rangle^* \tag{PrAlg26}$$

Proof . Due to **(PrAlg8)** it suffices to establish $\leq$ on the right-hand sides of **(PrAlg19)** through **(PrAlg22)**. For **(PrAlg19)** we observe

$$\begin{array}{ll} & U \\ = & \{\textbf{(PrAlg5)}\} \\ & U\varepsilon \\ \leq & \{\textbf{(PrAlg8)} \text{ for } V,\ \textbf{(PrAlgSeq)}\} \\ & UV \\ \leq & \{\text{antecedent}\} \\ & \varepsilon \end{array}$$

and similar for **(PrAlg20)**. For **(PrAlg21)** we observe

$$\begin{array}{ll} & U \\ \leq & \{\text{definition of } \leq,\ \textbf{(PrAlg2)}\} \\ & U + \varepsilon \\ \leq & \{\textbf{(PrAlg8)} \text{ for } V,\ \textbf{(PrAlgAlt)}\} \\ & U + V \end{array}$$

$$
\begin{array}{ll}
\leq & \{\text{antecedent}\} \\
& \varepsilon
\end{array}
$$

We prove **(PrAlg22)** by mutual implication:

$$
\begin{array}{ll}
& U \leq \varepsilon \\
\Leftarrow & \{\text{transitivity of } \leq\} \\
& U \leq U^* \\
\equiv & \{\textbf{(PrAlg13)}\} \\
& U \leq \varepsilon + UU^* \\
\Leftarrow & \{\textbf{(PrAlg8)}, \textbf{(PrAlgSeq)}, \textbf{(PrAlg2)}\} \\
& \text{true}
\end{array}
$$

and, for a right-handed progress algebra (the proof for a left-handed progress algebra is similar),

$$
\begin{array}{ll}
& \varepsilon^* \leq \varepsilon \\
\equiv & \{\textbf{(PrAlg5)}\} \\
& \varepsilon^*\varepsilon \leq \varepsilon \\
\Leftarrow & \{\textbf{(PrAlg11)}\} \\
& \varepsilon\varepsilon \leq \varepsilon \\
\equiv & \{\textbf{(PrAlg5)}\} \\
& \text{true}
\end{array}
$$

(PrAlg23) follows from **(PrAlg13)** and **(PrAlg8)**; similarly, **(PrAlg24)** follows from **(PrAlg14)** and **(PrAlg8)**.

As preparation for **(PrAlg25)** we note that for any U and V in $\mathcal{R}_F$, $U+V \leq UV$ holds by virtue of **(PrAlg8)**, **(PrAlgSeq)**, and **(PrAlg2)**.

We establish **(PrAlg25)** by induction over the length of W: the base case $|W| = 1$ is trivial; assuming that **(PrAlg25)** holds for all W of length n, we observe:

$$
\begin{array}{ll}
& \langle +U : W : U \rangle \\
= & \{\textbf{(PrAlg0)},\textbf{(PrAlg1)}\} \\
& \langle +U : W \circ \pi : U \rangle
\end{array}
$$

$$
\begin{aligned}
&= \quad \{\text{quantification over sequences}\}\\
&\quad (W \circ \pi).0 \ + \ \langle +U : \mathbf{tail}.(W \circ \pi) : U\rangle\\
&\leq \quad \{\text{induction hypothesis}\}\\
&\quad (W \circ \pi).0 \ + \ \langle \cdot U : \mathbf{tail}.(W \circ \pi) : U\rangle\\
&\leq \quad \{\text{observation above}\}\\
&\quad (W \circ \pi).0 \ \cdot \ \langle \cdot U : \mathbf{tail}.(W \circ \pi) : U\rangle\\
&= \quad \{\text{quantification over sequences}\}\\
&\quad \langle \cdot U : W \circ \pi : U\rangle
\end{aligned}
$$

As preparation for **(PrAlg26)** we note that for any U in $\mathcal{R}_F$ and any natural n, $U^n \leq U^*$, where U^n denotes the sequence of n copies of U. This fact is proved by induction over n by virtue of **(PrAlg8)**, **(PrAlgSeq)**, and **(PrAlg23)**.

We establish **(PrAlg26)** by mutual implication. One direction follows from **(PrAlg25)** and **(PrAlgStar)**, the other is shown by induction over the length of W: the base case, $|W| = 1$, is trivial; assuming that **(PrAlg26)** holds for all W of length n, we observe

$$
\begin{aligned}
&\quad \langle \cdot U : W \circ \pi : U\rangle^*\\
&\leq \quad \{\textbf{(PrAlg8)}, \textbf{(PrAlgAlt)}, \textbf{(PrAlgSeq)}\}\\
&\quad \langle \cdot U : W \circ \pi : \langle +U : W : U\rangle\rangle^*\\
&= \quad \{\text{sequence calculus}\}\\
&\quad (\langle +U : W : U\rangle^n)^*\\
&\leq \quad \{\text{observation above}, \textbf{(PrAlgStar)}\}\\
&\quad (\langle +U : W : U\rangle^*)^*\\
&= \quad \{\textbf{(PrAlg16)}\}\\
&\quad \langle +U : W : U\rangle^*
\end{aligned}
$$

End of Proof.

D.2.4 Regular Languages with Subsumption as an Example of a Progress Algebra

In this section we show that the regular languages over an alphabet Σ are an instance of a progress algebra when ordered with respect to the subsumption relation introduced in section 2.1.6. In the following we call the algebraic structure $\mathcal{R}$ and denote for any W in $\mathcal{R}$ the language of W by $\mathcal{L}.W$.

First, we have to show that the subsumption order $\leq$ is consistent with the equality defined as $U = V \;\equiv\; U \leq V \;\wedge\; V \leq U$; i.e., we have to show that $U \leq V \;\equiv\; U + V = V$. This is done by observing that for any U and V in $\mathcal{R}$

$$
\begin{array}{ll}
 & \mathcal{L}.(U+V) \;=\; \mathcal{L}.V \\
\equiv & \{\text{definition of } \mathcal{L} \text{ and } =\} \\
 & ((\mathcal{L}.U \cup \mathcal{L}.V) \;\leq\; \mathcal{L}.V) \;\wedge\; (\mathcal{L}.V \;\leq\; (\mathcal{L}.U \cup \mathcal{L}.V)) \\
\equiv & \{\text{definition of } \leq\text{, predicate calculus}\} \\
 & \mathcal{L}.U \;\leq\; \mathcal{L}.V
\end{array}
$$

Next, we have to show that all axioms of progress algebras are satisfied by $\mathcal{R}$. To this end, we establish a connection between $\mathcal{R}$ and the algebra of regular events $\mathbf{Reg}_\Sigma$. In the following we distinguish the equality in $\mathcal{R}$, written as $=$, from the equality on $\mathbf{Reg}_\Sigma$, which we write as $\equiv$, denoting the set-equality of the languages considered as sets of strings. It is obvious from the definitions of $=$, $\leq$, and $\equiv$, that for any U and V in $\mathcal{R}$, the following holds:

$$(\mathcal{L}.U \;\equiv\; \mathcal{L}.V) \quad\Rightarrow\quad (U \;=\; V)$$

This shows that all axioms, with the exception of **(PrAlg8)**, **(PrAlg11)**, and **(PrAlg12)**, are satisfied by $\mathcal{R}$ since they are satisfied by $\mathbf{Reg}_\Sigma$.

For any W in $\mathcal{R}$, $\mathcal{L}.\varepsilon \;\leq\; \mathcal{L}.W$ holds by virtue of the fact that the empty string is subsumed by any string, and that $\mathcal{L}.W$ is not empty. This shows that **(PrAlg8)** is satisfied.

Of the remaining **(PrAlg11)** and **(PrAlg12)** we show the first; the second is established similarly. We have to show that for any U and V in $\mathcal{R}$, $(\mathcal{L}.(UV) \leq \mathcal{L}.V) \Rightarrow (\mathcal{L}.(U^*V) \leq \mathcal{L}.V)$ holds. We observe

$$
\begin{aligned}
& \mathcal{L}.(U^*V) \leq \mathcal{L}.V \\
= \quad & \{\text{definition of } \mathcal{L}\} \\
& (\langle \cup i : i \in \mathrm{N} : \mathcal{L}.U^i \rangle)(\mathcal{L}.V) \leq \mathcal{L}.V \\
\equiv \quad & \{\text{set theory}\} \\
& \langle \forall i : i \in \mathrm{N} : (\mathcal{L}.U^i)(\mathcal{L}.V) \leq \mathcal{L}.V \rangle
\end{aligned}
$$

which is easily established by induction over i using the antecedent $(\mathcal{L}.U)(\mathcal{L}.V) \leq \mathcal{L}.V$.

D.3 Soundness and Completeness of the Generalized Leads-To Relation

In this section we prove the soundness and completeness of the generalized leads-to relation with respect to the operational semantics given in section 4.5.2:

Theorem 21 (Soundness and Completeness) *For any program F and regular expression W in $\mathcal{R}_F$ the deductive system defined for generalized leads-to properties is sound and relatively complete in the sense of Cook:*

$$F \models \quad p \overset{W}{\longmapsto} q \qquad \textit{iff} \qquad F \vdash \quad p \overset{W}{\longmapsto} q$$

Using the definition of the operational semantic (definition 6) and theorem 12 we rewrite the proof obligation as follows: for any W in $\mathcal{R}_F$, and any p and q in $\mathcal{P}_F$

$$
\begin{aligned}
& \langle \exists w : w \in \mathcal{S}.W : \\
& \qquad \langle \forall s, g : (s \models p \wedge \mathbf{si}.F) \wedge ((s,g) \ \mathbf{sat} \ w) : (s, \bar{g}) \models q \rangle\rangle \ \equiv \\
& [p \Rightarrow \mathbf{wltr}.W.q]
\end{aligned}
$$

We prove soundness (implication from right-to-left) and completeness (implication from left-to-right) separately. First we introduce the notion of the *canonical strategy* $C.W.q$ for W and q in section D.3.1. After having established some auxiliary lemmata in section D.3.2, we then prove the soundness in section D.3.3 using canonical strategies by showing that

$$\begin{aligned}&[p \Rightarrow \mathbf{wltr}.W.q] \;\Rightarrow \\ &\langle \forall s, g : (s \models p \wedge \mathbf{si}.F) \wedge ((s,g)\ \mathbf{sat}\ C.W.q) : (s, \bar{g}) \models q \rangle.\end{aligned}$$

Finally, in section D.3.4 we prove the completeness by establishing the contrapositive

$$\begin{aligned}&\neg[p \Rightarrow \mathbf{wltr}.W.q] \;\Rightarrow \\ &\langle \forall w : w \in \mathcal{S}.W : \langle \exists s, g : (s \models p \wedge \mathbf{si}.F) \wedge ((s,g)\ \mathbf{sat}\ w) : (s, \bar{g}) \not\models q \rangle\rangle.\end{aligned}$$

D.3.1 Canonical Strategy

For a regular expression W in $\mathcal{R}_F$ and a goal predicate q in $\mathcal{P}_F$ we define the canonical strategy $C.W.q$ inductively over the structure of W as a specific element of $\mathcal{S}.W$ as follows. For all α in $F.A$ and all U, V in $\mathcal{R}_F$:

$$\begin{aligned}
C.\varepsilon.q &= \mathbf{eps} \\
C.\alpha.q &= \mathbf{act}.\alpha \\
C.(UV).q &= \mathbf{seq}.(C.U.(\mathbf{wltr}.V.q), C.V.q) \\
C.(U+V).q &= \mathbf{alt}.(\mathbf{wltr}.U.q, C.U.q, C.V.q) \\
C.U^*.q &= \mathbf{star}.(q, C.U.q)
\end{aligned}$$

From the proofs of the completeness theorem in section D.3.4 below we obtain the result that the canonical strategy is a *most general* strategy; i.e., if $F \models p \overset{W}{\longmapsto} q$ can be established with some strategy w in $\mathcal{S}.W$, then it can be established with $C.W.q$ as well.

D.3.2 Auxiliary Lemmata

Before we embark on the proofs of the soundness and completeness theorem we state and prove a few lemmata for later use. We start with a lemma that establishes a (conditional) stability result for $\mathbf{wltr}.W.q$ predicates:

Lemma 22 *For any α in $F.A$, q in $\mathcal{P}_F$, s in $F.S$, and W in $\mathcal{R}_F$:*

$$(s \models \neg q \wedge \mathbf{wltr}.W.q) \quad \Rightarrow \quad (\alpha.s \models \mathbf{wltr}.W.q)$$

Proof . The proof proceeds by induction on the structure of W. We observe for any U and V in $\mathcal{R}_F$ and any β in $F.A$:

case $W = \varepsilon$:

The antecedent is false due to **(wltrEps)**, hence the implication is satisfied trivially.

case $W = \beta$:

$$\begin{array}{ll} & s \models \neg q \wedge \mathbf{wltr}.\beta.q \\ \Rightarrow & \{\mathbf{(wltrAct)}\} \\ & s \models \neg q \wedge (q \vee (\mathbf{wco}.(q \vee \mathbf{wltr}.\beta.q) \wedge \mathbf{wp}.\beta.q)) \\ \Rightarrow & \{\text{predicate calculus}\} \\ & s \models \mathbf{wco}.(q \vee \mathbf{wltr}.\beta.q) \\ \Rightarrow & \{\text{definition of } \mathbf{wco}\text{, property of } \mathbf{wp}\text{, } \mathbf{(wltrWeaken)}\} \\ & \alpha.s \models \mathbf{wltr}.\beta.q \end{array}$$

case $W = UV$:

We consider two cases: for $s \not\models \mathbf{wltr}.V.q$ we observe

$$\begin{array}{ll} & s \models \neg q \wedge \mathbf{wltr}.(UV).q \\ \equiv & \{s \models \neg\,\mathbf{wltr}.V.q,\ \mathbf{(wltrWeaken)},\ \mathbf{(wltrSeq)}\} \\ & s \models \neg\,\mathbf{wltr}.V.q \ \wedge \ \mathbf{wltr}.U.(\mathbf{wltr}.V.q) \end{array}$$

$$\Rightarrow \quad \{\text{induction hypothesis for } U \text{ with } \mathbf{wltr}.V.q \text{ for } q, \ \mathbf{(wltrSeq)}\}$$
$$\alpha.s \models \mathbf{wltr}.(UV).q$$

If, on the other hand, $s \models \mathbf{wltr}.V.q$ holds, we observe

$$s \models \neg q \wedge \mathbf{wltr}.(UV).q$$
$$\equiv \quad \{s \models \mathbf{wltr}.V.q, \ \mathbf{(wltrSeq)}, \ \mathbf{(wltrWeaken)}\}$$
$$s \models \neg q \wedge \mathbf{wltr}.V.q$$
$$\Rightarrow \quad \{\text{induction hypothesis for } V\}$$
$$\alpha.s \models \mathbf{wltr}.V.q$$
$$\Rightarrow \quad \{\mathbf{(wltrWeaken)}, \ \mathbf{(wltrSeq)}\}$$
$$\alpha.s \models \mathbf{wltr}.(UV).q$$

case $W = U + V$:

$$s \models \neg q \wedge \mathbf{wltr}.(U+V).q$$
$$\equiv \quad \{\mathbf{(wltrAlt)}, \text{ definition of } \models, \text{ predicate calculus}\}$$
$$(s \models \neg q \wedge \mathbf{wltr}.U.q) \ \vee \ (s \models \neg q \wedge \mathbf{wltr}.V.q)$$
$$\Rightarrow \quad \{\text{induction hypothesis, twice}\}$$
$$(\alpha.s \models \mathbf{wltr}.U.q) \ \vee \ (\alpha.s \models \mathbf{wltr}.V.q)$$
$$\equiv \quad \{\text{definition of } \models, \ \mathbf{(wltrAlt)}\}$$
$$\alpha.s \models \mathbf{wltr}.(U+V).q$$

case $W = U^*$:

From $s \models \neg q \wedge \mathbf{wltr}.U^*.q$ we obtain by virtue of **(wltrStar)** that there is an ordinal i satisfying $s \models \neg q \wedge \tau^i.\text{false}$, where τ is defined by $[\![\tau.X \equiv q \vee \mathbf{wltr}.U.X]\!]$. We prove by induction over the ordinals that for all ordinals i

$$(s \models \neg q \wedge \tau^i.\text{false}) \ \Rightarrow \ (\alpha.s \models \tau^i.\text{false}) \ .$$

This assertion, together with the observation about the existence of a suitable ordinal and with with **(wltrStar)**, establishes the required result. For $i = 0$ and $i = 1$ the antecedent is false and the implication is satisfied trivially. If i is a limit ordinal, we have

$$
\begin{array}{ll}
 & s \models \neg q \wedge \tau^{i}.\,\text{false} \\
\equiv & \{i \text{ is limit ordinal}\} \\
 & s \models \neg q \wedge \langle \exists l : l < i : \tau^{l}.\,\text{false}\rangle \\
\equiv & \{\text{definition of } \models, \text{ predicate calculus}\} \\
 & \langle \exists l : l < i : s \models \neg q \wedge \tau^{l}.\,\text{false}\rangle \\
\Rightarrow & \{\text{induction hypothesis for witness } l,\ \tau \text{ is monotonic}\} \\
 & \alpha.s \models \tau^{i}.\,\text{false}
\end{array}
$$

If i is a step ordinal i greater than 1, we observe

$$
\begin{array}{ll}
 & s \models \neg q \wedge \tau^{i}.\,\text{false} \\
\equiv & \{\text{definition of } \tau,\ i \text{ is step ordinal, predicate calculus}\} \\
 & s \models \neg q \wedge \mathbf{wltr}.U.(\tau^{i-1}.\,\text{false})
\end{array}
$$

Again, we consider two cases: for $s \not\models \tau^{i-1}.\,\text{false}$ we observe

$$
\begin{array}{ll}
 & s \models \neg q \wedge \mathbf{wltr}.U.(\tau^{i-1}.\,\text{false}) \\
\equiv & \{s \models \neg\tau^{i-1}.\,\text{false},\ [\![q \Rightarrow \tau^{i-1}.\,\text{false}]\!]\} \\
 & s \models \neg\tau^{i-1}.\,\text{false} \wedge \mathbf{wltr}.U.(\tau^{i-1}.\,\text{false}) \\
\Rightarrow & \{\text{induction hypothesis for } U \text{ with } \tau^{i-1}.\,\text{false for } q\} \\
 & \alpha.s \models \mathbf{wltr}.U.(\tau^{i-1}.\,\text{false}) \\
\Rightarrow & \{\text{definition of } \tau,\ (\mathbf{wltrStar})\} \\
 & \alpha.s \models \tau^{i}.\,\text{false}
\end{array}
$$

whereas for $s \models \tau^{i-1}.\,\text{false}$ we have

$$
\begin{array}{ll}
 & s \models \neg q \wedge \mathbf{wltr}.U.(\tau^{i-1}.\,\text{false}) \\
\equiv & \{s \models \tau^{i-1}.\,\text{false},\ (\mathbf{wltrWeaken})\} \\
 & s \models \neg q \wedge \tau^{i-1}.\,\text{false} \\
\Rightarrow & \{\text{induction hypothesis for } i-1,\ \tau \text{ is monotonic}\} \\
 & \alpha.s \models \tau^{i}.\,\text{false}
\end{array}
$$

End of Proof.

The intermediate assertion we established for the repetition case will be referred to later on; therefore, we restate it as a corollary as follows:

Corollary 23 *For any α in $F.A$, q in $\mathcal{P}_F$, s in $F.S$, W in $\mathcal{R}_F$, and ordinal i:*

$$(s \models \neg q \wedge \tau^i.\,\mathrm{false}) \quad \Rightarrow \quad (\alpha.s \models \tau^i.\,\mathrm{false}),$$

where τ is defined by $[\![\tau.X \equiv q \vee \mathbf{wltr}\,.W.X]\!]$.

The next lemma characterizes $\mathbf{wltr}\,.\alpha.q$ in terms of program executions:

Lemma 24 *For any α in $F.A$, q in $\mathcal{P}_F$, and s in $F.S$:*

$$\langle \forall x : x \in (F.A)^* : (s, x\alpha) \models q \rangle \quad \equiv \quad s \models \mathbf{wltr}\,.\alpha.q$$

Proof . The proof proceeds by mutual implication. Assuming $s \models \mathbf{wltr}\,.\alpha.q$, we define for any x consisting of k actions, $x.0$ through $x.(k-1)$, a sequence t of states as follows: $t.0 = s$, $t.(i+1) = (x.i).(t.i)$ for all i with $0 \leq i < k$, and $t.(k+1) = \alpha.(t.k)$. We show that there is an index i, $0 \leq i \leq k+1$, such that $t.i \models q$ by establishing that $\langle \forall i : i < k+1 : t.i \not\models q \rangle \Rightarrow (t.(k+1) \models q)$:

$$\begin{array}{ll}
& \langle \forall i : i < k+1 : t.i \not\models q \rangle \\
\Rightarrow & \{t.0 \models \mathbf{wltr}\,.\alpha.q, \text{ lemma 22 } k\text{-times}\} \\
& t.k \models \neg q \wedge \mathbf{wltr}\,.\alpha.q \\
\Rightarrow & \{(\mathbf{wltrAct}), \text{ predicate calculus}\} \\
& t.k \models \mathbf{wp}\,.\alpha.q \\
\Rightarrow & \{\text{property of } \mathbf{wp}, \text{ definition of } t.(k+1)\} \\
& t.(k+1) \models q
\end{array}$$

Conversely, we assume that $\langle \forall x : x \in (F.A)^* : (s, x\alpha) \models q \rangle$ holds. We define a predicate r by

$$(t \models r) \;\equiv\; \langle \forall x : x \in (F.A)^* : ((t, x) \not\models q) \;\Rightarrow\; (\alpha.(x.t) \models q) \rangle$$

Clearly, $s \models r$ holds. Next, we observe for all states t (the range for quantifications of x is understood to be $x \in (F.A)^*$):

$$
\begin{array}{ll}
& t \models r \\
\Rightarrow & \{\text{definition of } r, \text{ predicate calculus}\} \\
& (t \models q) \;\vee\; ((t \not\models q) \wedge \langle \forall x :: ((t,x) \not\models q) \;\Rightarrow\; (\alpha.(x.t) \models q)\rangle) \\
\Rightarrow & \{\text{predicate calculus, instantiating } \langle\rangle \text{ and } x\beta \text{ for } x\} \\
& (t \models q) \;\vee\; ((\alpha.t \models q) \wedge \langle \forall \beta, x : \beta \in F.A : ((t, x\beta) \not\models q) \;\Rightarrow\; (\alpha.(x\beta.t) \models q)\rangle) \\
\equiv & \{\text{predicate calculus, definition of } r\} \\
& (t \models q) \;\vee\; ((\alpha.t \models q) \wedge \langle \forall \beta : \beta \in F.A : \beta.t \models r\rangle) \\
\equiv & \{\text{definition of } \mathbf{wco}, \text{ property of } \mathbf{wp}\} \\
& (t \models q) \;\vee\; ((t \models \mathbf{wp}.\alpha.q) \wedge (t \models \mathbf{wco}.r)) \\
\Rightarrow & \{\mathbf{wco} \text{ is monotonic, predicate calculus, definition of } \models\} \\
& t \models q \vee (\mathbf{wco}.(r \vee q) \wedge \mathbf{wp}.\alpha.q)
\end{array}
$$

This establishes by virtue of **(wltrAct)** that r is a solution of the equation defining $\mathbf{wltr}.\alpha.q$. By the theorem of Knaster-Tarski we conclude that $[\![r \Rightarrow \mathbf{wltr}.\alpha.q]\!]$, which together with $s \models r$ establishes $s \models \mathbf{wltr}.\alpha.q$ as required.

End of Proof.

Lemma 25 *For any W and R in $\mathcal{R}_F$, such that R^* is a subexpression of W, and for any q in $\mathcal{P}_F$, there is an X in $\mathcal{R}_F$ such that the sub-strategy of $C.W.q$ corresponding to R^* has the form* $\mathbf{star}.(\mathbf{wltr}.X.q, R)$.

Proof . The required result follows from showing that for all W and R in $\mathcal{R}_F$ and all q and t in $\mathcal{P}_F$:

$$\mathbf{star}.(t, R) \text{ is a sub-strategy of } C.W.q \;\Rightarrow\; \langle \exists X : X \in \mathcal{R}_F : [\![t \equiv \mathbf{wltr}.X.q]\!]\rangle$$

This is proved by a straightforward induction over the structure of W, which is left as an exercise.

End of Proof.

D.3.3 Soundness

By virtue of the equivalence $[p] \;\equiv\; \langle \forall s : s \models \mathbf{si}.F : s \models p \rangle$ we can rephrase our proof obligation as follows:

$$\begin{array}{l} \langle \forall s : s \models p \wedge \mathbf{si}.F : \\ \qquad\qquad (s \models \mathbf{wltr}.W.q) \;\Rightarrow \langle \forall g : (s,g) \;\mathbf{sat}\; C.W.q : (s,\bar{g}) \models q \rangle \rangle . \end{array}$$

Proof . The proof proceeds by induction over the structure of W. We observe for all α in $F.A$, U and V in $\mathcal{R}_F$, p and q in $\mathcal{P}_F$, and s in $F.S$ satisfying $s \models p \wedge \mathbf{si}.F$:

case $W = \varepsilon$:

$$\begin{array}{ll} & (s,g) \;\mathbf{sat}\; C.\varepsilon.q \\ \equiv & \{\text{definition of } C.\varepsilon.q \text{, definition of } \mathbf{sat}\, \} \\ & g = \langle\rangle \end{array}$$

and, therefore,

$$\begin{array}{ll} & (s \models \mathbf{wltr}.\varepsilon.q) \;\Rightarrow\; \langle \forall g : (s,g) \;\mathbf{sat}\; C.\varepsilon.q : (s,\bar{g}) \models q \rangle \\ \equiv & \{\text{from above, } \mathbf{(wltrEps)}\} \\ & (s \models q) \;\Rightarrow\; ((s,\bar{\langle\rangle}) \models q) \\ \equiv & \{\text{predicate calculus}\} \\ & \text{true} \end{array}$$

case $W = \alpha$:

$$\begin{array}{ll} & (s,g) \;\mathbf{sat}\; C.\alpha.q \\ \equiv & \{\text{definition of } C.\alpha.q \text{, definition of } \mathbf{sat}\, \} \\ & \langle \exists x : x \in (F.A)^* : g = \langle x\alpha \rangle \rangle \end{array}$$

and, therefore,

$$\begin{array}{ll} & \langle \forall g : (s,g) \;\mathbf{sat}\; C.\alpha.q : (s,\bar{g}) \models q \rangle \\ \equiv & \{\text{from above, predicate calculus}\} \end{array}$$

$$\langle \forall x : x \in (F.A)^* : (s, x\alpha) \models q \rangle$$
$$\equiv \quad \{\text{lemma 24, antecedent, } s \models \mathbf{wltr}.\alpha.q\}$$
$$\text{true}$$

case $W = UV$:

For any game g for which (s,g) **sat** $C.(UV).q$ holds, we obtain from the definitions of $C.(UV).q$ and **sat** that there exist games e and f satisfying the following conditions:

$$g = e +\!\!+ f \qquad \textbf{(SC0)}$$
$$(s, e) \ \mathbf{sat}\ C.U.(\mathbf{wltr}.V.q) \qquad \textbf{(SC1)}$$
$$\mathbf{finite}.\bar{e} \ \Rightarrow\ (\bar{e}.s, f)\ \mathbf{sat}\ C.V.q \qquad \textbf{(SC2)}$$

Assuming that $s \models \mathbf{wltr}.(UV).q$, we need to show that $(s, \bar{g}) \models q$. We do this by considering two cases depending on whether $\bar{e}$ is finite or infinite:

First we consider the case that **finite** $.\bar{e}$ holds. Since $s \models \mathbf{wltr}.U.(\mathbf{wltr}.V.q)$ by **(wltrSeq)**, we have by virtue of **(SC1)** and the induction hypothesis for U (instantiating true for p and $\mathbf{wltr}.V.q$ for q):

$$(s, \bar{e}) \models \mathbf{wltr}.V.q \qquad \textbf{(SC3)}$$

If $(s, \bar{e}) \models q$ holds, we have $(s, \bar{g}) \models q$ because of **(SC0)**. Assuming $(s, \bar{e}) \not\models q$ we obtain from **(SC3)** that some state in the run $(s, \bar{e})$ satisfies $\neg q \wedge \mathbf{wltr}.V.q$. By repeated application of lemma 22 we obtain that $\bar{e}.s \models \mathbf{wltr}.V.q$. Together with **(SC2)** we have by virtue of the induction hypothesis for V (instantiating the reachable state $\bar{e}.s$ for s and true for p):

$$(\bar{e}.s, f) \models q$$

With the help of **(SC0)** we conclude $(s, \bar{g}) \models q$.

For the second case we assume $\bar{e}$ to be infinite. By **(SC1)**, e follows the strategy $C.U.(\mathbf{wltr}.V.q)$. There is only one way for $\bar{e}$ to be infinite: U contains a subexpression of the form R^*, the canonical sub-strategy corresponding to R^* is of the form $\mathbf{star}.(t, R)$ for some predicate t in $\mathcal{P}_F$, and from some point on e is made up from an infinite series of sub-games each of which follows the canonical strategy for R and ends in a state satisfying $\neg t$.

We define the infinite sequence E of runs by letting $E.i$ for natural i be the prefix of $\bar{e}$ up to the beginning of the R-sub-game number i (counting from 0). Since R is a subexpression of U, we have by lemma 25 that the sub-strategy for R has the form $\mathbf{star}.(\mathbf{wltr}.(XV).q, R)$ for some X in $\mathcal{R}_F$. As before, we have by virtue of **(SC1)** and the induction hypothesis for U:

$$(s, \bar{e}) \models \mathbf{wltr}.V.q$$

Let y be a prefix of $\bar{e}$ such that $y.s \models \mathbf{wltr}.V.q$ and let j be the smallest natural number for which y is a prefix of $E.j$. We show that $(s, E.j) \models q$: if this were not the case, we would have $y.s \models \neg q \wedge \mathbf{wltr}.V.q$. Repeated application of lemma 22 would establish $(E.j).s \models \mathbf{wltr}.V.q$, which together with **(wltrWeaken)** and **(wltrSeq)** would yield $(E.j).s \models \mathbf{wltr}.(XV).q$, contradicting the non-termination of e at the beginning of sub-game j. Hence, we have $(s, E.j) \models q$ and, by the definition of $E.j$ and by **(SC0)**, that $(s, \bar{g}) \models q$ holds as required.

case $W = U + V$:

Any game g, for which $(s, g)\ \mathbf{sat}\ C.(U{+}V).q$ holds, satisfies the following conditions by virtue of the definitions of $C.(U + V).q$ and **sat** :

$$(s \models \mathbf{wltr}.U.q) \;\Rightarrow\; ((s, g)\ \mathbf{sat}\ C.U.q) \qquad \textbf{(SC4)}$$

$$(s \not\models \mathbf{wltr}.U.q) \;\Rightarrow\; ((s, g)\ \mathbf{sat}\ C.V.q) \qquad \textbf{(SC5)}$$

We consider two cases depending on whether $\mathbf{wltr}.U.q$ holds in s or not: if $s \models$

$\mathbf{wltr}.U.q$, we obtain from **(SC4)** and the induction hypothesis for U that $(s, \bar{g}) \models q$. Similarly, if $s \not\models \mathbf{wltr}.U.q$, we obtain form the assumption $s \models \mathbf{wltr}.(U+V).q$, **(wltrAlt)**, **(SC5)** and the induction hypothesis for V that $(s, \bar{g}) \models q$.

case $W = U^*$:

From $s \models \mathbf{wltr}.U^*.q$ and **(wltrStar)** we conclude that there is a least ordinal l such that $s \models \tau^l.\text{false}$, where the predicate transformer τ is defined as $[\![\tau.X \equiv q \vee \mathbf{wltr}.U.X]\!]$.

In the following we define sequences k of ordinals, t of reachable states in $F.S$, e and f of games in $\mathcal{G}_F$, and E of finite runs in $(F.A)^*$. We define

$$\begin{aligned} k.0 &= l \\ t.0 &= s \\ f.0 &= g \\ E.0 &= \langle\rangle \end{aligned}$$

and maintain the following conditions for all natural numbers i, for which the respective elements are defined:

$$t.i \models \tau^{k.i}.\text{false} \qquad \textbf{(SC6)}$$

$$(t.i, f.i)\ \mathbf{sat}\ C.U^*.q \qquad \textbf{(SC7)}$$

$$E.i \mathbin{+\!\!+} \bar{f}.i = \bar{g} \qquad \textbf{(SC8)}$$

$$t.i = (E.i).s \qquad \textbf{(SC9)}$$

Clearly, all of these conditions are satisfied for $i = 0$ by virtue of the above definitions. We now describe the construction of later elements of the various sequences. The construction will establish $(s, \bar{g}) \models q$ eventually.

Following the definition of **sat** we consider two cases: if $t.i \models q$ holds then we have, by virtue of **(SC8)** and **(SC9)**, that $(s, \bar{g}) \models q$ and we are done.

We now assume $t.i \not\models q$. From the definition of **sat** we know that there exist games $e.i$ and $f.(i+1)$ satisfying the following conditions:

$$f.i = e.i +\!\!+ f.(i+1) \qquad \textbf{(SC10)}$$

$$(t.i, e.i) \ \textbf{sat} \ C.U.q \qquad \textbf{(SC11)}$$

$$\textbf{finite}.(\bar{e.i}) \ \Rightarrow \ ((\bar{e.i}).(t.i), f.(i+1)) \ \textbf{sat} \ C.U^*.q \qquad \textbf{(SC12)}$$

We, again, consider two cases depending on whether $\bar{e.i}$ is finite or infinite. For infinite $\bar{e.i}$ an argument similar to the one for the case of sequencing ($W = UV$) above establishes that $(t.i, \bar{e.i}) \models q$ and we are done by virtue of **(SC10)** and **(SC8)**.

For finite $\bar{e.i}$ we either have $(t.i, \bar{e.i}) \models q$ and we are done, or we have $(t.i, \bar{e.i}) \not\models q$. In this latter case we define $E.(i+1) = E.i +\!\!+ \bar{e.i}$ and define $t.(i+1)$ according to **(SC9)**. With this definition, **(SC8)** is maintained for $i+1$. **(SC12)** asserts **(SC7)** for $i+1$ as well. From **(SC6)** we conclude that $t.i \models \textbf{wltr}.U^*.q$; together with the fact that $(t.i, \bar{e.i}) \not\models q$ we obtain by repeated application of lemma 22 that $t.(i+1) \models \textbf{wltr}.U^*.q$ as well. Hence, there is a least ordinal $k.(i+1)$ for which $t.(i+1) \models \tau^{k.(i+1)}.\text{false}$, thereby establishing **(SC6)** for $i+1$.

Next, we show that $k.(i+1) < k.i$. With this fact we see that, due to the well-foundedness of the ordinals, the above construction can be done only finitely many times and has to terminate with one of the cases establishing the goal $(s, \bar{g}) \models q$.

In order to establish that $k.(i+1) < k.i$, we summarize some properties of elements of our construction:

$$\textbf{finite}.(\bar{e.i}) \qquad \textbf{(SC13)}$$

$$(t.i, \bar{e.i}) \not\models q \qquad \textbf{(SC14)}$$

$$((\bar{e.i}).(t.i)) \models \tau^{k.(i+1)}.\text{false} \qquad \textbf{(SC15)}$$

We prove that $k.(i+1) < k.i$ by induction over the ordinals. First, we note that $k.i > 0$ by **(SC6)**. Since $k.i$ is the least ordinal satisfying **(SC6)**, it is not a limit ordinal. Hence $k.i$ is a step ordinal. We observe by starting with **(SC6)**

$t.i \models \tau^{k.i}.\,\text{false}$

$\Rightarrow$ $\{k.i$ is step ordinal, definition of τ, **(SC14)**$\}$

$t.i \models \mathbf{wltr}\,.U.(\tau^{k.i-1}.\,\text{false})$

$\Rightarrow$ $\{$**(SC11)**, induction hypothesis for $U\}$

$(t.i, \bar{e.i}) \models \tau^{k.i-1}.\,\text{false}$

$\Rightarrow$ $\{$**(SC13)**, **(SC14)**, repeated application of corollary 23$\}$

$(\bar{e.i}).(t.i) \models \tau^{k.i-1}.\,\text{false}$

$\Rightarrow$ $\{$**(SC15)**, minimality of $k.(i+1)\}$

$k.(i+1) < k.i$

End of Proof.

D.3.4 Completeness

By virtue of the equivalence $[p] \;\equiv\; \langle \forall s : s \models \mathbf{si}.F : s \models p \rangle$ we can rephrase our proof obligation as follows:

$$\langle \exists s : s \models p \wedge \mathbf{si}\,.F : s \not\models \mathbf{wltr}\,.W.q \rangle \;\Rightarrow$$
$$\langle \forall w : w \in \mathcal{S}.W : \langle \exists s, g : (s \models p \wedge \mathbf{si}\,.F) \wedge ((s,g)\ \mathbf{sat}\ w) : (s, \bar{g}) \not\models q \rangle \rangle.$$

This proof obligation is discharged by showing that for any strategy w in $\mathcal{S}.W$, any s in $F.S$, and any q in $\mathcal{P}_F$:

$$(s \models \neg\,\mathbf{wltr}\,.W.q) \;\Rightarrow\; \langle \exists g : (s,g)\ \mathbf{sat}\ w : (s, \bar{g}) \not\models q \rangle$$

which we establish in the following for all W in $\mathcal{R}_F$.

Proof . The proof proceeds by induction over the structure of W. We observe for all q in $\mathcal{P}_F$, all α in $F.A$, and all U and V in $\mathcal{R}_F$:

case $W = \varepsilon$:

We choose $g = \langle\rangle$ which satisfies **eps**, the only strategy in $\mathcal{S}.\varepsilon$. We have

$(s, \bar{g}) \not\models q$

$$\equiv \quad \{g = \langle\rangle\}$$
$$(s \models \neg q)$$
$$\Leftarrow \quad \{\text{antecedent, } \mathbf{(wltrEps)}\}$$
$$\text{true}$$

case $W = \alpha$ for some $\alpha \in F.A$:

Careful inspection of the corresponding case in the soundness proof reveals that completeness has been established there as well (due to the equivalence in lemma 24).

case $W = UV$:

Any strategy w in $\mathcal{S}.W$ has the form $\mathbf{seq}.(u, v)$ for two strategies u in $\mathcal{S}.U$ and v in $\mathcal{S}.V$. Since $s \models \neg\,\mathbf{wltr}.U.(\mathbf{wltr}.V.q)$ by **(wltrSeq)**, there is by virtue of the induction hypothesis for U a game e satisfying (s, e) **sat** u, such that $(s, \bar{e}) \not\models \mathbf{wltr}.V.q$.

If $\bar{e}$ is infinite we have by **(wltrWeaken)** that $(s, \bar{e}) \not\models q$, and by the definition of **sat** that (s, e) **sat** w. Hence we choose $g = e$.

If $\bar{e}$ is finite, on the other hand, then from $(s, \bar{e}) \not\models \mathbf{wltr}.V.q$ it follows that $\bar{e}.s \models \neg\,\mathbf{wltr}.V.q$. By virtue of the induction hypothesis for V there is a game f satisfying $(\bar{e}.s, f)$ **sat** v, such that $(\bar{e}.s, \bar{f}) \not\models q$. We choose $g = e +\!\!+ f$ which satisfies both (s, g) **sat** w and $(s, \bar{g}) \not\models q$.

case $W = U + V$:

Any strategy w in $\mathcal{S}.W$ has the form $\mathbf{alt}.(t, u, v)$ for a predicate t and strategies u in $\mathcal{S}.U$ and v in $\mathcal{S}.V$. From the antecedent and from **(wltrAlt)** it follows that $s \models \neg\,\mathbf{wltr}.U.q \;\wedge\; \neg\,\mathbf{wltr}.V.q$. By virtue of the induction hypothesis for U, there is a game e with (s, e) **sat** u such that $(s, \bar{e}) \not\models q$. Similarly, there is a game f with

(s, f) **sat** v such that $(s, \bar{f}) \not\models q$. If $s \models t$ we choose $g = e$, otherwise we choose $g = f$.

case $W = U^*$:

We construct a game g for which $(s, \bar{g}) \not\models q$ holds as follows: starting with $f.0 = \langle\rangle$ we define games $f.i$ for natural i while maintaining the following two conditions:

$$(s, \bar{f.i}) \not\models q \qquad \textbf{(SC16)}$$

$$(\bar{f.i}).s \models \neg\, \mathbf{wltr}.U^*.q \qquad \textbf{(SC17)}$$

Since $s \models \neg\, \mathbf{wltr}.U^*.q$, both **(SC16)** and **(SC17)** are satisfied for $i = 0$. Any strategy w in $\mathcal{S}.W$ has the form $\mathbf{star}.(t, u)$ for a predicate t and a strategy u in $\mathcal{S}.U$. There are two possibilities for a game execution from state $(\bar{f.i}).s$ following strategy w: either $s \models t$, in which case the game terminates, or $s \not\models t$, in which case a game following strategy u is played. In the first case, we simply choose $g = f.i$, for which **(SC16)** establishes that $(s, \bar{g}) \not\models q$.

In the non-terminating case we have $(\bar{f.i}).s \models \neg\, \mathbf{wltr}.U.(\mathbf{wltr}.U^*.q)$, because $U^* = UU^*$ by **(PrAlg23)**. By virtue of the induction hypothesis for U, instantiating $\mathbf{wltr}.U^*.q$ for q, there exists a game e satisfying $((\bar{f.i}).s, e)$ **sat** u such that $((\bar{f.i}).s, e) \not\models \mathbf{wltr}.U^*.q$, which implies by virtue of **(wltrWeaken)** that $((\bar{f.i}).s, e) \not\models q$. We define $f.(i+1) = f.i \mathbin{+\!\!+} e$. Clearly, **(SC16)** is satisfied for $i+1$. If $\bar{e}$ is infinite we choose $g = f.(i+1)$ and are done. If $\bar{e}$ is finite we have from $((\bar{f.i}).s, e) \not\models \mathbf{wltr}.U^*.q$ that $\bar{e}.((\bar{f.i}).s) \models \neg\, \mathbf{wltr}.U^*.q$ establishing **(SC17)** for $i+1$. Hence we can repeat the construction until it terminates with one of the cases above, or we choose for g the limit of the $f.i$. In either case we have $(s, \bar{g}) \not\models q$ as required.

End of Proof.

Bibliography

[Ada95] W. Adams. Concurrent programming witha single thread of control. PhD dissertation proposal, The University of Texas at Austin, April 1995.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[Bak92] H. G. Baker. The treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3), March 1992.

[BBR90] K. S. Brace, R. E. Bryant, and R. L. Rudell. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.

[BC95] R. E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Design Automation conference*, pages 535–541, 1995.

[BCM91] J. R. Burch, E. M. Clarke, and K. M. McMillan. Representing circuits more efficiently in symbolic model checking. In *Proccedings of the 28th Design Automation Conference 1991*, pages 403–407, 1991.

[Bra93] J. C. Bradfield. *Verifying Temporal properties of Systems.* Progress in Temporal Computer Science. Birkhäuser, 1993.

[Bro93] M. Broy. *Program Design Calculi*, chapter (Inter-)Action Refinement: The Easy Way. Springer Verlag, 1993.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computing*, (6), 1986.

[Car94] A. Carruth. Real-time UNITY. Technical Report TR94-10, University of Texas at Austin, Austin, Texas, April 1994.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, June 1989.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Sience*, May 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, (2), 1986.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, (5), 1994.

[CK93] P. Collette and E. Knapp. A compositional proof system for UNITY based on rely/guarantee conditions. Submitted to the IFIP Workshop on Programming Concepts, Methods and Calculi, November 1993.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison Wesley, 1988.

[CM90] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *IEEE Int. Conference on CAD*, November 1990.

[Coh93] E. Cohen. *Modular Progress Proofs of Asynchronous Programs*. PhD thesis, The University of Texas at Austin, 1993.

[Coh96] E. Cohen. Personal communications. 1996.

[Col93] P. Collette. Application of the composition principle to UNITY-like specifications. In *Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Sience*. Springer Verlag, 1993.

[Con71] J. H. Conway. *Regular algebra and finite machines*. Chapman and Hall, London, 1971.

[Coo78] S. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.

[CR90] H. C. Cunningham and G.-C. Roman. A UNITY-style programming logic for a shared dataspace language. *IEEE Transactions on Parallel and Distributed Systems*, 1(3), 1990.

[CWB94] J. Cuellar, I. Wildgruber, and D. Barnard. The temporal logic of transitions. In *Formal Methods Europe (Barcelona, Spain)*, 1994.

[CZ96] E. M. Clarke and X. Zhao. Word-level symbolic model checking - a new approach for verifying arithmetic circuits. to be published, 1996.

[DF90] A. Dappert-Farquhar. A correction on "a family of 2-process mutual exclusion algorithms: Notes on UNITY: 13-90". *Notes on UNITY*, (22), 1990.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy, and the formal derivation of programs. *Communications of the ACM*, 8:453–457, 1975.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[Dij95] R. M. Dijkstra. DUALITY: A simple formalism for the analysis of UNITY. *Formal Aspects of Computing*, 7(4):353–388, 1995.

[DS90] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Text and Monographs in Computer Science. Springer Verlag, 1990.

[EL85] E. A. Emerson and C. L. Lei. Modalities for model checking: Branching time strikes back. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1985.

[EN95] E. A. Emerson and K. Namjoshi. Reasoning about rings. In *Proceedings of the 22th Annual ACM Symposium on Principles of Programming Languages*, 1995.

[Fil94] T. Filkorn. Personal communications. 1994.

[Fra86] N. Francez. *Fairness.* Springer Verlag, New York, 1986.

[GF93] A. Gupta and A. L. Fisher. Parametric circuit representation using inductive boolean functions. In *Proceedings of the 5th Conference on Computer Aided Verification.* LNCS 697, 1993.

[GL94] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[Gol92] D. M. Goldschlag. *Mechanically verifying concurrent programs.* PhD thesis, The University of Texas at Austin, 1992.

[Gro94] H. D. Group. HSIS: a BDD-based environment for formal verification. Technical report, University of California at Berkeley, 1994.

[HD93] A. J. Hu and D. L. Dill. Efficient verification with BDDs using implicitely conjoined invariants. In *Proceedings of the 5th Conference on Computer Aided Verification*, 1993.

[Hel92] D. Heller. *Motif Programming Manual*, volume 6 of *The Definitive Guides to the X Window System*. O'Reilly & Associates, Inc., 1992.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Mass., 1979.

[Hun93] H. Hungar. Combining model checking and theorem proving to verify parallel processes. In *Computer Aided Verification, 5th International Conference*, 1993.

[ID93] C. N. Ip and D. L. Dill. Better verification through symmetry. In *International Conference on Computer Hardware Description Languages*, pages 87–100, 1993.

[Jac80] N. Jacobson. *Basic Algebra II*. Freeman, 1980.

[Jai96] J. Jain. Personal communications. 1996.

[JKR89] C. S. Jutla, E. Knapp, and J. R. Rao. A predicate transformer approach to semantics of parallel programs. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, 1989.

[Kal94] M. Kaltenbach. Model checking for UNITY. Technical Report TR94-31, The University of Texas at Austin, December 1994.

[Kal95a] M. Kaltenbach. An interactive formal system for concurrent program design. PhD dissertation proposal, The University of Texas at Austin, April 1995.

[Kal95b] M. Kaltenbach. *The UV System, User's Manual*, February 1995. Revision 1.18.

[KM89] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proceedings of the 8th ACM Symposium on Distributed Computing*, 1989.

[Kna92] E. Knapp. *Refinement as a Basis for Concurrent Program Design*. PhD thesis, The University of Texas at Austin, 1992.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, pages 333–354, 1983.

[Koz90] D. Kozen. On Kleene algebras and closed semirings. In *Proceedings, Math. Found. of Comput. Sci.*, volume 452 of *Lecture Notes in Computer Sience*, pages 26–47. Springer Verlag, 1990.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton Series in Computer Science. Princeton Uiversity Press, 1994.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions of Software Engineering*, 3(2):125–143, 1977.

[M+90] J. Misra et al. Notes on UNITY, 1990.

[M+94] Z. Manna et al. STeP: the Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Stanford University, 1994.

[MCB89] O. G. M. C. Brown, E. C. Clarke. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81(1), April 1989.

[McM92] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1992.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[Mis] J. Misra. Closure properties. unpublished manuscript.

[Mis90a] J. Misra. A family of 2-process mutual exclusion algorithms. *Notes on UNITY*, (13), 1990.

[Mis90b] J. Misra. Soundness of the substitution axiom. *Notes on UNITY*, (14), 1990.

[Mis94] J. Misra. A discipline of multiprogramming. In *Specification of Parallel Algorithms*, volume 18 of *DIMACS*. American Mathematical Society, May 1994.

[Mis95a] J. Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[Mis95b] J. Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[Mis96] J. Misra. A discipline of multiprogramming, work in progress, ftp access at `ftp://ftp.cs.utexas.edu/pub/psp/seuss/discipline.ps.Z`, 1996.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems, Safety*. Springer Verlag, New York, 1995.

[OSR93] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb 1993. Three volumes: Language, System, and Prover Reference Manuals.

[Ous94] J. K. Ousterhout. *Tcl and the Tk toolkit*. Professional Computing Series. Addison-Wesley, 1994.

[Pix90] C. Pixley. A computational theory and implementation of sequential hardware equivalence. In R. Kurshan and E. Clarke, editors, *CAV'90 DIMACS Series*. ACM, June 1990.

[Pra88] V. Pratt. Dynamic algebras as a well-behaved fragment of relation algebras. In D. Pigozzi, editor, *Conference on Algebra and Computer Science*, LNCS. Springer Verlag, June 1988.

[Pra95] W. Prasetya. *Mechanically Supported Design of Self-Stabilizing Algorithms*. PhD thesis, Rijksuniversiteit Utrecht, 1995.

[QS82] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *5th International Symposium on Programming*, volume 137 of *LNCS*. Springer Verlag, April 1982.

[Rao95] J. R. Rao. *Extensions of the UNITY Methodology*. Springer, 1995.

[RC86] J. Rees and W. Clinger. The revised^3 report on the algorithmic language Scheme. AI Memo 848a, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1986.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In *Computer Aided Verification, CAV'95*, volume 939 of *LNCS*, pages 84–97, June 1995.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *IEEE/ACM Intenational Conference of Computer Aided Design*, 1993.

[San91] B. A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3, 1991.

[Sta92] M. G. Staskauskas. Specification and verification of large-scale reactive programs. Technical report, The University of Texas at Austin, 1992.

[Sto81] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1981.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TSL+90] H. J. Touti, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using bdd's. In *IEEE Int. Conference on CAD*, November 1990.

[Wec92] W. Wechler. *Universal Algebra for Computer Scientists*, volume 25 of *WATCS Monographs on Theoretical Computer Science*. Springer, 1992.

Vita

Markus Kaltenbach was born in Freiburg i. Br., Germany, on June 3, 1964, the son of Ingrid and Albert Kaltenbach. After graduation from the Berthold-Gymnasium in Freiburg in 1983, he entered the University of Karlsruhe, Germany, where he received his Vordiplom degree in Computer Sciences with a minor in Mathematics in 1985. After one year as a Fulbright exchange student at the University of Texas at Austin in 1986/87, he returned to Karlsruhe and was awarded his Diplom degree (M.A.) in Computer Sciences with a minor in Physics in 1990. He was admitted to the Ph.D. program in Computer Sciences at the University of Texas at Austin in fall 1991. Between 1984 and 1996 he held various appointments as teaching and research assistant and worked for McKinsey&Company, Siemens, and Motorola. He will have the opportunity to start his own research group when he joins the Corporate Research Center of Siemens in Munich, Germany, in summer 1996.

Current Address: Konradstraße 10a

D-80801 Munich, Germany

Electronic Address: markus@cs.utexas.edu

This dissertation was typeset with LaTeX 2_ε by the author.