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):

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

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

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

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:

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

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:

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

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

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

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 $\mathbf{wp}$ 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

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

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 $(i+1)$ mod 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 $(i+1)$ mod n via a communication action c, and performs a b-action and synchronization with process $(i-1)$ mod n in

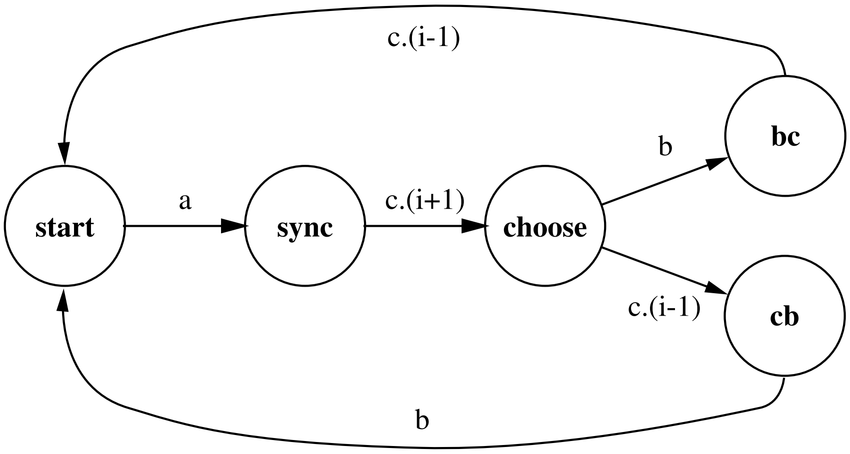

Figure 2.1: Transition Diagram for Process P_i of Milner's Cycler

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 → PC;
  initially
    last_a = N − 1;
    cyc.0 = start;
    ⟨∀i : 1 ≤ i < N : cyc.i = bc⟩;
  assign
    ⟨[]i : 0 ≤ i < N :
         last_a, cyc.i := i, sync                if cyc.i = start
      [] cyc.i := bc                             if cyc.i = choose
      [] cyc.i := start                          if cyc.i = cb
      [] cyc.(i − 1), cyc.i := choose, start     if cyc.(i − 1) = sync ∧ cyc.i = bc
      [] cyc.(i − 1), cyc.i := choose, cb        if cyc.(i − 1) = sync ∧ cyc.i = choose
    ⟩
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** :

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

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

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

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

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.