A Semantic Model For UNITY

Zhiming Liu*
University of Warwick

Abstract

This paper develops a semantic model for UNITY that reflects its particular aspects, such as nondeterminism, absence of control flow etc.. The proposed model is a kind of state-transition system within which observations and actions are the basic semantic objects. An alternative semantics for UNITY programs based on different fairness conditions is also defined using this model. Specially, the fairness condition assumed in [CM88] is defined and the UNITY logic based on it is modelled to show how the safety and liveness (progress) properties can be represented. This model is being used as the basis for developing some formal techniques for fault-tolerance within a UNITY framework.

*Address for correspondence: Department of Computer Science, University of Warwick, Coventry CV4 7AL. This work was supported by research grant GR/D 11521 from the Science and Engineering Research Council.
1 Introduction

Much of program development consists of refining specifications by adding detail. Given a problem specification, we begin by proposing a general solution strategy. Usually the strategy is broad, admitting many solutions. Next we give a specification of the solution strategy and prove that this strategy (as specified) solves the problem (as specified). When we consider a specific set of target architectures, we may choose to narrow the solution strategy, which means refining the specification further. At each stage of refinement, the programmer is obliged to prove that the specification proposed indeed refines a specification proposed at earlier step. The construction of a program is begun only after the program has been specified in extensive detail. A framework has been developed in UNITY [CM88] to meet all these purposes.

UNITY can be taken as a theory – a computational model and an associated proof system in which programs can be viewed as Unbounded Nondeterministic Iterative Transformations. It based on the thesis that the unity of the programming task transcends differences between the architectures on which programs can be executed and the application domains from which problems are drawn. The goal of the UNITY work is to show how programs can be developed systematically for a variety of architectures and applications.

Programs can be developed for different architectures, e.g. the von Neumann architecture, synchronous shared-memory multiprocessors, and asynchronous shared-memory multiprocessors, through a series of refinements by specifying program execution with as little detail as possible. Specifying little about program execution means that the programs may be nondeterministic. Different runs of the same program may execute statements in different orders, consume different amounts of resources, and even produce different results. Nondeterminism can be employed to derive simple programs, where simplicity is achieved by avoiding unnecessary determinism; such programs can be optimized by limiting the nondeterminism, i.e. by disallowing executions unsuitable for a given architecture.

A UNITY program consists of a declaration of variables, a specification of their initial values, and a set of multiple-assignments. A program execution starts from any state satisfying the initial condition and goes on forever; in each step of execution some assignment statement is selected nondeterministically and executed. Nondeterministic selection is constrained by the following fairness rule: every statement is selected infinitely often. A UNITY program describes
what should be done in the sense that it specifies the initial state and the state transformations (i.e. the assignments). A UNITY program does not specify precisely when an assignment should be executed – the only restriction is the rather weak fairness condition. Neither does a UNITY program specify where (e.g. on which processor in a multiprocessor system) an assignment is to be executed, how assignments are to be executed or how an implementation may halt a program execution. Thus, UNITY separates concerns between what on the one hand, and when, where and how on the other. The what is specified in a program, whereas the when, where, and how are specified in a mapping.

The computational model on which UNITY is based on state-transition systems [Pnu81, MP83, LT87]. A programming logic and a proof system associated with the model developed in UNITY allow the proof to be separated from the program text, and developed and studied a proof in its own right.

The specifications, refinement of specifications, proof system and mapping programs to architectures are all based on a semantic model that is assumed to exist in UNITY work. This paper develops such a semantic model for UNITY that reflects aspects of UNITY such as nondeterminism, absence of control flow, etc. The semantic model is intended to model the UNITY programming logic and proof system, and to support the UNITY specification. This model is a kind of state-transition systems within which observations and actions of assignments are the basic semantic objects. With this model, it is shown that how the different fairness conditions, termination of a program execution can be defined and how the safety and liveness (progress) properties can be represented.

2 Notes on UNITY

This section presents a subset of the UNITY notation that needed for the development of the semantic model.

- **Primitive Commands:**
  The primitive commands are conditional multiple assignments of the following form:
  \[
  x_1, \ldots, x_m := e_{11}, \ldots, e_{1m} \text{ if } b_1 \\
  \sim e_{21}, \ldots, e_{2m} \text{ if } b_2 \\
  \sim \vdots
  \]

\[ e_1, \ldots, e_m \] if \( b_n \),

where all the \( e \)'s are expressions and the \( b \)'s are boolean expressions.

Thus, a multiple assignment consists of variable list on the left side and expression lists with corresponding boolean expressions on the right side. If all component simple-expression lists (without boolean expressions) whose associated boolean expressions are \textit{true} are equal in value, any one of them can be chosen for assignment; if they are not equal in value, then the assignment results in a \textit{failure}; if no boolean expression is \textit{true}, the variable values are left unchanged.

- **Statements:**
  In [CM88], only primitive commands are UNITY statements. For defining different fairness conditions and making the semantic model mathematically elegant, the statements of the language are defined as follows:
  primitive commands are statements, and if \( S_1 \) and \( S_2 \) are statements, \((S_1[]S_2)\) is a statement. The following equivalences hold:
  \((S_1[]S_2) \equiv (S_2[]S_1)\) and \((S][]S) \equiv S\).

- **Programs:**
  An UNITY program consists of a \textit{program-name}, a \textit{declare-section} which declares the types of the program variables, \textit{initially-section} which defines the initial values of some variables in this program and an \textit{assign-section} which is a statement list, \( S_1[] \ldots []S_n \). (See [CM88] for more details).

Since the types of variables are going to be ignored in the semantic model, the \textit{declare-section} here is not considered any further here. The initially-section is a set of equations which is \textit{proper} in the sense that it can be used to determine the values of some variables named in the program. Here, the initially-section can be generally taken as a predicate about variables in the program. Therefore, a UNITY program can be abstractly denoted as

\[ P = (P_r, L), \]

where, \( P_r \) is a set of predicates over the initial values of the variables named in \( P \), and \( L \) is the assign-section of the program, \( P_r \) is in fact a set of equations of the form \( x = e \), where \( e \) is an expression.

4
In UNITY, there are two different kinds of compositions of programs. The \textit{union} of two programs is obtained by appending their codes together. Unlike \textit{union}, \textit{superposition} describes modifications of a given program by adding new variables and assignments, but not altering existing assignments. Thus, superposition preserves all the properties of the original program. For detailed definitions of the two kinds of compositions, see Section 4.

3 A Semantic Model

3.1 An Informal Overview

As seen in Section 2, an UNITY program contains a statement list. Let $S$ be a statement in such a program. In an execution of $S$, let each of its variables be represented by its \textit{observed value}, which is the sequence of values assigned to the variable. Thus, each multiple assignment to a set of variables appends an element to the previous observed value of each variable in the set. An execution of $S$ can then be described in terms of the observed values of its variables, and it is sufficient to record the observed values of its variables before and after the execution of each primitive command.

An \textit{observation} of a statement $S$ in a program is the set of observed values of the variables of the program taken at the same instant of time. If the empty sequence $\lambda$ is the initial observed value of the variables $x_1$, $x_2$, $y_1$ and $y_2$ for an execution of the statement

$$x_1, x_2 := 1, 2 \parallel y_1, y_2 := 3, 4,$$

then the sequence of the observed values of the $x$’s and $y$’s would be

\begin{verbatim}
Observations
  x1  λ  < 1 >
  x2  λ  < 2 >
  y1  λ  λ
  y2  λ  λ
\end{verbatim}

if $x_1, x_2 := 1, 2$ is selected to be executed, or
Observations
\[
\begin{array}{ccc}
x_1 & \lambda & \lambda \\
x_2 & \lambda & \lambda \\
y_1 & \lambda & < 3 > \\
y_2 & \lambda & < 4 > \\
\end{array}
\]
if \(y_1, y_2 := 3, 4\) is selected to be executed.

Thus, a statement in a program can be taken as an action, each execution of which transforms one observation into another; primitive commands can be taken as atomic actions.

Each element in an observed value is either taken from the value space of the programming variables or is a distinguished value representing a particular result of the execution of a statement. For example, execution of the statement
\[
x := 3[y := x + 1]
\]
will result in \(y\) the value of the expression \(x + 1\) only when that is defined and \(y := x + 1\) is selected to be executed. But if before executing this statement the observed value of \(x\) is \(\lambda\), the value of \(y\) is not defined if \(y := x + 1\) is selected to be executed, and the execution fails. The failure induces the \(\bot\) in all variables. Therefore, the execution of the statement will produce

Observations
\[
\begin{array}{ccc}
x & \lambda & < \bot > \\
y & < v > & < v, \bot > \\
\end{array}
\]
if \(\lambda\) and \(< v >\) are the initial observed values of \(x\) and \(y\) respectively and \(y := x + 1\) is selected to be executed.

A behaviour of an UNITY program \(P\) is an infinite sequence of observations over its variables. The first observation is determined by the initially-section of \(P\) and each of the rest is produced from its predecessor by an execution of a statement selected from the statement list of \(P\). A behaviour of a program is finite if after some observation in the sequence all the values of the variables appended in the observations keep unchanged. Properties of observations can be found in \([JG88, JG89]\) in more details.
3.2 Observations

Observations instead of states is used as the basic semantic objects. This makes it easy to define atomic actions and composed actions in a mathematical way. Let \( X \) be a set of program variables with value space \( V \) equipped with an equality relation ‘=’ which is the minimum reflexive relation on \( V \). We shall initially ignore the types of the variables, but it is relatively easy to extend the concepts in this paper to include types. Let \( V^* \) be the set of all the finite sequences (including the empty sequence \( \lambda \)) over \( V \). In addition to \( V \), we assume a set, \( \text{Stop} \), of distinguished values (\( \text{Stop} \cap V = \emptyset \)): after a variable has taken a value from \( \text{Stop} \), it can be only assigned a value in \( \text{Stop} \). Let

\[
V^\dagger = \{ s \prec d | s \in V^* \land d \in \text{Stop}^+ \},
\]

where \( \text{Stop}^+ \) is the set of all finite sequences over \( \text{Stop} \) without including the empty sequence \( \lambda \).

Then the set of observed values is defined as

\[
W = V^* \cup V^\dagger
\]

We also denote the set of all the one-element sequences over \( V \cup \text{Stop} \) as \( U \).

Some functions and predicates on \( W \) are needed. In what follows, \( S \), \( S_1 \) and \( S_2 \) are sequences in \( W \).

- \( \lambda \) the empty sequence;
- \( \#S \) the length of \( S \) (\( \# \lambda = 0 \));
- \( S[i] \) the \( i \)th element of \( S \) (\( 0 < i \leq \#S \));
- \( S_1 \preceq S_2 \) \( S_1 \) is a prefix of \( S_2 \);
- \( S_1 \prec S_2 \) \( S_1 \) is a proper prefix of \( S_2 \);
- \( S_1 \uplus S_2 \) the catenation of \( S_1 \) and \( S_2 \);
- \( S \uplus v \) appending the element \( v \) in \( V \) to the end of \( S \);
- \( \text{first}(S) \) the first element of \( S \) if \( S \) is not empty;
- \( \text{last}(S) \) the last element of \( S \) if \( S \) is not empty;
current($S$) the one-element sequence consisting of last($S$) if $S \neq \lambda$, and $\lambda$ otherwise;

rest($k, S$) the sequence obtained from $S$ by removing the first $k$ elements if $k \leq \#S$, and $\lambda$ otherwise;

rest($S, k$) the sequence obtained from $S$ by removing the last $k$ elements if $k \leq \#S$, and $\lambda$ otherwise;

rest($S_1, S_2$) rest($\#S_1, S_2$) if $S_1 \subseteq S_2$ or $S_2 \subseteq S_1$, and $S_2$ otherwise;

range($S$) the range of $S$, i.e. the set of the elements occurring in $S$;

fixed($k, S$) is true iff range(rest($k, S$)) is $\{S[k]\}$ or $\emptyset$.

With the informal description given in the previous subsection, an observation on a set of variables $X$ can be formally defined as a function as follows.

**Definition 3.1** An observation on $X$ is a function $Ob: X \rightarrow W$ satisfying

$$\exists x \in X (\text{last}(Ob(x)) = \perp) \Rightarrow \forall y \in X (\text{last}(Ob(y)) = \perp),$$

where $Ob$ can be denoted as the set of pairs $\{(x, Ob(x)) | x \in X\}$ and $\perp \in Stop$.

In the following, some of the functions and predicates over $W$ given above are extended to the set of all the observations on $X$ denoted as $OB_X$, $Ob$, $Ob'$ are observations and $k$ is an natural number.

- $Ob \preceq Ob'$ iff for each $x \in X$, $Ob_1(x) \preceq Ob'(x)$;
- $Ob_1 \prec Ob'$ iff $Ob_1 \preceq Ob'$ and there is some $x \in X$, $Ob_1(x) \prec Ob'(x)$;
- $fixed(k, Ob) \equiv \forall x \in X fixed(k, Ob(x))$;
- $current(Ob)(x) = current(Ob(x))$ for all $x \in X$;
- $rest(k, Ob)(x) = rest(k, Ob(x))$ for all $x \in X$;
- $Ob_1 \cap Ob'(x) = Ob_1(x) \cap Ob'(x)$ for all $x \in X$.

### 3.3 Actions

An action on a set of variables $X$ is a relation on the set of observations $OB_X$—that is, a set of pairs of observations. If $\xi$ is an action, then $(s, t) \in \xi$ means that executing $\xi$ starting in $s$ will get to the observation $t$. Thus a statement in a UNITY program can be defined as an action on the program variables, which specifies the transformation defined by the statement.
Definition 3.2 An action on $X$, $\xi$, is a relation on the set of all the observations on $X$ such that $\forall s \in OB_X, \exists t \in OB_X, [(s, t) \in \xi]$ and the following conditions are satisfied:

1. $(s, t) \in \xi \Rightarrow s \leq t$;
2. $(s, t) \in \xi \Leftrightarrow (current(s), current(s) \cap rest(s, t)) \in \xi$.
3. $\forall x \in X[(s, t) \in \xi \land range(current(s(x))) = \{\bot\} \Rightarrow t(x) = s(x) \cap \bot]$.

Condition 1 means that executing an action can only extend the observed values of its variables but cannot change any observed value which has already occurred. Condition 2 says that the transformation made by an action only depends on the current values of the observation with which the execution of the action started and Condition 3 implies that an action cannot have any further effect on a variable after it has taken the value $\bot$.

For an observation $s$ on $X$ and $v \in W$, $s^v$ is the function obtained from $s$ such that $(s^v(x) = v) \land \forall y \in X(y \neq x \Rightarrow s^v(x) = s(y))$. An action $\xi$ modifies a variable $x$ if there is a pair of observations $(s, t)$ in $\xi$ such that $\forall y \in X(last(t(y)) \notin Stop)$ and $s(x) \neq t(x)$; $\xi$ essentially modifies $x$ if there exists $(s, t) \in \xi$ such that $last(s(x)) \neq last(t(x)) \notin Stop$; $\xi$ does not read $x$ if for any $(s, t) \in \xi$ and any observed value $v$ of $x$ such that that $s^v$ and $t^v$ are valid observations with respect to Definition 3.1, $(s^v, t^v) \in \xi$; $\xi$ does not (essentially) access $x$ if $\xi$ does not (essentially) modify or read $x$.

Intuitively, $x$ is modified by $\xi$ if it takes value by an assignment in $\xi$, $x$ is read by $\xi$ if its value is used by an assignment in $\xi$. For examples, $x := y + 1 \text{ if } false$ does not modify $x$; $x := x$ modifies $x$ but does not essentially modify $x$.

Operations on actions:

For a set $X$ of variables and a subset $Y$ of $X$, let $s$ and $t$ be actions on $X$, $t'_s$ be the function obtained from $s$ and $t$ such that $\forall y \in Y(t'_s(x) = s(y)) \land \forall x \in (X - Y)(t'_s(x) = t(x))$. In the following, $Act$ denotes the set of all the actions on $X$.

- Union: $\cup: Act \times Act \rightarrow Act$, $\xi \cup \mu = \{(s, t)| (s, t) \in \xi \lor (s, t) \in \mu\}$.
- Composition: $\circ: Act \times Act \rightarrow Act$, $\xi \circ \mu = \{(s, t)| \exists t_1[(s, t_1) \in \xi] \land [(t_1, t) \in \mu]\}$.
• Closure: $*: \text{Act} \rightarrow \text{Act},$
  \[ \xi^* = \{(s, t) | \exists n \geq 0[(s, t) \in \xi^n]\}, \]
  where $\xi^{n+1} = \xi \circ \xi^n$ and $\xi^0 = \{(s, s)|s$
is an observation}. 

• Augmentation: $\|*: \text{Act} \times \text{Act} \rightarrow \text{Act},$
  for actions $\xi$ and $\mu$, let $X_1$ and $X_2$ be the sets of variables that are not
  modified by $\xi$ and $\mu$ respectively,
  $$(s, t) \in \xi \| \mu \text{ if } (s, t^{X_1}) \in \xi \text{ and } (s, t^{X_2}) \in \mu,$$
  or for any $x \in X$, $t(x) = s(x) \cap \bot$ if there is a $(s, t_1) \in \xi$ such that
  $\exists x \in X(last(t_1(x)) = \bot)$, or there is a $(s, t_2) \in \mu$ such that
  $\exists x \in X(last(t_2(x)) = \bot)$, or there are
  $(s, t_1) \in \xi$ and $(s, t_2) \in \mu$ such that $(t_1(y) \neq t_2(y))$ for some
  $y \in X$ which is modified by both $\xi$ and $\mu$.

Intuitively, $x := 0 if false|| x := x + 1$ is equivalent to $x := x + 1$ and
$x := x || x := x + 1$ produces $\bot$ in the semantic model.

**Definition 3.3** An action $\xi$ on $X$, is called **atomic** if it satisfies the following conditions:

1. (deterministic): $(s, t_1), (s, t_2) \in \xi \Rightarrow t_1 = t_2;$

2. (one-stepping): $(s, t) \in \xi \Rightarrow \forall x \in X[t(x) = s(x) \cap v)]$.

**Theorem 3.1** For actions $\xi$ and $\mu$, $\xi = \mu$ iff $\forall s \in OB_X(\forall x \in X(0 \leq \#s(x) \leq 1) \Rightarrow \forall t \in OB_X((s, t) \in \xi \Leftrightarrow (s, t) \in \mu)).$

Proof: It is trivial to show that if $\xi = \mu$, $\forall x \in X(0 \leq \#s(x) \leq 1) [(s, t) \in \xi \Leftrightarrow
(s, t) \in \mu]$. 

To prove that only if $\xi = \mu$, $\forall s \in OB_X(\forall x \in X(0 \leq \#s(x) \leq 1) \Rightarrow \forall t \in OB_X((s, t) \in \xi \Leftrightarrow (s, t) \in \mu))$; we assume that $X = \{x\}$ without loss of
generality. Let $(s, t) \in \xi$, if $\#s \leq 1$, we have already got $(s, t) \in \mu$. If $\#s > 1$, from
Condition 2 in Definition 3.2, we have $(\text{current}(s), \text{current}(s) \cap \text{rest}(s, t)) \in \xi,$
and thus $(\text{current}(s), \text{current}(s) \cap \text{rest}(s, t)) \in \mu$ because $\#\text{current}(s(x)) \leq 1$.
Again from Condition 2 in Definition 3.2, $(s, t) \in \mu.$

$\square$
Theorem 3.2  The set of all atomic actions is closed under the augmentation, that is, if $\xi$ and $\mu$ are atomic actions, then so is $\xi \parallel \mu$.

Proof: Let $\xi$ and $\mu$ be atomic actions, $X_1$ and $X_2$ be the sets of variables that are not modified by $\xi$ and $\mu$ respectively. It is obvious that $\xi \parallel \mu$ is one-stepping. Therefore, it is sufficient to show that $\xi \parallel \mu$ is deterministic.

Let $(s, t_1), (s, t_2) \in \xi \parallel \mu$, if $\forall x(t_1(x)) = s(x) \land \bot$, then $\forall x(last(s(x))) = \bot \lor \exists t[(s, t) \in \xi \land \forall x(last(t(x))) = \bot] \lor \exists t[(s, t) \in \mu \land \forall x(last(t(x))) = \bot]$, or there are $(s, t) \in \xi$, $(s, t') \in \mu$ and $y$ which is modified by both $\xi$ and $\mu$, such that $(t(y) \neq t'(y))$. Since $\xi$ and $\mu$ are atomic, it is easy to check that for each case listed above, $\forall x(t_2(x)) = s(x) \land \bot$, that is $t_1 = t_2$. Similarly, if $\forall x(t_1(x)) = s(x) \land \bot$, then $t_1 = t_2$. If for any $x$, $last(t_1(x)) \neq \bot$, then for any $x$, $last(t_2(x)) \neq \bot$ and $(s, t'_1(x)) \in \xi \land (s, t'_2(x)) \in \xi \land (s, t'_1(x)) \in \mu \land (s, t'_2(x)) \in \mu$. Since $\xi$ and $\mu$ are atomic, $(s, t'_1(x)) = (s, t'_2(x))$ and $(s, t'_1(x)) = (s, t'_2(x))$. Hence, $t_1 = t_2$.

\[\square\]

In the UNITY framework, augmentations are applied only to atomic actions.

Definition 3.4 The class of generated actions is defined recursively as follows:

1. Atomic actions are generated actions;
2. If $\xi$ and $\mu$ are generated actions, then $\xi \cup \mu$ is a generated action;
3. If $\xi$ and $\mu$ are generated actions, then $\xi \circ \mu$ is a generated action;
4. If $\xi$ and $\mu$ are generated actions, then $\xi \parallel \mu$ is a generated action;
5. Generated actions can be obtained only by applying these rules finitely many times.

Definition 3.5 An action, $\xi$, is a program action if it can be decomposed into the union of a set of atomic actions.

Theorem 3.3 The set of all program actions on $X$ is closed under the union, i.e. if $\xi$ and $\mu$ are program actions, $\xi \cup \mu$ is also a program action.

Proof: Trivial.
3.4 Executions

Definition 3.6 Let $X$ be a set of variables, $\Pi$ be a set of actions on $X$ and $R_0$ be an observation on $X$; then an infinite sequence of observations,

$$E = R_0R_1 \ldots R_i \ldots,$$

is an execution of $\Pi$ with $R_0$, if for each $i \geq 0$, there exists an action $\xi \in \Pi$, such that $(R_i, R_{i+1}) \in \xi$, and this is denoted as $E = (R_i)$.

We use $Ex(R_0, \Pi)$ to denote the set of all executions of $\Pi$ with $R_0$, and $EX(\Pi)$ to denote the set of all the executions of $\Pi$. Later, we will define an UNITY program as a set of program actions. The properties of the program are expressed as assertions over the set of executions. Also, we can treat an execution, $E = (R_i)$, as a function from the set of natural numbers $\mathbb{N}$ to the set of observations $O(B_X)$ such that for all $i \geq 0$, $R_i = E(i)$ and $(E(i), E(i+1)) \in \xi$ for some $\xi \in \Pi$. We use both sequences and functions to represent executions in this paper.

Definition 3.7 Let $\Pi$ be a set of actions on $X$ and

$$E = R_0R_1 \ldots R_i \ldots$$

is an execution of $\Pi$ with $R_0$, $E$ is called a finite execution of $\Pi$, if there exist $n \geq 0$ and $m \geq 0$ such that for any $i \geq n$, $\text{fixed}(m, R_i) = \text{true}$, and if $n$ is the least number for which such an $m$ can be found, the finite execution $E$ is represented as

$$E = R_0 \ldots R_n.$$

Definition 3.8 An execution of $\Pi$ is said to be fair, if for any $i \geq 0$ and $\xi \in \Pi$, there is some $k \geq i$ such that $(R_k, R_{k+1}) \in \xi$.

We use $Fxex(R_0, \Pi)$ to denote the set of all fair executions of $\Pi$ with $R_0$, and $FXE(\Pi)$ to denote the set of all the fair executions of $\Pi$.

Note that if a statement is defined as an action while a primitive command is defined as an atomic action, the definition of fairness seems to be weaker than that given in [CM88] in the following sense: a UNITY program in [CM88] can be defined as a set of atomic actions $\Pi$, and to say that an execution $E$ of $\Pi$ is fair just means that $E$ could be obtained by executing each atomic action in
Π infinitely many times, but possibly also if certain atomic actions in Π were executed only finitely many times. On the other hand, the fairness condition given here is abstract, and is sufficient for reasoning about programs and justifying implementations of programs (see Section 4). There can be many different sets Π that give the same same set of executions. For example, replacing two actions ξ and µ by the single action ξ ∪ µ does not change the set of executions but may change the set of fair executions. Thus, there can be many equivalent ways to write essentially the same program.

In [MP83] this particular form of fairness is called justice [GP89]. Some other alternative forms of fairness [Fra86] can be also be defined. Specially, let Π = Π1 ∪ Π2. An execution E of Π is said to be fair with respect to Π1, if for each i ≥ 0 and each action ξ ∈ Π1, there exists k ≥ i such that (E(k), E(k + 1)) ∈ ξ. E is said to bounded with respect to Π1, if there exists i ≥ 0 such that for any k ≥ i,(E(k), E(k + 1)) ∈ ξ for some ξ ∈ Π2. Such forms of fairness can be used to define the sumption that faults cannot occur infinitely often when dealing with fault-tolerance [Liu89]. The logic for specification and proof of correctness of programs is of course based on specific fairness. The UNITY logic is based on justice.

When we deal with the semantics of programming languages, we are usually interested only in the executions of sets of generated actions. For UNITY, we concentrate on the executions of sets of program actions.

4 The Semantics of the UNITY Notations

- The semantics of expressions:
  Let X be the set of the variables used in a program, Expr be the set of all the expressions over X and V, then the semantic function for the expressions is defined as:

    \[ E : \text{Expr} \rightarrow (\text{Func}(V \cup \text{Stop}) \rightarrow V \cup \text{Stop}) , \]

where Func(S) is the set of all the functions with finite arguments over S, and for each expression \( e(x_1, \ldots, x_n) \) with \( x_1, \ldots, x_n \) occurring within it, \( E(e(x_1, \ldots, x_n)) \) is determined by the operations used in \( e \) and satisfies the following strictness condition:
\( e(a_1, \ldots, a_n) = \bot \), if some \( a_i = \bot \).

where there is no confusion, we also use the expression \( e \) to represent the value of \( \mathcal{E}(e) \).

- The semantics of the UNITY Statements:
  Let \( \text{Statement} \) denote the set of all the UNITY statements and \( \text{Progact} \) denote the set of all the program actions, the semantic function for statements is defined recursively as

\[
\mathcal{S} : \text{Statement} \rightarrow \text{Progact}:
\]

1. Basis: For a primitive command \( p \),

\[
x_1, \ldots, x_m := e_{11}, \ldots, e_{1m} \text{ if } b_1
\]
\[
\sim e_{21}, \ldots, e_{2m} \text{ if } b_2
\]
\[
\vdots
\]
\[
\sim e_{n1}, \ldots, e_{nm} \text{ if } b_n,
\]

\( \mathcal{S}(p) \) consists of the pairs \((s, t)\) satisfying:

(a) \( \forall x \in X(\text{last}(x) = \bot \Rightarrow t(x) = s(x) \uparrow \bot) \);

(b) \( \forall i \leq n(b_i(\text{current}(s)) = \text{false}) \Rightarrow t = s \);

(c) if there is a \( k \) such that \( b_k(\text{current}(s)) = \text{true} \), and \( s(y) = \lambda \) for some \( y \) named in \( e_{ki} \) for some \( i \leq m \), then \( \forall x \in X(t(x) = s(x) \uparrow \bot) \);

(d) if there are \( i, j \leq m \) such that \( i \neq j \land x_i = x_j \) and there is a \( k \leq n \) such that \( b_k(\text{current}(s)) = \text{true} \), but \( e_{ki} \neq e_{kj} \), then \( \forall x \in X(t(x) = s(x) \uparrow \bot) \);

(e) if there are \( k, l \leq n \) such that \( b_k(\text{current}(s)) = b_l(\text{current}(s)) = \text{true} \), but there is an \( i \leq m \) such that \( e_{ki} \neq e_{li} \), then \( \forall x \in X(t(x) = s(x) \uparrow \bot) \); otherwise

(f) if there is a \( k \leq n \) such that \( b_k(\text{current}(s)) = \text{true} \), then \( \forall i \leq m \, t(x_i) = s(x_i) \uparrow e_{ki} \) and \( t(x) = s(x) \) for any \( x \) that is different from each \( x_i \).

2. Recursion: \( \mathcal{S}(S_1 \mid S_2) = \mathcal{S}(S_1) \cup \mathcal{S}(S_2) \).

14
The Semantics of Statement Lists:
A statement list is of the form as:
\[ L = S_1 \ldots S_n. \]

The semantics of \( L \) is defined as follows:
\[ \mathcal{L}(L) = \{ S(S_1), \ldots, S(S_n) \}. \]

So the semantics of a statement list is the set of the program actions determined by the statements of the list. It is reasonable to semantically treat statements and statement lists at different levels because of the fairness conditions. Note that for statement lists, \( L_1 \) and \( L_2 \),
\[ \mathcal{L}(L_1 || L_2) = \mathcal{L}(L_1) \cup \mathcal{L}(L_2). \]

It is not very essential for us to consider the empty statement list \( NULL \), i.e. a list without any statement. If we have to consider it, just let \( \mathcal{L}(NULL) = \emptyset \).

Now we can define the set of behaviours of a statement list, \( L \), as the set of executions of \( \mathcal{L}(L) \), and denote it as \( \mathcal{BEH}(L) \), thus,
\[ \mathcal{B EH}(L) = EX(\mathcal{L}(L)). \]

In the same way, we can define the set of fair behaviours of a statement list \( L \), and denote it as \( \mathcal{FBEH}(L) \).

The Semantics of UNITY Programs:
We define the semantics of an UNITY program as a set of behaviours of the statement list contained within it, in which each behaviour satisfies the initial conditions defined in the initially-section of the program and some fairness conditions.

For a UNITY program \( P = (Pr, L) \), we say a behaviour of \( L, F \in \mathcal{B EH}(L) \) satisfies the initial condition \( Pr \), if for all \( x \in X(\#F(0) \leq 1) \) and \( Pr[F(0)] = true \), and \( FSatPr \) denotes that \( F \) satisfies \( Pr \). Here, the meaning of \( Pr[F(0)] = true \) is just the same as that in first order logic.
Now we can define alternative semantics for UNITY programs based on different fairness conditions.

**Definition 4.1** Let $P = (Pr, L)$ be an UNITY program,

1. The **loose semantics** of $P$ is defined as the set of the behaviours of $L$ that satisfies the initial condition $Pr$ of $P$, i.e.
   \[ \mathcal{B} \mathcal{E} \mathcal{H}(P) = \{ F \mid F \in \mathcal{B} \mathcal{E} \mathcal{H}(L) \land F \text{Sat} Pr \} \]

2. The **fair semantics** of $P$ is defined as the set of the fair behaviours of $L$ that satisfies the initial condition $Pr$ of $P$, i.e.
   \[ \mathcal{F} \mathcal{B} \mathcal{E} \mathcal{H}(P) = \{ F \mid F \in \mathcal{F} \mathcal{B} \mathcal{E} \mathcal{H}(L) \land F \text{Sat} Pr \} \]

Note that if $Pr = false$ in $P = (Pr, L)$, then
\[ \mathcal{B} \mathcal{E} \mathcal{H}(P) = \mathcal{F} \mathcal{B} \mathcal{E} \mathcal{H}(P) = \emptyset. \]

The fair semantics of $P$ can be implemented by executing each statement in $L$ infinitely many times. We can also define some other alternative semantics for UNITY programs. Specially, let $P = (Pr, \langle L_1 \rangle \langle L_2 \rangle)$. We say that a behaviour, $F$, of $L = \langle L_1 \rangle \langle L_2 \rangle$ is **fair with respect to** $L_1$, if $F$ is fair with respect to $\mathcal{L}(L_1)$. $F$ is **bounded** with respect to $L_1$, if $F$ is bounded with respect to $\mathcal{L}(L_1)$. $F$ is bounded with respect to $L_1$ means that $F$ can be obtained by executing $L_1$ only finite times.

Now let us look into composition and superposition of programs. If $P_1 = (Pr_1, L_1)$ and $P_2 = (Pr_2, L_2)$ are programs, the **union** of $P_1$ and $P_2$ is written as $P_1 \parallel P_2$. We use $\mathcal{B}$ to denote any one of the semantic functions based on different fairness conditions, such as $\mathcal{B} \mathcal{E} \mathcal{H}$ and $\mathcal{F} \mathcal{B} \mathcal{E} \mathcal{H}$ etc, then we define
\[ \mathcal{B}(P_1 \parallel P_2) = \mathcal{B}(Pr_1 \land Pr_2, \langle L_1 \rangle \langle L_2 \rangle). \]

Note this definition is consistent with the assumption that there are no inconsistencies in the definitions of variables.

For a program $P = (Pr, L)$, and a primitive command $s$ in $P$, if $r$ is a primitive command, let $P_s^r = (Pr, L_s^r)$ represent the program $Q$ that is obtained from $P$ by replacing $s$ in $P$ by $r$. We can define an **augmentation** of a program $P = (Pr, L)$ with $Pr'$ and a primitive command $r$ as follows:
\[ \mathcal{A}(P, s, Pr', r) = (Pr \land Pr', L_{s[r]}^r), \]
where, $s$ is a primitive command in $P$, $r$ is a primitive command which does not modify any variable named in $P$ and $Pr'$ is a predicate about the initial values of the variables named in $r$. The program $P$ in the above equation is called the underlying program.

Now we have the semantics of an augmentation of a program as

$$B(A(P, s, Pr', r)) = B((Pr \land Pr', L^s_{\|Pr})), $$

i.e. replacing the atomic action determined by $s$ by the atomic action determined by $s \| r$.

After modelling the UNITY programming logic, it is easy to prove the Union Theorem in [CM88], but the Superposition Theorem in [CM88] is required to be restated in our framework as: Every property of the underlying program is a property of the augmentation if no failure is introduced by the superposed variables.

5 Modelling the UNITY Programming Logic

This section shows that within the semantic model given in this paper, the UNITY programming logic can be modelled and thus safety and progress properties of a program can be described. In this section, all predicates used in the logic are about observations on specified sets of variables. For each variable $x$ with value space $V_x$, a sequence variable $\pi$ is introduced whose value space is $W_x = V_x \cup V^x$. A predicate $p$ over $OBX$ names only variables in $X$ or some $\pi$ for $x \in X$. Let $p$ be a predicate over $OBX$ and $Ob$ be an observation in $OBX$, $Ob \models p[Ob]$ is true, where $p[Ob]$ is obtained by replacing each occurrence of $x \in X$ by last$(Ob(x))$ and each occurrence of $\pi$ by Ob$(x)$.

Most properties of a UNITY program are expressed using assertions of the form $\{p\}S\{q\}$, where $S$ is universally or existentially quantified over the statements (here statements are assignment statements) of the program. $\{p\}S\{q\}$ denotes that an execution of statement $S$ in any state (observation) that satisfies predicate $p$ results in a state (observation) that satisfies predicate $q$, if execution of $S$ terminates. Properties that hold for all program observations during execution are defined using only universal quantification. Properties that hold eventually are defined using existential quantification as well.
Traditionally, a program property is regarded as either a safety or a progress property. Existential quantification over program statements is essential in stating progress properties, whereas safety properties can be stated using only universal quantifications over statements (using the initial condition). For describing certain kinds of properties that should be considered with any program, the terms, unless, stable, invariant, ensures, leads-to and fixed point, are introduced and theorems about them are derived in UNITY. Therefore, the essential part of modelling this logic is modelling assertions of the form \( \{p\}S\{q\} \).

Since statements are defined as actions, \( \{p\}S\{q\} \) is defined as the triple \( \{p\}S(S)\{q\} \). A general definition of the Hoare logic of actions is needed [Lam87].

Let \( / \) be an action on \( X \), \( p \) and \( q \) be predicates over \( OB \); the Hoare triple \( \{p\} \xi \{q\} \) is defined to mean \( \forall (s,t) \in \xi [(s \models p) \Rightarrow (t \models q)] \). In other words, \( \{p\} \xi \{q\} \) asserts that if \( p \) is true in observation \( s \) and executing \( \xi \) in observation \( s \) can produce observation \( t \), then \( q \) is true in \( t \). It is easy to check that the usual rules for reasoning about Hoare triples [Hoa69] hold for triples of actions. Some of these are listed as follows:

- \( \{p\}\xi \{true\} \);
- \( \{false\}\xi \{q\} \);
- \( \{p\}\xi \{false\} \Rightarrow \neg p; \)
- \( \{p\}\xi \{q\}, \{p’\}\xi \{q’\} \Rightarrow \{p \lor p’\}\xi \{q \lor q’\}, \{p \land p’\}\xi \{q \land q’\} \);
- if \( p’ \Rightarrow p, \{p\}\xi \{q\} \) and \( q \Rightarrow q’ \), then \( \{p’\}\xi \{q’\} \);
- if \( \{p\}\xi \{q\} \) and \( \{p’\} \mu \{q’\} \), then \( \{p \lor p’\}\xi \cup \mu \{q \lor q’\} \) and \( \{p \land p’\}\xi \parallel \mu \{q \land q’\} \); note the special case when \( p = p’ \) and \( q = q’ \).
- if \( \{q\} \xi \cup \mu \{q\} \), then \( \{q\}\xi \{q\} \) and \( \{q\} \mu \{q\} \).

For a given set of actions (program actions) \( \Pi \), \( p \) unless \( q \) in \( \Pi \) is defined as follows:

\[
p \text{ unless } q \equiv < \forall \xi \in \Pi(\{p \land \neg q\} \xi \{p \lor q\}) > .
\]

Thus if \( p \) is true at some point in an execution of \( \Pi \) and \( q \) is not, in the next step (i.e. after execution of an action) \( p \) remains true or \( q \) becomes true. Hence, if \( p \) holds at any point during an execution of \( \Pi \), then either
• $q$ never holds and $p$ continues to hold forever, or

• $q$ holds eventually (it may holds initially when $p$ holds) and $p$ continues to hold at least until $q$ holds.

This can be formalized and easily proved as the following theorem.

**Theorem 5.1** $p \textit{ unless } q$ in $\Pi$ iff for each execution $E$ of $\Pi$ and any $i \geq 0$, $\pi q i \Rightarrow (p \lor q)](E(i + 1))$.

**Corollary 1** If $p \textit{ unless } q$ in $\Pi$, then for any any execution $E$ of $\Pi$ and $i \geq 0$, $p]i \Rightarrow < \forall j : j \geq i \Rightarrow (p \land q)[E(j)] > (i.e. p \land q holds forever) \lor $ $< \exists k : k \geq i \Rightarrow q[E(k)] > (i.e. q holds eventually) \land $ $< \forall j : i \leq j < k \Rightarrow (p \land q)](E(j)] > (i.e. until then p \land q holds)]$.

$\square$

The special cases of $\textit{unless}$ are stable and invariant. For a given set of actions $\Pi$,

• $p$ is stable $\equiv p \textit{ unless } false$

• $q$ is invariant $\equiv (\text{initial condition } \Rightarrow q) \land q$ is stable

We say that an action $\xi$ leaves $p$ stable if and only if $\{p\}\xi\{p\}$ holds. For example, every action leaves $\forall x \in X(\text{last}(ob(x)) = \bot)$ stable. If $\xi$ leaves both $p$ and $q$ stable, then it leaves $p \land q$ and $p \lor q$ stable as well; if both $\xi$ and $\mu$ leave $q$ stable, then $\xi \cup \mu$ leaves $p$ stable; if $\xi$ does not modify the variables in $p$, then $\xi$ leaves both $p \lor \forall x(\text{last}(ob(x)) = \bot)$ and $p \lor \forall x(\text{last}(ob(x)) = \bot)$ stable; if $\xi$ leaves $p$ stable and $\mu$ leaves $q$ stable, then $\xi \cup \mu$ leaves $p \lor q$ stable and if no variable can be modified by both $\xi$ and $\mu$, $\xi \parallel \mu$ leaves $p \land q$ stable. $p$ is stable in $\Pi$ iff each action in $\Pi$ leaves $p$ stable.

The $\textit{ensures}$ relation (over predicates) is used to define the most basic progress properties of programs. For a given set of actions (program actions), $\Pi$, $p \textit{ ensures } q$ is defined as follows:

$$p \textit{ ensures } q \equiv (p \textit{ unless } q) \land < \exists \xi : \xi \in \Pi :: \{p \land \xi\} >$$.
Thus if $p$ is true at some point in an execution of $\Pi$, $p$ remains true as long as $q$ is false, and eventually $q$ becomes true if the execution is fair. Hence we have

**Theorem 5.2**  \ p \text{ ensures } q \text{ holds in } \Pi \text{ iff for any fair execution } E \text{ of } \Pi \text{ and } i \geq 0,

\[
p[E(i)] \Rightarrow \exists j : j \geq i :: q[E(j)] \land \forall k : i \leq k < j :: p[E(k)] >>
\]

\[\square\]

The leads-to ($\rightarrow$) relation over predicates is defined by ensures and inference rules [CM88]. In terms of action-executions, $p \rightarrow q$ holds in $\Pi$ if for any fair execution $E$ of $\Pi$ and $i \geq 0$,

\[
p[E(i)] \Rightarrow \exists j : j \geq i :: q[E(j)] >.
\]

Finally, the fixed point $FP$ of $\Pi$ is defined as: for any execution $E$ of $\Pi$ and $i \geq 0$,

\[
FP[E(i)] \equiv \forall j : j \geq i :: current(E(i)) = current(E(j)) >.
\]

Obviously, $FP$ is reachable (i.e. such an $i$ exists in the above equivalence) for $E$ iff $E$ is a finite execution.

It is straightforward (but tedious) to prove the validity in our framework of the rules of the UNITY proof system and the theorems about ensures, leads-to, stable, invariant and fixed point. We leave these to the interested reader.

### 6 Conclusions and Discussions

The basic semantic objects of our approach are observed values of program variables and actions - relations over sets of observed values. The purpose of introducing actions into the model is to formalize alternative fairness conditions so as to obtain alternative semantics for programs. The semantic model has been constructed for justifying the UNITY specification and proof system. Obviously, specifications and proof systems heavily depend upon the assumed fairness conditions.

In fact, the semantic model established in this paper is quite language independent. It is easy to be extended to model conventional programming language constructs and architectures [JG88, JG89]. Therefore, this semantic model could
be suitable for justifying proofs of the correctness of mapping UNITY programs to alternative architectures.

The proposed semantic model could also support the techniques for developing fault-tolerant systems that are presented in UNITY. And some concepts, such as “an action ξ does not (effectively) modify a variable x”, might be useful for describing properties of faulty actions, e.g. to show that a faulty action does not (essentially) modify a safe variable, e.g. a variable used for checkpointing. For programs \( P_1 = (P_{r1}, L_1) \) and \( P_2 = (P_{r2}, L_2) \), we can define for \( P_1 || P_2 \) the semantics that is fair with respect to \( L_1 \) and bounded with respect to \( L_2 \). When taking \( P_2 \) as the program which simulates the specified hardware faults, some properties of its behaviour could be described. Furthermore, this semantics fits the supposition that faults can only occur finitely often. This model is being used as the basis for developing some techniques for fault-tolerance within the UNITY framework [Liu89].

Acknowledgment

I wish to thank my supervisor, Prof. Mathai Joseph, for his constant encouragement, guidance, and his comments on earlier versions of this report as well. My thanks also go to my colleagues, A. Goswami and A.M. Lord, for the helpful discussion during this work.

References


