On the Relation Between Unity Properties and Sequences of States

R.T. Udink and J.N. Kok

RUU-CS-93-07
February 1993

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

On the Relation Between Unity Properties and Sequences of States

R.T. Udink and J.N. Kok

Technical Report RUU-CS-93-07
February 1993

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0024-3275

On the Relation Between Unity Properties and Sequences of States

R.T. Udink* and J.N. Kok

Utrecht University,
Department of Computer Science,
P.O. Box 80.089,
3508 TB Utrecht, the Netherlands.
Email: {rob,joost}@cs.ruu.nl.

Abstract

Stepwise refinement of programs has proven to be a suitable method for developing parallel and distributed programs. We examine and compare a number of different notions of program refinement for Unity. Two of these notions are based on execution sequences. Refinement corresponds to the reduction of the set of execution sequences, i.e., reducing the amount of nondeterminism. The other refinement notions are based on Unity properties as introduced by Chandy and Misra. The Unity approach is to refine specifications. Although it has proven a suitable formalism for deriving algorithms, it seems less suitable for handling implementation details. Following Sanders and Singh, we formalize program refinement in the Unity framework as the preservation of Unity properties. We show that Unity properties are not powerful enough to characterize execution sequences. As a consequence, the notion of property-preserving refinement differs from the notion of reducing the set of execution sequences.

1 Introduction

Developing correct parallel and distributed programs from specification to implementation is a difficult task. Stepwise refinement has proven to be a useful methodology for this task.

The Unity framework, as introduced by Chandy and Misra in [CM88], consists of a programming language and a programming logic. The logic is based on a set of temporal properties. These properties are used to give specifications. The Unity approach is to refine specifications toward a specific architecture until a program can be derived easily. A specification is refined by a stronger set of properties. As can be seen from the case studies in [CM88], this method is useful for deriving parallel and distributed algorithms. However, it is not easy to deal with low-level implementation details at the level of specification. So, the specification refinement seems less suitable for the final stage of program development. In this stage of the development process, program refinement is more useful. This consists of transforming programs toward a specific architecture in such a way that semantic properties of the program are preserved. There are different notions of what kind of properties are to be preserved. Because we are also interested in interactive programs, we need a semantic notion that takes into account some temporal behavior of the program, not only its pre- and postconditions. Sanders [San90] defines a syntactic notion of program refinement similar to reactive refinement as defined by Back in [Bac90]. For this kind of refinements, she is interested in the preservation of adjusted Unity properties. In [Sin91], Singh uses a similar approach for the original Unity properties. Lamport and Abadi [AL88] base their work on behaviors, sequences of states that can occur during program execution. Refinement of a program should reduce the set of behaviors, that is, it reduces the amount of nondeterminism.

*This research has been supported by the Foundation for Computer Science in the Netherlands SION under project 612-317-107

In this paper the relation between these notions of program refinement is examined for Unity programs. Therefore, we define a number of semantic models for Unity programs. First, we define two models based on sequences of states. The first semantics of a program is the set of stutter-free sequences of states that can occur during program execution. An extension for compositionality results in the second model. Secondly, we define some models based on Unity properties. The semantics of a program is the set of properties (safety and progress properties) that it satisfies. We can base the semantics models on two notions of progress: either *leadsto* properties, or *ensures* properties. Since we can use properties defined by Chandy and Misra as well as those defined by Sanders, we define four different models. Each model yields a notion of refinement. At first sight one might think that the notion of refinement in terms of sequences and properties are equivalent. We will show that Unity properties are not powerful enough to characterize sequences. Consequently, the notion of property-preserving refinement differs from the notion of reducing the set of execution sequences.

This paper is organized as follows. In section 2 we give a brief introduction of the Unity programming language. In section 3, the notion of program executions is formalized in terms of sequence of states and section 4 gives an introduction to the Unity logic. Section 5 discusses some notions of program refinement, based on executions sequences and Unity properties. These notions will be compared in section 6. Section 7 mentions some conclusions and further research.

2 Unity programs

In this section, a brief overview of the Unity programming language, as introduced by Chandy and Misra in [CM88], is given. We will denote Unity programs by F and G. A Unity program has several parts that are called sections. We will only consider a subset of Unity programs, namely, programs that are made up of the following sections.

- An **initially**-section defining the initial values of variables. We denote by $init.F$ the set of possible states that satisfy the requirements of the **initially**-section of a program F.
- An **assign**-section containing a non-empty set of (possibly multiple and/or conditional) assignment statements (for a program F denoted by $assign.F$). Assignment statements are separated by the symbol $[\!]$. Assignment statements are deterministic and the execution of each statement always terminates.

When it is clear from the context, we may use F to denote $init.F$ or $assign.F$.

An example of a Unity program is given below.

Program F
  initially
    $x \geq 0$
  assign
    $x := x + 1$ **if** $x \geq 0$
  $[\!]$ $x := x - 1$ **if** $x < 0$
end$\{F\}$

The idea of program execution is as follows. Execution of a Unity program F starts in a state contained in $init.F$. In each step an assignment statement is chosen from the set $assign.F$ and executed. Furthermore, an execution has to be fair, that is, each statement should be chosen infinitely often. If the guard of the statement evaluates to *false*, execution of that statement is a skip statement. The execution of a Unity program never terminates. However, there is the notion of fixed point: if after some moment the state cannot be changed by any statement of F, one can view this state as the result of the computation.

Execution of the program given above starts in a state where $x = c$ for some $c \geq 0$ and repeatedly a statement is executed. Execution of the first statement increases x by one. This

happens infinitely often. The second statement, which is also selected infinitely often, does not change the value of x. So, the execution results in an ever increasing value of x.

Two programs can be composed with the union operator $\|$:

Definition 2.1 *Let F and G be Unity programs. The union of F and G, denoted by $F\|G$, is defined as*

$$init.(F\|G) \mathrel{\hat{=}} init.F \cap init.G,$$

$$assign.(F\|G) \mathrel{\hat{=}} assign.F \cup assign.G.$$

For example, the program F, as given above, can be composed with program G. This results in program $F\|G$:

Program G	**Program** $F\|G$
initially	**initially**
$x > 0$	$x > 0$
assign	**assign**
$x := -x$	$x := x + 1$ **if** $x \geq 0$
end$\{G\}$	$\|\; x := x - 1$ **if** $x < 0$
	$\|\; x := -x$
	end$\{F\|G\}$

Program union can be interpreted as a kind of parallel composition, where parallelism is modeled as interleaving of actions.

3 Operational semantics

In this section, we define two operational semantic models for Unity programs. One consists of sequences of states, the other is an extension to make the model compositional. Chandy and Misra have defined an execution model for Unity programs in terms of sequences of tuples. Each tuple consists of a state and a label of the statement that is executed. We use the operational view that only states can be observed. This means that neither stutterings, nor the statement that is executed can be observed. This corresponds to the idea of the Unity properties that also abstract from stutterings. In [Liu89], Liu gives the semantics of Unity programs in terms of fair execution sequences. This model resembles our first operational semantics.

Our first model gives a set of stutter-free sequences of states that may occur during an execution of the program. We will first define some preliminaries. Let Σ be the set of states, which could be modeled as functions from variables to values, and $Seq = \mathcal{P}(\Sigma^*)$ be the domain of sets of infinite state sequences. We use the $\langle\!\langle . \rangle\!\rangle$-brackets to denote sequences. We denote function application by a dot that associates to the left: $f.g.x = (f.g).x$. For sequences an operator $\natural$ is defined by Abadi and Lamport in [AL88] that removes stutterings from a sequence, i.e., it replaces all maximal finite segments $\rho\rho\cdots\rho$ of identical states by the single state ρ. E.g., $\natural.\langle\!\langle \rho_0\rho_1\rho_1\rho_2\rho_2\rho_2\cdots\rangle\!\rangle = \langle\!\langle \rho_0\rho_1\rho_2\cdots\rangle\!\rangle$, if ρ_0,ρ_1, and ρ_2 are different states. We call a sequence z stutter-free if $\natural.z = z$. This means that z may only contain a suffix of stutterings.

Lemma 3.1 *For every sequence $z = \langle\!\langle \sigma_0, \sigma_1, \sigma_2, \cdots\rangle\!\rangle$,*

$$\natural.z = z \equiv \langle \forall i :: \sigma_i = \sigma_{i+1} \Rightarrow \langle \forall j : i \leq j :: \sigma_j = \sigma_{j+1}\rangle\rangle.$$

By $\sigma\ s\ \sigma'$ we denote that execution of statement s in state σ results in state σ'.

Now we define the first operational semantics of a Unity program.

Definition 3.2 *The operational semantics $\mathcal{O}_1 : Unity \rightarrow Seq$ is defined for a Unity program F as the set of stutter-free sequences of states z for which there exists an infinite sequence*

$$z' = \langle\!\langle \sigma_0, \sigma_1, \sigma_2, \ldots\rangle\!\rangle$$

such that

- $\natural.z' = z$,
- $init.F.\sigma_0$,
- $\langle \forall i :: \langle \exists s : s \in assign.F :: \sigma_i \; s \; \sigma_{i+1} \rangle \rangle$, *and*
- $\langle \forall s : s \in assign.F :: \langle \exists_i^\infty :: \sigma_i \; s \; \sigma_{i+1} \rangle \rangle$.

The quantification : $\exists_i^\infty$ says that there are infinitely many i. The third condition states that a sequence only contains transitions that can be made by a statement of the program. The fourth condition formalizes fairness. Remark that the definition of fairness is slightly different from the definition used by Chandy and Misra. They require that every statement is executed infinitely often. We only require that the effect of each statement is visible as a state transition (that may be caused by executing another statement with the same effect) infinitely often. However, since a Unity program only contains a finite number of statements, both notions result in the same set of sequences of states. Note that $\mathcal{O}_1[\![F]\!]$ can contain only infinite sequences. Termination of a program (reaching a fixed point) corresponds to an infinite suffix of stutterings. The following lemma gives a characterization of $\mathcal{O}_1$ that is more suitable for calculation.

Lemma 3.3 *For every sequence* $z = \langle\!\langle \sigma_0, \sigma_1, \cdots \rangle\!\rangle$ *and Unity program* F*:*

$$z \in \mathcal{O}_1[\![F]\!] \equiv init.F.\sigma_0 \wedge \natural.z = z \wedge ok_1.F.z \wedge fair_1.F.z$$

where

$$ok_1.F.z = \langle \forall i :: \langle \exists s : s \in assign.F :: \sigma_i \; s \; \sigma_{i+1} \rangle \rangle,$$

$$fair_1.F.z = \langle \forall i, s : s \in F :: \langle \exists j : i \leq j :: (\sigma_j \; s \; \sigma_j) \vee (\sigma_j \; s \; \sigma_{j+1}) \rangle \rangle$$

The model $\mathcal{O}_1$ has the drawback that it is not compositional with respect to union of programs. This can be seen from the following example.

Program F	**Program** G
initially	**initially**
$x \geq 0$	$x \geq 0$
assign	**assign**
$x := x + 1$ **if** $x \geq 0$	$x := x + 1$ **if** $x \geq 0$
$[\!]$ $x := x - 1$ **if** $x < 0$	**end**$\{G\}$
end$\{F\}$	

Both programs F and G have the same stutter-free sequences of states, i.e., $\mathcal{O}_1[\![F]\!] = \mathcal{O}_1[\![G]\!]$. However, composition with the program consisting of the statement $x := -x$ will result in different sequences because x may become negative. If $x < 0$, x can be decreased by the second statement of F, no statement of G has a similar effect.

Hence, for some compositional semantics it is not sufficient to have all execution sequences: we need a semantic model that allows for interleaving. Like [BKPR91], we use an extended notion of sequences to make the model compositional. Extended sequences have holes and the intuition is that the holes can be filled by the environment (that is, another Unity program). So, an extended sequence gives the contribution of a program when it operates in a composition. Extended sequences are sequences of pairs of states. The first state of each pair is arbitrary, reached by the program or its environment, the second is the result of the execution of any statement of the program. Because the first state of the first pair of an extended sequence can be any state, we have to take care of the initial states explicitly. So, we define the domain of extended sequences by $ESeq = (\mathcal{P}(\Sigma), \mathcal{P}((\Sigma \times \Sigma)^*))$.

We also want to abstract from stuttering in the compositional model. However, it is not possible to remove all stuttering from each extended sequence. Then, it would not be possible to derive the set of connected sequences. Like [BKPR91], we only remove connected stutterings, and to make the model more abstract, it is allowed to add stutterings. To remove stutterings, we

define the operator $\natural$ for extended sequences. This operator removes all maximal finite segments of connected stutterings, i.e., it replaces all finite segments $(\sigma, \rho), (\rho, \rho), \cdots, (\rho, \rho)$ by (σ, ρ) and $(\rho, \rho), \cdots, (\rho, \rho), \cdots, (\rho, \sigma)$ by (ρ, σ). Then the compositional model is defined as follows.

Definition 3.4 *The operational semantics $\mathcal{O}_2$: Unity $\rightarrow$ ESeq is defined for a Unity program F as the pair (I, V) where $I = init.F$ and V is the set of all sequences of pairs of states v for which there exists an extended sequence*

$$v' = \langle\!\langle(\sigma_0, \sigma'_0), \cdots, (\sigma_n, \sigma'_n), \cdots\rangle\!\rangle$$

such that

- $\natural v' = v$,
- $\langle \forall i :: \langle \exists s : s \in F.assign :: \sigma_i \; s \; \sigma'_i \rangle \vee (\sigma_i = \sigma'_i) \rangle$, *and*
- $\langle \forall s : s \in F :: \langle \exists_i^\infty :: \sigma_i \; s \; \sigma'_i \rangle \rangle$.

Similar as for $\mathcal{O}_1$, we make an alternative characterization of $\mathcal{O}_2$ that is better for calculations.

Lemma 3.5 *For every extended sequence $v = \langle\!\langle(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \cdots\rangle\!\rangle$ and Unity program F:*

$$v \in V_F \equiv \natural.v = v \wedge ok_2.F.v \wedge fair_2.F.v$$

where

$$ok_2.F.v = \langle \forall i :: \langle \exists s : s \in assign.F :: \sigma_i \; s \; \sigma'_i \rangle \vee \sigma_i = \sigma'_i \rangle,$$

$$fair_2.F.v = \langle \forall i, s : s \in F :: \langle \exists j : i \leq j :: (\sigma_j \; s \; \sigma_j) \vee (\sigma_j \; s \; \sigma'_j) \vee (\sigma'_j \; s \; \sigma'_j) \rangle \rangle$$

The predicates ok_2 and $fair_2$ have a nice relation with the removal of stutterings. It follows directly from the fact that $\natural$ only removes finite segments.

Lemma 3.6 *For each extended sequence v and Unity program F*

$$ok_2.F.v \equiv ok_2.F.(\natural.v) \quad \wedge \quad fair_2.F.v \equiv fair_2.F.(\natural.v)$$

We define an abstraction function β that relates the two operational models. The idea is to take all extended sequences that are connected, i.e., each pair ends in a statement in which the next pair continues. These sequences can be concatenated.

Definition 3.7 *Let β : ESeq $\rightarrow$ Seq be defined by*

$$\beta.(I, V) \mathrel{\widehat{=}} \{cat.z \mid startin.(z, I) \wedge connected.z \wedge z \in V\},$$

where

$$startin.(\langle\!\langle(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \cdots\rangle\!\rangle, I) \mathrel{\widehat{=}} (\sigma_0 \in I),$$

$$connected.\langle\!\langle(\sigma_0, \sigma'_0), \cdots, (\sigma_n, \sigma'_n), \cdots\rangle\!\rangle) \mathrel{\widehat{=}} \langle \forall i :: \sigma_i = \sigma_{i+1} \rangle,$$

and

$$cat.\langle\!\langle(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \cdots\rangle\!\rangle \mathrel{\widehat{=}} \langle\!\langle\sigma_0, \sigma_1, \sigma_2, \cdots\rangle\!\rangle.$$

Before proving our theorem, we will prove some lemmas about the relation between stutter free-sequences and extended stutter-free sequences.

Lemma 3.8 *For every extended sequence v*

$$connected.v \Rightarrow (\natural.v = v \equiv \natural.(cat.v) = (cat.v))$$

Lemma 3.9 *For every Unity program F and every extended sequence v such that connected.v and* $\natural.v = v$

$$(fair_2.F.v \equiv fair_1.F.(cat.v)),$$

$$(fair_2.F.v) \Rightarrow (ok_2.F.v \equiv ok_1.F.(cat.v)).$$

Proof: The proof for fairness is straight forward,

$$
\begin{array}{ll}
 & ok_2.F.\langle\!\langle(\sigma_0, \sigma_0'), (\sigma_1, \sigma_1'), \cdots\rangle\!\rangle \\
\equiv & \quad \{\text{definition } ok_2\} \\
 & \langle \forall i :: \langle \exists s : s \in F :: \sigma_i \; s \; \sigma_i' \rangle \vee \sigma_i = \sigma_i' \rangle \\
\equiv & \quad \{connected\} \\
 & \langle \forall i :: \langle \exists s : s \in F :: \sigma_i \; s \; \sigma_{i+1} \rangle \vee \sigma_i = \sigma_{i+1} \rangle \\
\equiv & \quad \natural.(cat.v) = (cat.v) \text{ and lemma 3.1} \\
 & \langle \forall i :: \langle \exists s : s \in F :: \sigma_i \; s \; \sigma_{i+1} \rangle \vee \langle \forall j : i \leq j :: \sigma_j = \sigma_{j+1} \rangle \rangle \\
\equiv & \quad \{fair_1.F.(cat.v)\} \\
 & \langle \forall i :: \langle \exists s : s \in F :: \sigma_i \; s \; \sigma_{i+1} \rangle \rangle \\
\equiv & \quad \{\text{definition } ok_1\} \\
 & ok_1.F.(cat.\langle\!\langle(\sigma_0, \sigma_0'), (\sigma_1, \sigma_1'), \cdots\rangle\!\rangle)
\end{array}
$$

□

Theorem 3.10 $\mathcal{O}_1 = \beta \circ \mathcal{O}_2.$

Proof:

$$
\begin{array}{ll}
 & \beta.\mathcal{O}_2[\![F]\!] \\
\equiv & \quad \{ \text{ lemma 3.5 } \} \\
 & \beta.(init.F, \natural.v = v \wedge ok_2.F.v \wedge fair_2.F.v) \\
\equiv & \quad \{ \text{ definition } \beta\} \\
 & \{cat.v | init.F.v_0 \wedge connected.v \wedge \natural.v = v \wedge ok_2.F.v \wedge fair_2.F.v\} \\
\equiv & \quad \{ \text{ lemma 3.8 en lemma 3.9}\} \\
 & \{cat.v | init.F.v_0 \wedge connected.v \wedge \natural.(cat.v) = (cat.v) \wedge ok_1.F.(cat.v) \wedge fair_1.F.(cat.v)\} \\
\equiv & \quad \{cat.v \Rightarrow \langle \exists v' :: cat.v' = cat.v \wedge connected.v' \rangle\} \\
 & \{cat.v | init.F.v_0 \wedge \natural.(cat.v) = (cat.v) \wedge ok_1.F.(cat.v) \wedge fair_1.F.(cat.v)\} \\
\equiv & \quad \{z = cat.v\} \\
 & \{z | init.F.z_0 \wedge \natural.z = z \wedge ok_1.F.z \wedge fair_1.F.z\} \\
\equiv & \quad \{ \text{ lemma 3.3 } \} \\
 & \mathcal{O}_1[\![F]\!]
\end{array}
$$

□

To show that this model is compositional, we define the function $\tilde{[\!]}$, which is the semantic equivalent of program union.

Definition 3.11 *Let F, G be Unity programs and* $\mathcal{O}_2[\![F]\!] = (I_F, V_F)$, *and* $\mathcal{O}_2[\![G]\!] = (I_G, V_G)$. *Then*

$$\mathcal{O}_2[\![F]\!] \; \tilde{[\!]} \; \mathcal{O}_2[\![G]\!] \triangleq (I_F \cap I_G, \{\natural.v | \langle \exists v_f, v_g :: v_f \in V_F \wedge \; v_g \in V_G :: v \in merge(v_f, v_g) \rangle\}.$$

The operation merge is the standard fair interleaving on sequences.

Since $\{\natural.v | merge(v_1, v_2)\} = \{\natural.v | merge(\natural.v_1, \natural.v_2)\}$ for all extended sequences v_1, v_2 it is straight forward to verify the following theorem.

Theorem 3.12 *For Unity programs F, G,*

$$(\mathcal{O}_2[\![F]\!] \; \tilde{[\!]} \; \mathcal{O}_2[\![G]\!]) = \mathcal{O}_2[\![F [\!] G]\!].$$

4 Unity logics

In this section, we give an overview of two logics for the Unity programming language: a variation of the logic in [CM88] and a logic given by Sanders in [San91]. Both logics are based on a small set of temporal properties that are defined in terms of the statements of a program. To provide an intuition for the properties, we will give an interpretation of the properties in terms of the operational models given in the previous section.

Before introducing the logics, we need to introduce some predicate transformers. The semantics of a single assignment statement can be given by its weakest liberal precondition *wlp* or strongest postcondition *sp* predicate transformers (see for example [DS90]).

Definition 4.1 *Let $(x := E)$ be an assignment statement. The weakest liberal precondition and the strongest postcondition are defined as*

$$wlp.(x := E).p \mathrel{\hat{=}} p(E/x),$$

$$sp.(x := E).p \mathrel{\hat{=}} \langle \exists y : x = E(y/x) :: p(y/x) \rangle.$$

The predicate $p(E/x)$ is the predicate p in with E substituted for x.

The definitions for multiple assignments are similar, using simultaneous substitution.

Because the execution of each assignment statement always terminates, the notions of weakest liberal precondition and weakest precondition are the same. In the sequel, we model state-predicates as functions from states to booleans, however, we sometimes interpret them as a set of states. Since Unity statements are deterministic assignment statements, $\sigma\ s\ \sigma'$ corresponds with $p_\sigma \Rightarrow wlp.s.p_{\sigma'}$ and $sp.s.p_\sigma = p_{\sigma'}$, where p_σ is the predicate that is *true*in σ and *false*otherwise, that is $p_\sigma.\tau = (\sigma = \tau)$. We sometimes use σ to denote the predicate p_σ. In the next definition we lift the predicate transformers *wlp* and *sp* to Unity programs.

Definition 4.2 *Let F be a Unity program and p a predicate. The predicate transformers wlp and sp are lifted to Unity programs by*

$$wlp.F.p \mathrel{\hat{=}} \langle \forall s : s \in assign.F :: wlp.s.p \rangle,$$

$$sp.F.p \mathrel{\hat{=}} \langle \exists s : s \in assign.F :: sp.s.p \rangle.$$

Next, we introduce the predicate transformer *sst*, the strongest stable predicates, as given in [San91].

Definition 4.3 *The predicate sst.F.p is the strongest solution for q of*

$$[q \Rightarrow wlp.F.q \wedge p \Rightarrow q].$$

Sanders has proven the following characterization of *sst* that can be used to compute this predicate transformer.

Theorem 4.4 *Define function f by $f.y = (sp.F.y) \vee p$. Then*

$$sst.F.p = \langle \exists i : i \geq 0 :: f^i.false \rangle,$$

where $f^0 = identity$ and $f^i = f \circ f^{i-1}$ if $i > 0$.

We will use the *sst*-predicate transformer to characterize the set of all reachable states of a program F:

Lemma 4.5 *For a Unity program F*

$$sst.F.(init.F).\sigma \equiv \langle \exists z, i : z = \langle\!\langle \rho_0, \rho_1, \cdots \rangle\!\rangle \in \mathcal{O}_1[\![F]\!] :: \rho_i = \sigma \rangle.$$

We abbreviate the set of reachable states $sst.F.(init.F)$ by $sinv.F$, the strongest invariant of F.

Now we define the logics for Unity programs. Although the logic of Chandy and Misra is older and the logic of Sanders is a modification of it, we introduce the latter first, since the interpretation of the properties of this logic is closer to the operational intuition. The logic of Sanders is introduced in [San91] and is based on four temporal properties: ***invariant***, ***unless***, ***ensures***, and ***leadsto***. The properties are attached to an entire program and they are defined in terms of the set of reachable states and the effect of the statements of the program Because we introduce the Chandy and Misra logic later we subscript the properties by S. Following Dijkstra and Scholten [DS90], we use square brackets to denote universal quantification over all states.

Definition 4.6 (Sanders Logic) *Let p, q be arbitrary predicates and F a Unity program. Define the following properties of F by*

1. *invariant property:*

$$invariant_S\ q \mathrel{\hat{=}} [sinv.F \Rightarrow p]$$

2. *unless property:*

$$p\ unless_S\ q \mathrel{\hat{=}} [sinv.F \Rightarrow (p \wedge \neg q \Rightarrow wlp.F.(p \vee q))].$$

3. *ensures property:*

$$p\ ensures_S\ q \mathrel{\hat{=}} p\ unless_S\ q \wedge \langle \exists s : s \in F :: [sinv.F \Rightarrow ((p \wedge \neg q) \Rightarrow wlp.F.q)]\rangle.$$

4. *leadsto property: $\mapsto_S$ is defined as the smallest binary relation R between predicates satisfying the following conditions:*
   (a) $R \supseteq ensures_S$,
   (b) *R is transitive,*
   (c) *for any set W, if $\langle \forall m : m \in W :: p_m\ R\ q\rangle$ then $\langle \exists m : m \in W :: p_m\rangle\ R\ q$.*

If the program is not clear from the context we will mention the program explicitly using the connective **in** , e.g., $p\ unless_S\ q$ **in** F.

The Sanders logic differs from the logic of Chandy and Misra in the fact that all properties are restricted to reachable states (expressed by the $sinv.F$ in each definition). This has two important consequences. Firstly, the following substitution principle holds:

*if **invariant** $(a = b)$ then we may substitute a for b in every property of the program.*

Secondly, the interpretation of the properties in terms of normal sequences corresponds to the intuition of the properties.

The $unless_S$ property is a safety property. The operational interpretation of $p\ unless_S\ q$ is that if p becomes *true* during the execution of the program it remains *true* as long as q is *false*.

Lemma 4.7 *For a Unity program F,*

$$p\ unless_S\ q\ \textbf{in}\ F \equiv \langle \forall z, i : z = \langle\!\langle \sigma_0, \sigma_1, \cdots \rangle\!\rangle \in \mathcal{O}_1[\![F]\!] :: (p \wedge \neg q).\sigma_i \Rightarrow (p \vee q).\sigma_{i+1}\rangle.$$

Proof:

$$\begin{array}{ll}
 & p\ unless_S\ q\ \textbf{in}\ F \\
\equiv & \quad \{\text{definition } unless_S\ \} \\
 & [sinv.F \Rightarrow ((p \wedge \neg q) \Rightarrow wlp.F.(p \vee q))] \\
\equiv & \quad \{\text{quantification over states}\ \} \\
 & \langle \forall \sigma : \sigma \in \Sigma :: sinv.F.\sigma :: ((p \wedge \neg q) \Rightarrow wlp.F.(p \vee q)).\sigma\rangle \\
\equiv & \quad \{\text{lemma 4.5}\ \}
\end{array}$$

$$
\begin{array}{ll}
& \langle \forall z, i : z = \langle\!\langle \sigma_0, \sigma_1, \cdots \rangle\!\rangle \in \mathcal{O}_1[\![F]\!] :: ((p \wedge \neg q).\sigma_i \Rightarrow wlp.F.(p \vee q).\sigma_i) \rangle \\
\equiv & \quad \{\text{definition } wlp.F\ \} \\
& \langle \forall z, i : z = \langle\!\langle \sigma_0, \sigma_1, \cdots \rangle\!\rangle \in \mathcal{O}_1[\![F]\!] :: ((p \wedge \neg q).\sigma_i \Rightarrow \langle \forall s : s \in F :: wlp.s.(p \vee q).\sigma_i \rangle) \rangle \\
\equiv & \quad \{\text{ predicate calculus }\} \\
& \langle \forall s : s \in F :: \langle \forall z, i : z \in \mathcal{O}_1[\![F]\!] :: ((p \wedge \neg q).\sigma_i \Rightarrow wlp.s.(p \vee q).\sigma_i) \rangle \rangle \\
\equiv & \quad \{\ \langle \forall s, z, i : s \in F \wedge z \in \mathcal{O}_1[\![F]\!] :: \langle \exists z', j : z' \in \mathcal{O}_1[\![F]\!] :: \sigma_i = \sigma'_j \wedge \sigma'_j = wlp.s.\sigma'_{j+1} \rangle \rangle\ \} \\
& \langle \forall s : s \in F :: \langle \forall z, i : z \in \mathcal{O}_1[\![F]\!] \wedge \sigma_i = wlp.s.\sigma_{i+1} :: ((p \wedge \neg q).\sigma_i \Rightarrow wlp.s.(p \vee q).\sigma_i) \rangle \rangle \\
\equiv & \quad \{\text{ interpretation } wlp\ \} \\
& \langle \forall s : s \in F :: \langle \forall z, i : z \in \mathcal{O}_1[\![F]\!] \wedge \sigma_i = wlp.s.\sigma_{i+1} :: (p \wedge \neg q).\sigma_i \Rightarrow (p \vee q).\sigma_{i+1} \rangle \rangle \\
\equiv & \quad \{\text{predicate calculus }\} \\
& \langle \forall z, i : z \in \mathcal{O}_1[\![F]\!] :: (p \wedge \neg q).z_i \Rightarrow (p \vee q).z_{i+1} \rangle
\end{array}
$$

□

The $ensures_S$ and $leadsto_S$ properties are progress properties. The operational interpretation of p $ensures_S$ q is that if p holds it remains to hold until q holds and q will hold within finite time, i.e., a finite number of execution steps. The definition of the $leadsto_S$ property is the definition of Pachl given in [Pac90]. It differs slightly from the definition of Sanders where the inference rules may only be applied a finite number of times. We use Pachl's notion because of its correspondence to the operational intuition of the *leadsto* property, i.e., $p \mapsto_S q$ holds in F iff whenever p is *true*, q will become *true* within finite time. This is expressed in the following lemma that was proven by Pachl in [Pac92]:

Lemma 4.8 *For a Unity program F*

$$p \mapsto_S q \ \textbf{in} \ F \equiv \langle \forall z, i : z = \langle\!\langle \sigma_0, \sigma_1, \cdots \rangle\!\rangle \in \mathcal{O}_1[\![F]\!] :: p.\sigma_i \Rightarrow \langle \exists j : j \geq i :: q.\sigma_j \rangle \rangle.$$

Now we define the original Unity logic as defined by Chancy and Misra in [CM88].

Definition 4.9 (Chandy-Misra Logic) *Let p, q be arbitrary predicates and F a Unity program. Define the following properties of F by*

1. *unless property:*

$$p \ unless_{CM} \ q \ \hat{=} \ [(p \wedge \neg q) \Rightarrow wlp.F.(p \vee q)].$$

2. *ensures property:*

$$p \ ensures_{CM} \ q \ \hat{=} \ p \ unless_{CM} \ q \ \wedge \ \langle \exists s : s \in assign.F :: [(p \wedge \neg q) \Rightarrow wlp.s.q] \rangle.$$

3. *leadsto property: $\mapsto_{CM}$ is defined as the smallest binary relation R between predicates satisfying the following conditions:*
   - *(a) $R \supseteq ensures_{CM}$,*
   - *(b) R is transitive,*
   - *(c) for any set W, if $\langle \forall m : m \in W :: p_m \ R \ q \rangle$ then $\langle \exists m : m \in W :: p_m \rangle \ R \ q$.*

In this framework the *invariant* property (as some more properties) is defined as a "derived" property, in terms of the three basic properties:

$$stable_{CM} \ p = p \ unless_{CM} \ false,$$

$$invariant_{CM} \ p = (init.F \Rightarrow p) \wedge (stable_{CM} \ p),$$

$$p \ until_{CM} \ q = (p \ unless_{CM} \ q) \wedge (p \mapsto_{CM} q).$$

The properties defined here are stronger than the properties defined by Sanders, i.e., $pR_{CM}\,q \Rightarrow pR_S\,q$, but the reverse implication does not hold. In fact, we can express the Sanders's properties in the Chandy and Misra logic as follows.

Lemma 4.10 *For a Unity property $R \in$ unless , ensures , $\mapsto$ and a Unity program F,*

$$p\ R_S\ q \equiv (sinv.F \wedge p)\ R_{CM}\ (sinv.F \wedge q)$$

Proof: For R is the *unless* property:

$$
\begin{array}{ll}
& p\ unless_S\ q \ \textbf{ in } F \\
\equiv & \quad \{ \text{ definition } unless_S \} \\
& [sinv.F \Rightarrow (p \wedge \neg q \Rightarrow wlp.F.(p \vee q))] \\
\equiv & \quad \{ \text{ predicate calculus } \} \\
& [(sinv.F \wedge p \wedge \neg q) \Rightarrow wlp.F.(p \vee q)] \\
\equiv & \quad \{ sinv.F \Rightarrow wlp.F.(sinv.F) \} \\
& [(sinv.F \wedge p \wedge \neg q) \Rightarrow (wlp.F.(sinv.F) \wedge wlp.F.(p \vee q))] \\
\equiv & \quad \{ \text{ } wlp \text{ is conjunctive } \} \\
& [(sinv.F \wedge p \wedge \neg q) \Rightarrow wlp.F.(sinv.F \wedge (p \vee q))] \\
\equiv & \quad \{ \text{ predicate calculus } \} \\
& [((sinv.F \wedge p) \wedge \neg(sinv.F \wedge q)) \Rightarrow wlp.F.((sinv.F \wedge p) \vee (sinv.F \wedge q))] \\
\equiv & \quad \{ \text{ definition } unless_{CM} \} \\
& (sinv.F \wedge p)\ unless_{CM}\ (sinv.F \wedge q) \ \textbf{ in } F
\end{array}
$$

The proof of the *ensures* is similar:

$$
\begin{array}{ll}
& p\ ensures_S\ q \ \textbf{ in } F \\
\equiv & \quad \{ \text{ definition } ensures_S \} \\
& p\ unless_S\ q \ \textbf{ in } F \wedge \langle \exists s : s \in F :: [(sinv.F \wedge p \wedge \neg q) \Rightarrow wlp.s.q] \rangle \\
\equiv & \quad \{ sinv.F \Rightarrow wlp.s.(sinv.F) \} \\
& p\ unless_S\ q \ \textbf{ in } F \wedge \langle \exists s : s \in F :: [(sinv.F \wedge p \wedge \neg q) \Rightarrow (wlp.s.(sinv.F) \wedge wlp.s.q)] \rangle \\
\equiv & \quad \{ \text{ previous, } wlp \text{ is conjunctive} \} \\
& (sinv.F \wedge p)\ unless_{CM}\ (sinv.F \wedge q) \ \textbf{ in } F \wedge \\
& \langle \exists s : s \in F :: [((sinv.F \wedge p) \wedge \neg(sinv.F \wedge q)) \Rightarrow wlp.s.(sinv.F \wedge q)] \rangle \\
\equiv & \quad \{ \text{ definition } ensures_{CM} \} \\
& (sinv.F \wedge p)\ ensures_{CM}\ (sinv.F \wedge q) \ \textbf{ in } F
\end{array}
$$

For the $\mapsto$ property the relation directly follows from the *ensures*. □

As a consequence, the interpretations given in lemma 4.7 and 4.8 only hold in one direction, when we examine the Chandy and Misra properties. Neither does the substitution principle hold as can be seen from the following example. For example, for program F, introduced in section 2, we can derive the properties $(x = c \wedge x \geq 0)\ unless_{CM}\ x > c$ and $invariant_{CM}\ (x \geq 0) \equiv true$. The substitution principle says that $(x = c)\ unless_{CM}\ (x \geq c)$ should be a property of F. However, this is not true if c is negative. This is because Unity properties also say something about the behavior of the program in states that are never reached during any program execution, e.g., the states where $x < 0$ for program F. This characteristic, however, make these properties suitable for compositional reasoning.

$$
\begin{array}{rcl}
p\ unless_{CM}\ q \ \textbf{ in } F [] G & = & (p\ unless_{CM}\ q \ \textbf{ in } F \wedge p\ unless_{CM}\ q \ \textbf{ in } G), \\
p\ ensures_{CM}\ q \ \textbf{ in } F [] G & = & (p\ ensures_{CM}\ q \ \textbf{ in } F \wedge p\ unless_{CM}\ q \ \textbf{ in } G) \vee \\
& & (p\ ensures_{CM}\ q \ \textbf{ in } G \wedge p\ unless_{CM}\ q \ \textbf{ in } F).
\end{array}
$$

Chandy and Misra give an intuitive interpretation in terms of extended sequences.

Lemma 4.11 *For a Unity program F and $\mathcal{O}_2[\![F]\!] = (I, V)$.*

$$p\ unless_{CM}\ q \ \textbf{ in } F \equiv \langle \forall (\sigma, \sigma') : \langle\!\langle \cdots, (\sigma, \sigma'), \cdots \rangle\!\rangle \in V :: (p \wedge \neg q).\sigma \Rightarrow (p \vee q).\sigma' \rangle.$$

Proof:

$$
\begin{array}{ll}
 & p\ \mathit{unless}_{CM}\ q\ \textbf{in}\ F \\
\equiv & \quad \{\text{ definition } \mathit{unless}_{CM}\ \} \\
 & [(p \wedge \neg q) \Rightarrow wlp.F.(p \vee q)] \\
\equiv & \quad \{\text{ quantification over state space }\} \\
 & \langle \forall \sigma :: (p \wedge \neg q).\sigma \Rightarrow (\langle \forall s : s \in F :: wlp.s.(p \vee q) \rangle).\sigma \rangle \\
\equiv & \quad \{\ \forall \sigma :: \langle\!\langle \cdots, (\sigma, \sigma'), \cdots \rangle\!\rangle \in \mathcal{O}_2[\![F]\!]\ \} \\
 & \langle \forall \sigma : \langle\!\langle \cdots, (\sigma, \sigma'), \cdots \rangle\!\rangle \in \mathcal{O}_2[\![F]\!] :: (p \wedge \neg q).\sigma \Rightarrow (\langle \forall s : s \in F :: wlp.s.(p \vee q) \rangle).\sigma \rangle \\
\equiv & \quad \{\text{ predicate calculus }\} \\
 & \langle \forall s, \sigma : s \in F \wedge \langle\!\langle \cdots, (\sigma, \sigma'), \cdots \rangle\!\rangle \in \mathcal{O}_2[\![F]\!] :: (p \wedge \neg q).\sigma \Rightarrow (wlp.s.(p \vee q)).\sigma \rangle \\
\equiv & \quad \{\text{from definition of } \mathcal{O}_2\text{: } \sigma\ s\ \sigma' \vee \sigma = \sigma'\ \} \\
 & \langle \forall (\sigma, \sigma') : \langle\!\langle \cdots, (\sigma, \sigma'), \cdots \rangle\!\rangle \in \mathcal{O}_2[\![F]\!] :: (p \wedge \neg q).\sigma \Rightarrow (p \vee q).\sigma' \rangle
\end{array}
$$

□

Lemma 4.12 *For a Unity program F and $\mathcal{O}_2[\![F]\!] = (I, V)$.*

$$p\ \mathit{ensures}_{CM}\ q\ \textbf{in}\ F \Rightarrow (\langle \forall i : (p \wedge \neg q).\sigma_i \rangle \Rightarrow \langle \exists i : q.\sigma_i' \rangle)$$

for every sequence $v = \langle\!\langle (\sigma_1, \sigma_1'), (\sigma_2, \sigma_2'), \cdots \rangle\!\rangle \in V$.

Proof: Assume that $p\ \mathit{ensures}_{CM}\ q\ \textbf{in}\ F$, then:

$$
\begin{array}{ll}
 & \langle \forall i :: (p \wedge \neg q).\sigma_i \rangle \\
\Rightarrow & \quad \{\ p\ \mathit{unless}\ q \text{ and lemma 4.11}\} \\
 & \langle \forall i :: (p \wedge \neg q).\sigma_i \wedge (p \vee q).\sigma_i' \rangle \\
\Rightarrow & \quad \{\text{predicate calculus}\} \\
 & \langle \forall i :: (p \wedge \neg q).\sigma_i \wedge (p \wedge \neg q).\sigma_i' \rangle \vee \langle \exists i :: q \sigma_i' \rangle \\
\Rightarrow & \quad \{\text{fairness}\} \\
 & (\langle \forall s : s \in F :: \langle \exists i :: \sigma_i\ s\ \sigma_i \vee \sigma_i\ s\ \sigma_i' \vee \sigma_i'\ s\ \sigma_i' \rangle \rangle \wedge \langle \forall i :: (p \wedge \neg q).\sigma_i \wedge (p \wedge \neg q).\sigma_i' \rangle) \\
 & \vee \langle \exists i :: q.\sigma_i' \rangle \\
\Rightarrow & \quad \{\text{predicate calculus}\} \\
 & \langle \forall s : s \in F :: \langle \exists i :: (\sigma_i\ s\ \sigma_i \vee \sigma_i\ s\ \sigma_i' \vee \sigma_i'\ s\ \sigma_i') \wedge (p \wedge \neg q).\sigma_i \wedge (p \wedge \neg q).\sigma_i' \rangle \rangle \\
 & \vee \langle \exists i :: q.\sigma_i' \rangle \\
\Rightarrow & \quad \{\exists s \in F : [(p \wedge \neg q) \Rightarrow wlp.s.q]\ \} \\
 & \langle \exists i :: (q.\sigma_i \vee q.\sigma_i') \wedge (p \wedge \neg q).\sigma_i \wedge (p \wedge \neg q).\sigma_i' \rangle \vee \langle \exists i :: q.\sigma_i' \rangle \\
\Rightarrow & \quad \{\text{predicate calculus}\} \\
 & \langle \exists i :: q.\sigma_i' \rangle
\end{array}
$$

□

5 Refinement relations

In the previous sections we have defined notions of execution sequences and we have given in introduction in Unity logic. In this section, we will concentrate on refinement. We will define some notions of refinement based on the models given in the previous sections.

First, we look at the model $\mathcal{O}_1$, the sets of sequences. For this model we use a notion of refinement that corresponds to the idea of implementation as defined by Abadi and Lamport in [AL88]; a specification S_1 is implemented by a specification S_2 if every behavior of S_2 is allowed by S_1. Going from specification to implementation, the set of execution sequences reduces. The number of choices that can be made decreases, in other words, the amount of nondeterminism decreases.

Definition 5.1 *Let F, G be Unity programs*

$$F \sqsubseteq G \textbf{ in } \mathcal{O}_1 \mathrel{\hat{=}} (\mathcal{O}_1[\![F]\!] \supseteq \mathcal{O}_1[\![G]\!]).$$

We use the connective **in** to indicate the model. In a similar way we give a refinement relation based on the model $\mathcal{O}_2$, the sets of extended sequences.

Definition 5.2 *For Unity programs F, G; $\mathcal{O}_2[\![F]\!] = (I_F, V_F)$, and $\mathcal{O}_2[\![G]\!] = (I_G, V_G)$,*

$$F \sqsubseteq G \textbf{ in } \mathcal{O}_2 \mathrel{\hat{=}} (I_F \supseteq I_G \wedge V_F \supseteq V_G).$$

In [CM88], Chandy and Misra defined a notion of refinement for specifications, that are sets of properties: a specification $S1$ is refined by a specification $S2$ iff all properties of $S1$ can be derived from the properties of $S2$. Sanders ([San90]) and Singh ([Sin91]) used this idea to define refinement for Unity programs as preservation of properties: a program F is refined by program G if every property of F is a property of G.

To define this notion formally, we first define some semantic models for Unity programs based on their properties. We define in total four different models; two based on the Unity logic of Chandy and Misra, and the others based on the logic of Sanders. For each logic one model is based on *ensures* properties and the other on *leadsto* properties. Therefore, we need the following domains: $U = \mathcal{P}(\Sigma) \times \mathcal{P}(P) \times \mathcal{P}(P)$ as a triple containing a set of initial states, and two sets of properties. The domain P is the property domain, i.e., a pair of sets of states, $P = \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$ (pairs of predicates). We often switch between predicates and sets of states.

Definition 5.3 *Define for $* \in \{S, CM\}$ and a Unity program F*

- $\mathcal{IUE}_*[\![F]\!] \mathrel{\hat{=}} (I, U, E)$,
  where $I = F.init$, $U = \{(p, q) |\ p \textit{ unless}_* \ q \textbf{ in } F\}$, and
  $E = \{(p, q) |\ p \textit{ ensures}_* \ q \textbf{ in } F\}$.
- $\mathcal{IUL}_*[\![F]\!] \mathrel{\hat{=}} (I, U, L)$,
  where $I = F.init$, $U = \{(p, q) |\ p \textit{ unless}_* \ q \textbf{ in } F\}$, and $L = \{(p, q) |\ p \mapsto_* q \textbf{ in } F\}$.

Remark that $\mathcal{IUE}_{CM}$ is the strongest model of the four and that it is compositional. This follows directly from the theory given in section 4. For these property-based models we can define refinement as preservation of properties as follows.

Definition 5.4 *For $* \in \{S, CM\}$ and Unity programs F, G.*
Let $\mathcal{IUE}_*[\![F]\!] = (I_F, U_F, E_F)$ and $\mathcal{IUE}_*[\![G]\!] = (I_G, U_G, E_G)$. Then,

$$F \sqsubseteq G \textbf{ in } \mathcal{IUE}_* \mathrel{\hat{=}} (I_F \supseteq I_G \wedge U_F \subseteq U_G \wedge E_F \subseteq E_G).$$

Let $\mathcal{IUL}_*[\![F]\!] = (I_F, U_F, L_F)$ and $\mathcal{IUL}_*[\![G]\!] = (I_G, U_G, L_G)$. Then,

$$F \sqsubseteq G \textbf{ in } \mathcal{IUL}_* \mathrel{\hat{=}} (I_F \supseteq I_G \wedge U_F \subseteq U_G \wedge L_F \subseteq L_G).$$

6 Relation between the models

In the previous sections we gave a number of semantic models for Unity programs and for each model we defined a refinement relation. In this section we examine the relations between the models. We want to know how the notions of refinement for the models are related. Therefore, we define the following relation on models.

Definition 6.1 *For two models $\mathcal{M}_1$ and $\mathcal{M}_2$,*

$$\mathcal{M}_1 \rightarrow \mathcal{M}_2 \mathrel{\hat{=}} \langle \forall F, G :: (F \sqsubseteq G \textbf{ in } \mathcal{M}_1) \Rightarrow (F \sqsubseteq G \textbf{ in } \mathcal{M}_2) \rangle,$$

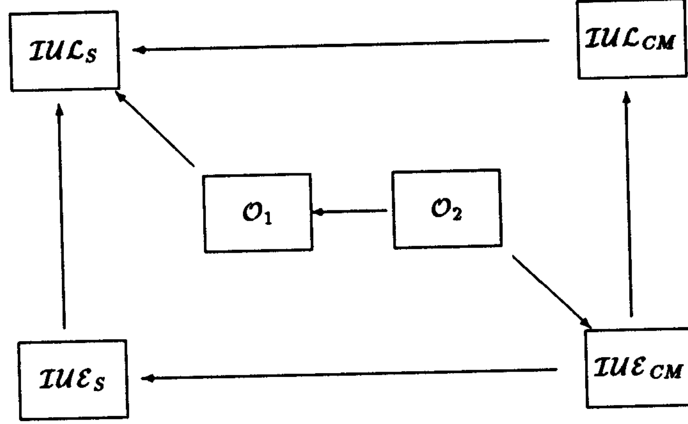


Figure 1: Relation of Refinements

To denote that the arrow relation does not hold, we use the symbol $\not\to$. It is straightforward to prove that the arrow relation is transitive. If the arrow relation holds for two models, we can conclude that the semantic equality of programs is related in the same way.

Lemma 6.2 *For two models $\mathcal{M}_1$ and $\mathcal{M}_2$,*

$$\mathcal{M}_1 \to \mathcal{M}_2 \;\Rightarrow\; \langle \forall F, G :: (\mathcal{M}_1[\![F]\!] = \mathcal{M}_1[\![G]\!]) \;\Rightarrow\; (\mathcal{M}_2[\![F]\!] = \mathcal{M}_2[\![G]\!])\rangle.$$

Proof: Suppose $\mathcal{M}_1 \to \mathcal{M}_2$. Then

$$
\begin{array}{ll}
 & (\mathcal{M}_1[\![F]\!] = \mathcal{M}_1[\![G]\!] \\
\equiv & \\
 & (F \sqsubseteq G \textbf{ in } \mathcal{M}_1) \wedge (G \sqsubseteq F \textbf{ in } \mathcal{M}_1) \\
\Rightarrow & \quad \{\ \mathcal{M}_1 \to \mathcal{M}_2\ \} \\
 & (F \sqsubseteq G \textbf{ in } \mathcal{M}_2) \wedge (G \sqsubseteq F \textbf{ in } \mathcal{M}_2) \\
\equiv & \\
 & (\mathcal{M}_2[\![F]\!] = \mathcal{M}_2[\![G]\!]
\end{array}
$$

□

In figure 1, the arrow relation is shown for all models given in the previous section. Arrows that are not given and do not follow from transitivity do not hold. In this section, we establish the arrow relations. Since the relations between the property-based models follow directly from the theory given in the previous sections, they will not be discussed here. We are especially interested in the relations between the operational semantics and the semantics based on Unity properties. Since the models $\mathcal{IUE}_{CM}$ and $\mathcal{O}_2$ are both compositional and contain some information about atomicity, one might think that these models are equivalent. Also, one might think that $\mathcal{O}_1$ and $\mathcal{IUL}_S$ are equivalent. In this section we will show that this is not true.

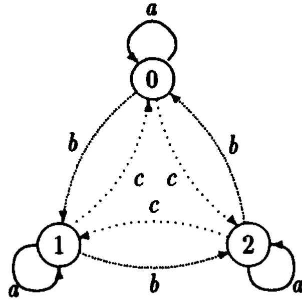
First, we are going to examine the relation between $\mathcal{O}_2$ and $\mathcal{IUE}_{CM}$. As can be seen from figure 1, there is a refinement-preserving abstraction relation from $\mathcal{O}_2$ to $\mathcal{IUE}_{CM}$, but not the other way around. We will start by proving the latter by counterexample.

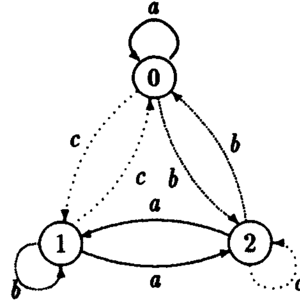
Theorem 6.3 $\quad \mathcal{IUE}_{CM} \not\to \mathcal{O}_2$.

Proof: Consider the following programs and their state transition diagrams.

Program F
  initially
    $x \in \{0,1,2\}$
  assign
    $x := x \bmod 3$
    $\| \; x := (x+1) \bmod 3$
    $\| \; x := (x+2) \bmod 3$
end$\{F\}$

Program G
  initially
    $x \in \{0,1,2\}$
  assign
    $x := -x \bmod 3$
    $\| \; x := (1-x) \bmod 3$
    $\| \; x := (2-x) \bmod 3$
end$\{G\}$





We start to show that $\mathcal{IUE}_{CM}[F] = \mathcal{IUE}_{CM}[G]$, in other words, that the properties of both programs are the same. Firstly, it is obvious that the initial sections of both programs are the same. Secondly, for $unless_{CM}$ properties:

$$
\begin{array}{ll}
 & p \; unless_{CM} \; q \textbf{ in } F \\
\equiv & \quad \{\text{definition}\} \\
 & [(p \wedge \neg q) \Rightarrow wlp.F.(p \vee q)] \\
\equiv & \quad \{\text{definition } wlp.\text{F}\} \\
 & [(p \wedge \neg q) \Rightarrow \langle \forall s : s \in F :: wlp.s.(p \vee q) \rangle] \\
\equiv & \quad \{\text{definition } wlp\} \\
 & [(p \wedge \neg q) \Rightarrow \langle \forall \alpha : \alpha \in \{0,1,2\} :: (p \vee q).((x+\alpha) \bmod 3/x) \rangle] \\
\equiv & \\
 & [(p \wedge \neg q) \Rightarrow \langle \forall y : y \in \{0,1,2\} :: (p \vee q).(y/x) \rangle] \\
\equiv & \\
 & [(p \wedge \neg q) \Rightarrow \langle \forall \alpha : \alpha \in \{0,1,2\} :: (p \vee q).((\alpha - x) \bmod 3/x) \rangle] \\
\equiv & \quad \{\text{derivation in reverse}\} \\
 & p \; unless_{CM} \; q \textbf{ in } G
\end{array}
$$

So, F and G have the same $unless_{CM}$ properties.

Also, the $ensures_{CM}$ properties are the same. This is outlined in table 1. This table shows whether or not a statement exists such that $[(p \wedge \neg q) \Rightarrow wlp.s.q]$ holds for the given predicates $(p \wedge \neg q)$ and q. A one in the table indicates that there exists such a statement, a zero denotes that such a statement does not exist, and $-$ indicates that the combination of predicates is not possible. The predicates $(p \wedge \neg q)$ and q are given by the set of states on which they are true. It is easy to check that the table holds for both programs F and G. Using symmetry arguments we may conclude that the existence of the statements is the same for all predicates.

From this table and the fact that the $unless_{CM}$ properties of both programs are the same, it can be concluded that the $ensures_{CM}$ properties are the same for both programs. So, we proved that $\mathcal{IUE}_{CM}[F] = \mathcal{IUE}_{CM}[G]$.

Although the properties are the same, the operational semantics $\mathcal{O}_2[F]$ and $\mathcal{O}_2[G]$ differ. The following extended sequence is an element of $\mathcal{O}_2[G]$ but not of $\mathcal{O}_2[F]$:

$$\langle\!\langle (0,1),(1,2),(2,0),(0,1),(1,2),\cdots \rangle\!\rangle .$$

q / $p \wedge \neg q$	$\emptyset$	0	2	01	12	012
$\emptyset$	1	1	1	1	1	1
0	0	–	1	–	1	–
01	0	–	0	–	–	–
012	0	–	–	–	–	–

Table 1: Existence of a statement s such that $[(p \wedge \neg q) \Rightarrow wlp.s.q]$.

This sequence is not fair in F because, to produce this sequence, the third statement of F should be ignored forever. To summarize, we have found that

$$(\mathcal{IUE}_{CM}[F] = \mathcal{IUE}_{CM}[G]) \wedge (\mathcal{O}_2[F] \neq \mathcal{O}_2[G])$$

So, by lemma 6.2, $\mathcal{IUE}_{CM} \not\rightarrow \mathcal{O}_2$. □

The same counterexample shows that $\mathcal{IUE}_{CM} \not\rightarrow \mathcal{O}_1$ and $\mathcal{IUE}_S \not\rightarrow \mathcal{O}_1$. In the proof above, it is shown that the notion of execution sequences is a stronger notion than Unity properties. Two programs having the same Unity properties may have different execution sequences. In other words, Unity properties are not expressive enough to characterize the set of execution sequences. As a consequence, property preserving refinement is a weaker notion than the reduction of the set of execution sequences. As we have seen in the proof, the programs F and G have the same Unity properties. So, $F \sqsubseteq G$ **in** $\mathcal{IUE}_{CM}$ and also $F \sqsubseteq G$ **in** $\mathcal{IUE}_S$. However, these refinements do not reduce the set of execution sequences. In fact, they extend the set of sequences. So, neither $F \not\sqsubseteq G$ **in** $\mathcal{O}_2$ nor $F \not\sqsubseteq G$ **in** $\mathcal{O}_1$.

Now we are going to prove an interesting relation between extended sequences and Unity properties: the arrow $\mathcal{O}_2 \rightarrow \mathcal{IUE}_{CM}$. The model $\mathcal{O}_2$ only contains information about state transitions and fairness. This proves to be sufficient to retain some information about the statements of a program, namely *ensures* properties. Intuitively this can be argued as follows. A statement is in fact a set of state transition, for each state it contains a transition. Programs with the same transitions may have different groupings. Due to fairness, this grouping is essential for the behavior of the program. For example, take two programs F and G with the same state transitions, but a different grouping, the transitions (a, b) and (c, d) are grouped into one statement for F and into different statements for G. Take an execution sequence with pairs starting in a and c, then, due to fairness, there is a pair (a, b) or (c, d) in this sequence, this is not necessary for G. Formally, the proof is:

Theorem 6.4 $\quad \mathcal{O}_2 \rightarrow \mathcal{IUE}_{CM}$.

Proof: We have to prove that for all Unity programs F and G

$$(I_F \supseteq I_G \wedge V_F \supseteq V_G) \Rightarrow (I_F \supseteq I_G \wedge U_F \subseteq U_G \wedge E_F \subseteq E_G),$$

where $(I_F, V_F) = \mathcal{O}_2[F]$, $(I_G, V_G) = \mathcal{O}_2[G]$, $(I_F, U_F, E_F) = \mathcal{IUE}_{CM}[F]$, and $(I_G, U_G, E_G) = \mathcal{IUE}_{CM}[G]$. The initial part is trivial and the *unless* part follows from lemma 4.11. We only look at the *ensures* part here, and assume that the *unless*$_{CM}$ properties of both programs are the same. We use contraposition:

$$\begin{array}{ll}
 & p\ ensures_{CM}\ q \ \textbf{in}\ F \wedge \neg(p\ ensures_{CM}\ q\ \textbf{in}\ G) \\
\equiv & \quad \{\text{ definition } ensures \} \\
 & p\ ensures_{CM}\ q\ \textbf{in}\ F \wedge \\
 & (\neg(p\ unless\ q\ \textbf{in}\ G) \vee \neg\langle \exists s : s \in G :: [(p \wedge \neg q) \Rightarrow wlp.s.q]\rangle) \\
\equiv & \quad \{\text{ predicate calculus }\} \\
 & (p\ ensures_{CM}\ q\ \textbf{in}\ F \wedge \neg(p\ unless_{CM}\ q\ \textbf{in}\ G)) \vee \\
 & (p\ ensures_{CM}\ q\ \textbf{in}\ F \wedge \langle \forall s \exists \sigma : s \in G :: (p \wedge \neg q).\sigma \wedge \neg wlp.s.q.\sigma\rangle \\
\equiv & \quad \{\text{ assumption } unless \}
\end{array}$$

$$
\begin{array}{ll}
 & p \; ensures_{CM} \; q \textbf{ in } F \wedge \langle \forall s \exists \sigma : s \in G :: (p \wedge \neg q).\sigma \wedge \neg wlp.s.q.\sigma \rangle \\
\Rightarrow & \quad \{ \text{ lemma 4.12 } \} \\
 & \langle \forall v : v \in V_F \wedge v = \langle\!\langle (\sigma_1, \sigma_1'), (\sigma_2, \sigma_2'), \cdots \rangle\!\rangle :: \langle \forall i : (p \wedge \neg q).\sigma_i \rangle \Rightarrow \langle \exists i : q.\sigma_i' \rangle \rangle \wedge \\
 & \langle \forall s \exists \sigma : s \in G :: (p \wedge \neg q).\sigma \wedge \neg wlp.s.q.\sigma \rangle \\
\Rightarrow & \quad \{ \text{ straightforward construction of } v \text{ } \} \\
 & \langle \forall v : v \in V_F \wedge v = \langle\!\langle (\sigma_1, \sigma_1'), (\sigma_2, \sigma_2'), \cdots \rangle\!\rangle :: \langle \forall i : (p \wedge \neg q).\sigma_i \rangle \Rightarrow \langle \exists i : q.\sigma_i' \rangle \rangle \wedge \\
 & \langle \exists v : v \in V_G \wedge v = \langle\!\langle (\sigma_1, \sigma_1'), (\sigma_2, \sigma_2'), \cdots \rangle\!\rangle :: \langle \forall i : (p \wedge \neg q).\sigma_i \rangle \wedge \neg q.\sigma_i' \rangle \\
\Rightarrow & \quad \{ \; v \text{ induces by second conjunct } \} \\
 & \langle \exists v : v \in ESeq :: v \notin V_F \wedge v \in V_G \rangle \\
\Rightarrow & \\
 & V_F \not\supseteq V_G.
\end{array}
$$

□

Next, we examine the relation between sequences and Sanders's logic. There is a clear relation between the properties *unless*$_S$ and $\mapsto_S$, and sequences of states corresponding the intuitive idea of the properties. These relations were given in the lemmas 4.7 and 4.8. This gives direct the following theorem.

Theorem 6.5 $\quad \mathcal{O}_1 \rightarrow \mathcal{IUL}_S$.

However, the reverse arrow does not hold.

Theorem 6.6 $\quad \mathcal{IUL}_S \not\rightarrow \mathcal{O}_1$.

In fact, the counter-example of theorem 6.3 is a counter-example for this theorem also. However, this is not the only cause of trouble. In [Mis90], Misra shows that the notion of ensuring is essential when program composition is examined. The following theorem shows that the *ensures* also provides a really finer distinction of sequences than the *leadsto* when programs are examined in isolation.

Theorem 6.7 *There are Unity programs F, G for which*

$$
\begin{array}{rcl}
\mathcal{O}_1[\![F]\!] & \neq & \mathcal{O}_1[\![G]\!], \\
\mathcal{IUL}_S[\![F]\!] & = & \mathcal{IUL}_S[\![G]\!], \\
\mathcal{IUE}_S[\![F]\!] & \neq & \mathcal{IUE}_S[\![G]\!].
\end{array}
$$

Proof: Consider the following programs

Program F	**Program** G
assign	**assign**
$x := x + 1$	$x := x + 1$
$[\!]\; x := x + 2$ **if** $even(x)$	$[\!]\; x := x + 2$ **if** $even(x)$
$[\!]\; x := x + 2$ **if** $odd(x)$	$[\!]\; x := x + 2$ **if** $odd(x)$
$[\!]\; x := x + 2$	**end**$\{G\}$
end$\{F\}$	

The programs F and G have the same state transitions, so the *unless*$_S$ relations are the same. Using lemma 4.8 one can prove that both programs have the same $\mapsto_S$ properties. The programs F and G have different sequences, i.e. $\mathcal{O}_1[\![F]\!] \neq \mathcal{O}_1[\![G]\!]$. For example, the sequence $\langle\!\langle 0, 1, 2, 3, \cdots \rangle\!\rangle$ is an element of $\mathcal{O}_1[\![G]\!]$, but it is not an element of $\mathcal{O}_1[\![F]\!]$. This difference can be expressed with the *ensures*$_S$ property

$$true \; ensures_S \; ((x \bmod 4 = 0) \vee (x \bmod 4 = 1))$$

which is property a property of F but not of G. □

In this section we have shown that the notion of sequences is a stronger notion than Unity properties. Unity properties cannot characterize the (extended) sequences of programs completely. As a consequence, the notion of property preserving refinement is a weaker notion than the reduction of execution sequences or reduction of nondeterminism. It is also shown that the *ensures* property, although it is too strong ([Mis90]), is essentially stronger than $\mapsto$ in characterizing sequences.

7 Conclusions and Further Research

We have defined a number of semantic models to justify refinement of Unity programs and compared the notions of refinement induced by the different models. We have shown that the two notions of sequences are more expressive than Unity properties. Programs that have the same properties may have different execution sequences. Consequently, preservation of Unity properties, as used by Sanders ([San90]) and Singh ([Sin91]), differs from the usual notion of refinement, reducing the set of execution sequences. It is a weaker notion that may introduce new execution sequences. We have also shown that extended sequences are stronger than Unity properties. I.e., it is possible to retain *unless* and *ensures* properties from extended sequences.

Unity properties have proven to be insufficient to characterize sequences. The real expressive power of properties is not clear. We want to find a model of execution sequences that is equivalent to properties. It also might be of interest to find a Unity-like property model that is powerful enough to characterize sequences. As we have seen, $\mathcal{IUL}_S$ is not compositional, neither is $\mathcal{IUE}_S$. It is interesting to know whether $\mathcal{IUE}_{CM}$ is the fully abstract model above $\mathcal{IUL}_S$.

Acknowledgements

We like to thank the Calculi for Distributed Program Construction Club headed by Lambert Meertens and Doaitse Swierstra and the Formal Models Club at Utrecht University. We also want to acknowledge Patrick Lentfert, Frans Rietman, Beverly Sanders, David Meier, Kaisa Sere, Nissim Francez, Ted Herman, Jan van de Snepscheut, Harm Peter Hofstee and Robert Harley for their comments, discussions and carefully reading of preliminary versions, and Wim Hesselink for suggestions and references to the literature.

References

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proc. of the 3rd Annual IEEE Symp. on Logic in Computer Science*, pages 165–175, Washington D.C., July 1988. Computer Society Press.

[Bac90] R.-J.R. Back. Refinement calculus, part II: Parallel and reactive programs. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, pages 67–93. Springer-Verlag, 1990.

[BKPR91] F.S. de Boer, J.N. Kok, C Palamidessi, and J.J.M.M. Rutten. The failure of failures in a paradigm of asynchronous communication. In J.C.M. Baeten and J.F. Groote, editors, *CONCUR '91, Proceedings of the 2nd International Conference on Concurrency Theory*, pages 111–126. Springer-Verlag, August 1991.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation*. Addison-Wesley Publishing Company, Inc., 1988.

[DS90] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 1990.

[Liu89] Z. Liu. A semantic model for UNITY. Technical Report 144, Computer Science Department, University of Warwick, August 1989.

[Mis90] J. Misra. The importance of ensuring. *Notes on UNITY*, 11-90, January 1990.

[Pac90] J. Pachl. Three definitions of *leads-to* for UNITY. *Notes on UNITY*, 23-90, December 1990.

[Pac92] J. Pachl. A simple proof of a completeness result for *leads-to* in the UNITY logic. *Information Processing Letters*, 41:35–38, 1992.

[San90] B.A. Sanders. Stepwise refinement of mixed specifications of concurrent programs. In M. Broy and Jones C.B., editors, *Proceedings of the IFIP Working Conference on Programming and Methods*, pages 1–25. Elsevier Science Publishers B.V. (North Holland), May 1990.

[San91] B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.

[Sin91] A.K. Singh. Parallel programming: Achieving portability through abstraction. In *11th International Conference on Distributed Computing Systems*, May 1991.