Program Refinement in UNITY-like Environments

Programma verfijning in UNITY-achtige Omgevingen (met een samenvatting in het Nederlands)

PROEFSCHRIFT
ter verkrijging van de graad van doctor aan de Universiteit Utrecht
op gezag van de Rector Magnificus, Prof. dr. J.A. van Ginkel,
ingevolge het besluit van het College van Decanen
in het openbaar te verdedigen
op vrijdag 15 september 1995 des ochtends te 10:30 uur

door

Rob Theodorus Udink
geboren op 2 december 1967
te Terborg
CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Udink, Rob Theodorus

Program refinement in UNITY-like environments / Rob Theodorus Udink. - Utrecht : Universiteit Utrecht, Faculteit Wiskunde en Informatica
Proefschrift Universiteit Utrecht. - Met index, lit. opg.
- Met samenvatting in het Nederlands.
ISBN 90-393-0959-0
Trefw.: computerprogramma’s ; ontwerpen / parallel verwerking.

NWO

These investigations were supported by the Netherlands Computer Science Research Foundation (SION) with the financial support from the Netherlands Organization for Scientific Research (NWO).
Contents

Acknowledgements iii

1 Introduction 1

2 Preliminaries 7
   2.1 Refinement Calculus ........................................... 7
   2.2 The UNITY Framework .......................................... 14

3 Program Refinement in UNITY 21
   3.1 Operational Semantics .......................................... 24
      3.1.1 Refinement ................................................ 30
      3.1.2 Best Compositional Refinement ............................. 30
   3.2 Property based Semantics ...................................... 33
      3.2.1 Refinement ................................................ 34
      3.2.2 Best Compositional Refinement ............................. 35
   3.3 Sequences versus Properties ................................... 37
   3.4 Conclusions .................................................... 41

ImpUNITY 43

4 Interference 45
   4.1 Modifiers ...................................................... 45
   4.2 Modifiers in ImpUNITY ......................................... 49
   4.3 The ImpUNITY logic ............................................ 52
   4.4 Program Transformation Rules .................................. 63
   4.5 Conclusions .................................................... 69

5 Observability 71
   5.1 Observability .................................................. 71
   5.2 Observability in ImpUNITY ...................................... 76
   5.3 Program Transformation Rules .................................. 78
## Contents

5.3.1 Rescheduling Stutterings ........................................ 82
5.3.2 Refining Statements ................................................ 87
5.4 Data Refinement ......................................................... 90
5.5 Refinement of Modules ................................................ 95
5.6 Conclusions ............................................................... 98

6 Case Study: Register Refinement ....................................... 101
  6.1 Registers in ImpUNITY ............................................... 102
  6.2 Refinement of a Register .......................................... 104
  6.3 Conclusions ............................................................. 108

7 Procedures ................................................................. 109
  7.1 Procedures ............................................................ 109
  7.2 Procedures in ImpUNITY .......................................... 112
  7.3 Program Transformation Rules .................................... 117
  7.4 Conclusions ............................................................. 121

8 Case Study: A Memory Component ..................................... 123
  8.1 The Procedure Interface ........................................... 123
  8.2 A Memory Component ............................................... 125
  8.3 The RPC Component ................................................ 128
  8.4 Implementation of the Memory .................................... 131
  8.5 Conclusions ............................................................. 133

A UNITY Theorems .......................................................... 135

B ImpUNITY ................................................................. 139

Bibliography ........................................................................ 151

Glossary ............................................................................. 159

Index .................................................................................. 161

Curriculum Vitae .................................................................. 169
Here I want to acknowledge everyone who supported me in preparing this thesis. First of all, I want to thank my promotores Joost Kok and Doaitse Swierstra for their advice, encouragements and comments on preliminary versions of this thesis. I am particularly grateful to my supervisor Joost Kok. With his enthusiasm, positive attitude and good advice he managed to guide and motivate me at moments I was lost. I want to thank him for his patience and the effort he put in improving the numerous preliminary versions of this thesis and our papers.

The department of computer science at the Utrecht University has been a pleasant place to work and I want to thank all my colleagues, especially Frans Rietman, Wishnu Prasetya, Patrick Lentfert, Ted Herman, Carroll Morgan, Ralph Back and Kaisa Sere for their listening ears and helping hands. Ralph Back and Kaisa Sere are acknowledged for their hospitality during my stay at the Åbo Akademi University. Acknowledgements are also made to the participants of the Wednesday afternoon meetings (which also had a lot of other names during the last four years) for sharing their interest and knowledge in the mathematical foundations of programming. Furthermore, I want to acknowledge Susanne Biemans, Pierre Collette, Henk Goeman, Ted Herman, Peter Hofstee, Patrick Lentfert, Wishnu Prasetya, Frans Rietman, Beverly Sanders, Kaisa Sere, and the anonymous referees for their useful suggestions and careful reading of preliminary versions of this thesis or the articles on which this thesis has been based. I want to thank the review committee consisting of Prof. Dr. R.J.R. Back, Dr. F. de Boer, Prof. L.G.L.T. Meertens and Prof. Dr. J.-J. Ch. Meyer for reviewing and giving comments on the manuscript of this thesis.

The department has not only been a pleasant place to work, it has also been a very nice place to be, especially thanks to my fellow PhD-students and the numerous non-scientific activities we did together: drinking K2V coffee, hacking quasi-serious applications, having diners, watching movies, playing squash and sailing. Sharing a room with Frans Rietman has been a very nice experience, professionally as well as personally. Finally, I want to thank my parents for their continuous stimulation and Susanne for her moral support and understanding during the last phase of writing my thesis.
Chapter 1

Introduction

In general terms, this thesis is about the formal design and implementation of parallel and distributed systems by stepwise refinement in a compositional way. More specifically, we extend the UNITY framework with new notions of program refinement and features that increase the possibilities for compositional program design.

A parallel or distributed system is a system in which a number of interconnected computers can cooperate [DC86]. Nowadays, many people use distributed systems. A well-known example of a distributed system is the automatic teller machine, which makes it possible to draw money from an account at a bank at (almost) any time and (almost) any place in the world. Techniques and infrastructure for communication, like fibreglass networks, satellite communication and compression techniques, are being developed rapidly. These developments enlarge the possibilities for building more complex distributed systems and, therefore, make the investigation of the foundations of distributed systems of increasing importance.

Although the human body, and especially the human brain, is full of parallel processes that we seem to control without effort, thinking about parallel processes is a difficult task. Experience has shown that the design and implementation of parallel and distributed programs is more difficult than the design and implementation of sequential programs. Formal methods can help in the design of this kind of programs. For example, they can be used to write unambiguous specifications and to guarantee the correctness of implementations, and in general they can help to get a better understanding of distributed systems.

Several formalisms supporting the design of parallel and distributed systems have been proposed and each formalism has its own special features. For example, process algebras [BW90], CSP [Hoa85], LOTOS [ISO87] and data flow networks [Kah74] are used to reason about networks of processes that communicate with each other by sending messages. The UNITY framework [CM88], the action system formalism [BKS83, BKS88] and TLA [Lam91] develop abstract programs that can be mapped to distributed ar-
chitectures. In this thesis we focus on the UNITY framework and the action system formalism. First we give a short overview of these formalisms.

![Diagram](image)

**Figure 1.1.** Program Design in UNITY

The UNITY framework was introduced by Chandy and Misra [CM88]. The goal of the framework is to provide a relative “small,” formal framework for the design of parallel and distributed systems independently of the target architecture. The UNITY framework consists of a programming language and a programming logic. The programming language is simple; in essence, a program consists of statements that are executed repeatedly in a nondeterministic but fair order. The UNITY logic consists of temporal properties that are used to specify and to reason about programs. The basic idea of program design in UNITY is shown in figure 1.1. One starts by writing a specification $Spec_1$ consisting of a set of temporal properties that the program must satisfy. This specification is refined in a number of steps to a new specification $Spec_2$ such that $Spec_2$ implies $Spec_1$. In the steps from $Spec_1$ to $Spec_2$ one can use the various theorems that are given in the UNITY book [CM88]. Often, the steps are about algorithmic problems, and are usually independent of the target architecture. As soon as the specification is specific enough, a UNITY program $Prog_1$ is developed, and it is shown that the program satisfies the specification. The program is then mapped on a target architecture. Since mappings are not formally supported by the UNITY framework, the mappings must be as simple as possible. For several program schemes mappings to different architectures have been proposed. However, often the program does not have a proper shape for obtaining a simple mapping to a specific target architecture. In that case, there are two ways to proceed within the UNITY framework. First, one can refine the last specification ($Spec_2$) to obtain a new specification $Spec_3$. For this specification a new program $Prog_2$ can be derived that can be mapped more easily onto the target architecture. It can be difficult to find a new specification since the problems about the mapping are more difficult to handle at the level of specifications than at the level of programs. A different approach is to find a new UNITY program $Prog_2$ directly, and prove it correct with respect to
specification $Spec_2$. One of the problems of the UNITY framework is that it does not support a method that allows $Prog_1$ to be transformed to $Prog_2$ while preserving the correctness with respect to the specification.

The action system formalism was originally proposed by Back and Kurki-Suonio [BKS83, BKS88] and further extended by Back et al. [Bac90, Ser90, BS94a, BvW94]. It lifts the standard refinement calculus for reasoning about sequential programs to a framework for reasoning about parallel and distributed systems. Like the UNITY framework, the action system formalism models distributed systems by a program consisting of a set of statements, and in this way abstracts from specific architectures. The action system formalism supports the refinement of programs and has two notions of refinement: preservation of total correctness and preservation of temporal behaviour. The first notion is based on the standard notion of refinement for sequential programs. Simulation, based on data refinement, is the method to preserve temporal properties of an action system. The initial specification is already a program and an implementation is derived by stepwise refinement of programs. The programming language is richer than the UNITY programming language, since it also supports termination of programs, local variables and procedure call mechanism [BS94a]. These features are important for a compositional way of program refinement and for the simplification of the mapping to the target architecture.

In this thesis we construct a framework that supports a compositional notion of refinement both at the level of specifications and at the level of programs. First, we focus on the standard UNITY framework. We study how program refinement can be integrated in the UNITY framework and we examine the possibilities for compositional program design. Second, we combine the UNITY framework and the action system formalism to increase the possibilities for compositional refinement. Refinement of specifications is taken from the UNITY framework, and the action systems formalism provides the tools for refinement of programs. Although the programming languages of the UNITY framework and the action systems formalism have many things in common, there are also differences. First, the language of the action system formalism supports local variables and procedure calls. Since these features play an essential role in program refinement, we examine how these features of action systems can be added to UNITY. We also extend the UNITY logic to a new UNITY-like logic that is better suited for compositional reasoning. Second, in the UNITY framework there is a fairness restriction on the execution of statements. Hence, we have to adapt the program transformation rules of the action system formalism.

The combination of the UNITY formalism and the action system formalism results in a design process that is depicted in figure 1.2. First a specification in terms of temporal properties is given and this specification is refined in the UNITY style to solve algorithmic problems. Then a program is derived. In case a program does not match
a specific target architecture, transformation rules like the ones in the action system formalism are used to transform the program without going back to the specification level. Furthermore, in all stages of the refinement process we want to be able to split specifications or programs into components and refine the components independently.

This thesis is organised as follows. In chapter 2 we give an overview of the two formalisms: we present the main elements of Back's refinement calculus, which forms the basis for the action system formalism, and we discuss the UNITY framework and some extensions of this framework that have been proposed in literature. Then, we present a study about compositional program refinement in the UNITY framework. This study consists of two parts. The first part, chapter 3, studies the semantics of UNITY programs. The second part, comprising chapters 4, 5 and 7, examines how the UNITY framework can be extended to increase the possibility for compositional reasoning. This results in the ImpUNITY framework, which supports new properties and a notion of program refinement.

Chapter 3 deals with the semantics of UNITY programs and gives an overview of compositional program refinement in UNITY. By defining different semantic models, we formalise the semantics of UNITY programs. These semantic models fall into two categories. The first category consists of operational models based on sequences of states that occur during the execution of a program. Models in the second category are based on UNITY properties. For each category we give a compositional semantics. Based on these semantic models, we define different notions of program refinement. The compositional models induce compositional notions of refinement in the sense that they allow programs to be refined in any environment. Moreover, we show that these notions are, in a certain sense, the best compositional notions of refinement in their category. Then we take a closer look at the relation between properties in the UNITY logic and the operational semantics of programs.

By adding more structure to the way UNITY programs are composed, more tools for
compositional program refinement become available. This is done in chapters 4, 5 and 7. In these chapters we introduce the ImpUNITY framework. The ImpUNITY framework supports three features for structuring program union: a way to restrict interference of an environment, a way to restrict observations about the state space, and a procedure call mechanism. These new features not only increase the possibilities of compositional reasoning, they also make some low-level programming constructs available in the language. This simplifies (informal) mappings from ImpUNITY programs to different target architectures. We present the ImpUNITY framework in three chapters, and in each chapter we present a new feature.

Chapter 4 deals with external interference. The restriction of interference is modelled by modifiers and the UNITY programming language is extended with a new section for specifying that parts of the state space can not be changed by an environment. Furthermore, interference is incorporated in the ImpUNITY logic. This logic is a UNITY like logic in the sense that theorems about UNITY properties also hold for ImpUNITY properties. Moreover, the ImpUNITY properties are more suitable for compositional reasoning than the UNITY properties. We derive a set of program transformation rules based on the preservation of ImpUNITY properties.

Chapter 5 introduces a notion of observability. Observability concerns the way in which the state space can be observed. The restriction of observability is modelled by views and the UNITY programming language is extended with a new section for specifying that parts of the state space can not be observed by an environment. Using the notion of observability in the ImpUNITY framework, we derive a set of program transformation rules based on the preservation of ImpUNITY properties under some abstraction. These rules allow stutterings in a program to be rescheduled. By restricting both observability and interference, it is possible to specify that variables are local to a program. As is known from other formalisms, local variables may be used to introduce notions of data refinement, i.e. modifications of programs that allow local variables to be replaced and modified. We show how the idea of forward data refinement from action systems can be used in the ImpUNITY framework too.

Chapter 7 introduces a remote procedure call mechanism, a mechanism for communication between ImpUNITY components. The procedure call mechanism gives a new way of communication between ImpUNITY programs. By using this mechanism, more variables in a program can become local variables. We lift the action system approach [BS94a, BS94b] to the ImpUNITY framework. One of our motivations for this extension was the Dagstuhl Specification Problem [Dag94] which asks for a specification of components that communicate by procedures. This specific problem is discussed in chapter 8.

We present two case studies illustrating the use of the ImpUNITY framework. The first is about the refinement of a so-called safe register and shows how the interference and observability restrictions work. This case study is given in chapter 6. The second is
about the refinement of a memory component showing the usefulness of the procedure
call mechanism and is presented in chapter 8. Both cases studies are based on the
application of the program transformation rules.

There are two appendices. In appendix A we give a short overview of the standard
UNITY theorems used in the thesis. In appendix B we give an overview of the complete
ImpUNITY framework.

This thesis is based on material presented at conferences and in journal articles. The
semantic models for UNITY programs are based on the models presented in [UK93a,
UK93b]. The results about the best compositional notions of refinement are based the
full abstractness results presented in [UK93c]. The comparison between operational
models and property based models is studied in [UK93d]. The article [UHK94] is the
base for chapter 4, 5 and 6. The procedure call mechanism for ImpUNITY was presented
in [UK94] and [UK95], and the case study of chapter 8 stems from [UK94].
Chapter 2

Preliminaries

In this chapter we give an overview of the two main formalisms that are used in this thesis. Section 2.1 deals with the refinement calculus, a formalism for the development of sequential imperative programs. It forms the basis of the action system formalism. In section 2.2 the UNITY framework is presented. The UNITY framework is a formal framework for the design of parallel and distributed programs. We give an overview of the original framework and some modifications that are proposed in the literature.

2.1 Refinement Calculus

In this section we give a short overview of important elements in the refinement calculus [Bac78, BvW90b, Bac93a, MV94]. There are several ways to formalise the refinement calculus. Here we follow the lattice theoretical approach as given in Back [Bac93a].

Predicates and predicates transformers are defined as pointwise extensions of the lattice of truth values $\text{Bool} \overset{\text{def}}{=} \{F, T\}$. Ordered by the implication ordering, this domain is a complete boolean lattice.

For a state space $\Sigma$, predicates on $\Sigma$ are elements of $\text{Pred}_\Sigma \overset{\text{def}}{=} \Sigma \to \text{Bool}$ and are denoted by $p$, $q$, and $r$. The truth values and boolean operations are extended to operations on predicates as follows:

- $\text{false}(\sigma) \overset{\text{def}}{=} F$,
- $\text{true}(\sigma) \overset{\text{def}}{=} T$,
- $p \land q(\sigma) \overset{\text{def}}{=} p(\sigma) \land q(\sigma)$,
- $p \lor q(\sigma) \overset{\text{def}}{=} p(\sigma) \lor q(\sigma)$,
- $\neg p(\sigma) \overset{\text{def}}{=} \neg(p(\sigma))$.

The ordering on $\text{Bool}$ is extended pointwise to the domain of predicates:

- $p \leq q \overset{\text{def}}{=} (\forall \sigma : \sigma \in \Sigma : p(\sigma) \Rightarrow q(\sigma))$. 


Following Dijkstra and Scholten [DS90], we sometimes use square brackets for quantification over the state space, so \( p \leq q = [p \Rightarrow q] \).

Domain \( PTrans_{\Sigma,\Sigma'} \defeq \text{Pred}_\Sigma \rightarrow \text{Pred}_{\Sigma'} \) is the domain of predicate transformers from state space \( \Sigma \) to state space \( \Sigma' \). A predicate transformer \( PT \) is monotonic if for all predicates \( p \) and \( q \), \((p \leq q) \Rightarrow (PT(p) \leq PT(q))\), and \( MTrans_{\Sigma,\Sigma'} \defeq \text{Pred}_\Sigma \rightarrow_m \text{Pred}_{\Sigma'} \) is the domain of monotonic predicate transformers. Monotonic predicate transformers are also called commands and are denoted by \( A \) and \( B \). The domain of commands is the basic domain of the refinement calculus. Commands are interpreted as weakest precondition predicate transformers: for a command \( A \) and a predicate \( q \), predicate \( A(q) \) is the weakest predicate \( p \) such that execution of \( A \) starting in a state satisfying \( p \) is guaranteed to terminate in a state satisfying \( q \).

Back also gives a language for describing commands. The syntax of the language is as follows, and its semantics in terms of predicate transformers is given below.

\[
A ::= (f) \mid [p] \mid \{p\} \mid \bigwedge_{i \in I} A_i \mid \bigvee_{i \in I} A_i \mid A_1; A_2,
\]

where \( f \) is a function on states, \( p \) is a predicate and \( I \) is an arbitrary index set. The language is typed, but we omit the details about the typing. It can be shown that all commands in this language are monotonic predicate transformers and that all monotonic predicate transformers can be expressed in this language [BvW90a].

A state transformer \( f \) is a function in \( \Sigma \rightarrow \Sigma' \) and can be lifted to the update command \( (f) \in MTrans_{\Sigma,\Sigma} \) by

\[
((f)q)(\sigma) \defeq q(f(\sigma)).
\]

So, update command \( (f) \) establishes postcondition \( q \) for an initial state \( \sigma \) if and only if \( q \) holds in \( f(\sigma) \). A state predicate \( p \) can be lifted to a guard command by

\[
[p](q) \defeq (p \Rightarrow q),
\]

or to an assert command defined by

\[
\{p\}(q) \defeq (p \land q).
\]

From a set of commands \( \{A_i \mid i \in I\} \), a new command can be constructed using demonic composition \( \land \) and angelic composition \( \lor \) as follows.

\[
(\bigwedge_{i \in I} A_i)(q) \defeq (\forall i : i \in I : A_i(q)),
\]

\[
(\bigvee_{i \in I} A_i)(q) \defeq (\exists i : i \in I : A_i(q)).
\]

The sequential composition \( ; \) of commands \( A_1 \) and \( A_2 \) results in a command defined by

\[
(A_1; A_2)(q) \defeq A_1(A_2(q)).
\]
The commands \( \text{skip} \), \( \text{magic} \) and \( \text{abort} \) are often used in the refinement calculus and are defined by

\[
\begin{align*}
\text{skip}(q) & \overset{\text{def}}{=} q, \\
\text{magic}(q) & \overset{\text{def}}{=} \text{true}, \\
\text{abort}(q) & \overset{\text{def}}{=} \text{false}.
\end{align*}
\]

These commands can also be expressed by as \( \text{skip} = \{\text{true}\} = \{\text{true}\} \), \( \text{magic} = \{\text{false}\} \) and \( \text{abort} = \{\text{false}\} \). Hence, the guard command \([p]\) behaves as \( \text{skip} \) when \( p \) holds; otherwise as \( \text{magic} \). The assert command \([p]\) behaves as \( \text{skip} \) when \( p \) holds; otherwise as \( \text{abort} \).

Another way to define the language of commands is to lift binary relations on states to commands. For a relation \( R \subseteq \Sigma \times \Sigma' \to \text{Bool} \) this can be done in two ways. It can be lifted to an angelic update command:

\[
\{R\}(p)(\sigma) \overset{\text{def}}{=} \exists \sigma' : R(\sigma, \sigma') : p(\sigma'),
\]

or to a demonic update command:

\[
[R](p)(\sigma) \overset{\text{def}}{=} \forall \sigma' : R(\sigma, \sigma') : p(\sigma').
\]

The ordering on commands is defined as the pointwise extension of the ordering on predicates:

\[
A \leq A' \overset{\text{def}}{=} \forall p : p \in \text{Pred}_\Sigma : A(p) \leq A'(p).
\]

This ordering is called the refinement ordering and \( A \leq A' \) expresses that command \( A' \) is a refinement of \( A \). Using this ordering, refinement of commands corresponds to preservation of total correctness. Another notion of refinement in the refinement calculus is refinement through a command. For a command \( B \), refinement through \( B \) is defined by

\[
A \leq_B A' \overset{\text{def}}{=} B; A \leq A'; B.
\]

This form of refinement is useful for modelling data refinement [Wri94].

Commands can have different properties. A command \( A \) is said to be terminating if \( A(\text{true}) = \text{true} \) and it is non-miraculous if \( A(\text{false}) = \text{false} \). Command \( A \) is conjunctive if \( A(\forall i : i \in I : p_i) = (\forall i : i \in I : A(p_i)) \) for an arbitrary non-empty set of predicates \( \{p_i : i \in I\} \) and disjunctive if \( A(\exists i : i \in I : p_i) = (\exists i : i \in I : A(p_i)) \). A command that is both conjunctive and terminating is called universally conjunctive, and a command that is disjunctive and non-miraculous is called universally disjunctive. A terminating, non-miraculous, conjunctive command is called a statement and is denoted by \( S \) and \( T \).

For a command \( A \), the dual or conjugated command \( A^\circ \) is defined by

\[
A^\circ(p) \overset{\text{def}}{=} \neg A(\neg p).
\]
The intuition is that dualisation exchanges demonic and angelic nondeterminism as well as miracles and nontermination. For a statement $S$ that is interpreted as a weakest precondition predicate transformer, predicate $S(p)$ corresponds to the set of initial states in which execution of $S$ must terminate in a state satisfying $p$. Then, the predicate $S^\circ(p)$ corresponds to the set of states in which execution of $S$ may terminate in $p$. A statement $S$ is called deterministic if these sets coincide, i.e. $S = S^\circ$.

We also need the notion of adjoint pairs. The commands $A$ and $B$ form an adjoint pair if

$$A; B \leq \text{skip} \quad \text{and} \quad \text{skip} \leq B; A.$$  

Command $B$ is called the right adjoint of $A$ and $A$ is called the left adjoint of $B$. A command $A$ has a left adjoint if and only if $A$ is universally conjunctive; the left adjoint is unique and is denoted by $A_L$. Dually, command $A$ has a right adjoint if and only if $A$ is universally disjunctive; the right adjoint is unique and is denoted by $A_R$. For an adjoint pair $(A, B)$, the following property is often used in proofs:

$$[A(p) \Rightarrow q] = [p \Rightarrow B(q)].$$

This concludes the general overview of the basic refinement calculus. Now, we give some special definitions, abbreviations and theorems that are of use in the rest of this thesis.

First, we define the notion of equivalence commands, commands that are based on equivalence relations. In the refinement calculus we define them in the following way.

**Definition 2.1** A command $A$ is called a disjunctive equivalence command if it is universally disjunctive and

$$\text{skip} \leq A,$$

$$A; A = A,$$

$$A_R = A^\circ.$$  

For a relation $R \in \Sigma \times \Sigma \rightarrow \text{Bool}$, the command $\{R\}$ is a disjunctive equivalence command if $R$ is a total equivalence relation. An equivalence command can be seen as an action that can change the state to an arbitrary equivalent state. To reason about equivalence commands, we give the following lemma.

**Lemma 2.2** For disjunctive equivalence command $A$,

$$A(\text{true}) = \text{true},$$

$$A_R; A = A.$$  

**Proof:** Since $A$ is a refinement of $\text{skip}$, it is straightforward to prove that $A$ is terminating. Furthermore,
\[ A_R; A \]
\[ \leq \{ \text{skip } \leq A \} \]
\[ A \]
\[ \leq \{ \text{adjoints} \} \]
\[ A_R; A; A \]
\[ = \{ \text{definition} \} \]
\[ A_R; A \]

\[ \square \]

**Lemma 2.3** For disjunctive equivalence command \( A \) and terminating disjunctive command \( B \) such that \( B \leq A \),

\[
\begin{align*}
B; A & = A, \\
B^\circ; A & = A, \\
B(p \wedge A(q)) & = B(p) \wedge A(q), \\
B^\circ(p \wedge A(q)) & = B^\circ(p) \wedge A(q).
\end{align*}
\]

**Proof:** Since \( B \) is terminating and disjunctive, we can derive \( B^\circ \leq B \). Then, the first two items of the lemma are proven as follows.

\[
\begin{align*}
B; A & \\
\leq & \{ B \leq A \} \\
A; A & \\
= & \{ A \text{ equivalence command} \} \\
A & \\
= & \{ \text{lemma 2.2} \} \\
A^\circ; A & \\
\leq & \{ B \leq A \} \\
B^\circ; A & \\
\leq & \{ \text{above} \} \\
B; A &
\end{align*}
\]

Furthermore,
\( B(p \land A(q)) \)

\[ \Rightarrow \{ \text{monotonicity} \} \]

\( B(p) \land (B; A)(q) \)

\[ = \{ \text{above} \} \]

\( B(p) \land A(q) \)

\[ = \{ B \text{ disjunctive, predicate calculus} \} \]

\((B(p \land A(q)) \lor B(p \land \neg A(q))) \land A(q)\)

\[ \Rightarrow \{ \text{monotonicity, predicate calculus} \} \]

\( B(p \land A(q)) \lor (B(\neg A(q)) \land A(q)) \)

\[ = \{ \text{definition conjugated} \} \]

\( B(p \land A(q)) \lor (\neg(B^o; A(q)) \land A(q)) \)

\[ = \{ \text{above} \} \]

\( B(p \land A(q)) \lor (\neg A(q) \land A(q)) \)

\[ = \{ \text{predicate calculus} \} \]

\( B(p \land A(q)) \)

Since \( B \) is disjunctive, \( B^o \) is conjunctive. Then, \( B^o(p \land A(q)) = B^o(p) \land A(q) \) follows directly from the second item of the lemma. \( \square \)

The notion of conjunctive equivalence commands is the dual of the notion of disjunctive equivalence commands, and properties like the ones given in lemma 2.2 can be derived from this duality.

The language of Back is complete for monotonic predicate transformers, in the sense that every command can be described in this language. However, it is convenient to introduce some shorthands.

- If a demonic composition consists of only two elements, say \( A_1 \) and \( A_2 \), then we write \( A_1 \land A_2 \) for the demonic composition. Similar, we write \( A_1 \lor A_2 \) for angelic composition of \( A_1 \) and \( A_2 \).

- For a command \( A \), command \( A^* \) corresponds to a demonic choice between all commands that are arbitrary but finite number of executions of \( A \). So, \( A^* \overset{\text{def}}{=} \bigwedge_{i \in \mathbb{N}} A^i \) where \( A^0 \overset{\text{def}}{=} \text{skip} \), and for \( i \in \mathbb{N} \), \( A^{i+1} \overset{\text{def}}{=} A; A^i \).
• The if-statement
\[
\text{if } p_1 \rightarrow S_1 \\
\quad \text{[} p_2 \rightarrow S_2 \\
\quad \cdots \\
\quad \text{fi}
\]
is an abbreviation for
\[
(\land_{i \in \{1,2,\ldots\}}([p_i];S_i)) \land ([\forall i : i \in \{1,2,\ldots\} : \neg p_i]).
\]
The if-statement chooses nondeterministically one of its branches of which the guard evaluates to true. When all the guards are false, then the statement skips. This is different from a more standard if-statement that would abort when every guard is false. The if-statement is terminating, non-miraculous and conjunctive. We define the if-statement in this way because it corresponds to the one used in the UNITY framework.

• The statement \( p \rightarrow S \) is an abbreviation of \( \text{if } p \rightarrow S \text{ fi} \) and is called a guarded statement.

In the sequel, we use state spaces that are based on sets of variables, i.e., a state is a function from variables to some value domain \( Vals \). Let \( Vars \) be the set all of program variables. Elements of \( Vars \) are denoted by \( x, y \) and \( z \), subsets of \( Vars \) by \( X, Y \) and \( Z \), and the complement of a set \( X \) is defined by \( X^c \triangleq Vars \setminus X \). For a set of program variables \( X \subseteq Vars \), the state space \( \Sigma_X \triangleq X \rightarrow Vals \) is the state space on \( X \). For commands on this kind of state spaces we define the following shorthands.

• The multiple assignment statement is a special case of the update command. For example, for variables \( x_1, \ldots, x_n \) and expressions \( e_1, \ldots, e_2 \) (functions from states to values) the multiple assignment \( x_1, \ldots, x_n := e_1, \ldots, e_2 \) is equivalent to the command \( \langle f \rangle \), where \( f \) assigns the value of expression \( e_i \) to variable \( x_i \) for \( 1 \leq i \leq n \). The assignment \( x_1, \ldots, x_n := e_1, \ldots, e_2 \) is also denoted by \( x := e_1 | \cdots | x_n := e_n \). For a set of values \( E \), the nondeterministic assignment of some value \( e \in E \) to a variable \( x \) is defined by \( x :\in E \triangleq \land_{e \in E} x := e \).

• For a command \( A \), and a variable \( x \) in the state space of \( A \), hiding \( x \) in a block construct is denoted by

\[
[\text{var } x := e. \ A]
\]
and is defined by \( \langle f \rangle; A; \langle g \rangle \), where \( f \) introduces variable \( x \) in the state space and assigns its initial value: \( f(\sigma)(x) = e(\sigma) \) and \( f(\sigma)(y) = \sigma(y) \) for all other variables \( y \in X \). Function \( g \) removes variable \( x \) from the state space.
• The assignment statements can be used to model renaming. For a predicate $p$, renaming of variable $x$ by $y$ in $p$ is defined by $p[y/x] \overset{\text{def}}{=} (x := y)(p)$. Renaming of $x$ by $y$ in a command $A$ not containing $y$ is defined by $A[y/x] \overset{\text{def}}{=} y := x; A; x := y$. Substitution of lists of variables is modelled by multiple assignments. In the sequel, we also use substitution for sets of variables and in these cases they should be interpreted as lists.

• For a set of program variables $Y \subseteq Vars$, we define a pair of equivalence commands $\{\forall Y\}$ and $[\forall Y]$ that change the values of variables in $Y$ in an angelic way and in a demonic way, respectively. These commands are defined by

\[
\{\forall Y\} \overset{\text{def}}{=} \{R\}, \text{ and } [\forall Y] \overset{\text{def}}{=} [R],
\]

where $R(\sigma, \rho) \overset{\text{def}}{=} (\forall y \in Y^c: \sigma(y) = \rho(y))$. This pair of commands forms an adjoint pair. As shorthands we define the commands $\{= Y\} \overset{\text{def}}{=} \{\forall Y^c\}$ and $[= Y] \overset{\text{def}}{=} [\forall Y^c]$ that can change the values of all variables except the variables in $Y$. (Recall that $Y^c$ is the complement of $Y$ in the set of program variables.)

For equivalence commands based on variables we can derive the following lemmas.

**Lemma 2.4** For sets of variables $Y$ and $Z$,

\[
\{\forall Y \cup Z\} = \{\forall Y\}; \{\forall Z\},
\]

\[
[\forall Y \cup Z] = [\forall Y]; [\forall Z].
\]

Then, the monotonicity properties expressed by the following lemma follow directly from $\text{skip} \leq \{\forall Y\}$ and $[\forall Y] \leq \text{skip}$.

**Lemma 2.5** For sets of variables $Y$ and $Z$, such that $Y \subseteq Z$,

\[
\{\forall Y\} \leq \{\forall Z\} \text{ and } [\forall Z] \leq [\forall Y].
\]

## 2.2 The UNITY Framework

In this section a brief overview of UNITY is given. First, we describe the UNITY programming language and then we give two programming logics: the standard UNITY logic of Chandy and Misra [CM88], and a modified logic introduced by Sanders [San91b].

The UNITY framework, introduced by Chandy and Misra [CM88], is a formal framework for the design and development of parallel and distributed programs. UNITY consists of a programming language and a programming logic. The programming language
is restricted: a UNITY program is made up of a set of statements that are repeatedly executed in a nondeterministic order. A UNITY program is an abstract representation of a distributed computation: it only specifies what should be done, it does not specify when, where or how it should be done. Therefore, a program can be mapped onto different parallel or distributed architectures. The simplicity of the language also makes reasoning about execution of these programs feasible. A similar way of modelling distributed programs has been propagated by Back and Kurki-Suonio in their action system formalism [BKS83, Bac90], by Lamport in his Temporal Logic of Actions [Lam91], by Brunekreef in Process Algebra [Bru94], and by Banâtre and Le Métayer in the Gamma framework [BM90].

For reasoning about programs, UNITY uses a programming logic based on a small set of temporal properties. These properties are also used to give specifications. UNITY supports the design of programs by stepwise refinement of specifications.

After its introduction, the UNITY framework has been an object of several studies. Sanders gave a modification of the UNITY logic to reason about programs as closed systems [San91b, San91a], and clarified the confusion around the UNITY substitution axiom. A similar modification of the logic has been given by Knapp [Kna94]. Both Julta and Rao [JR92], and Dijkstra [Dij94] described the UNITY logic in a setting of predicate transformers. Julta and Rao show that this gives generalisation of the logic to different notions of fairness [JR92, Rao95]. Together with Knapp, they show that it also gives a simple relation between properties in the UNITY framework and properties in temporal logic [JKR89, Kna89]. This relation is also studied by Gerth and Pnueli [GP89] and Knapp [Kna94]. Collette also extends the logic with rely-guarantee properties to give a complete compositional proof system [Col94b, Col94a].

The UNITY programming language has been designed such that programs can be easily mapped onto different architectures, but this step is not always trivial. Moreover, this step is not a formal design step and therefore a point in the program development that requires some extra attention. To simplify mappings and reduce informal reasoning, Sanders [San90] and Singh [Sin93] proposed notions of program refinement in UNITY.

UNITY has been successfully applied to a number of programming problems. Chandy and Misra give many examples in their book [CM88] and in some of their articles [Mis89, Mis90b]. Knapp used UNITY to design and verify an efficient algorithm for finding maximum flows in graphs [Kna90] and for deriving a parallel linear search and an asynchronous fixed point computation [Kna88]. Staskauskas used the UNITY framework to solve some industrial programming problems. By designing an electronic funds-transfer system [Sta88] and an I/O subsystem portion of an existing operating system [Sta93] he showed that UNITY can be helpful in the design of relatively complex distributed programs. Charkravarty et al. show how a parallel simulation program of a physical process can be developed within the UNITY framework [CKW+91]. They
derive a machine independent and efficient UNITY program and show how it can be implemented on different architectures without losing efficiency. Also routing problems have been treated in the UNITY framework. Creveuil and Roman derive a message router \[\text{CR94}\], and Lentfert and Swierstra use UNITY to derive distributed hierarchical routing algorithms \[\text{LS93, Len93}\].

Several tools for mechanical verification of proofs in UNITY have been developed. Goldschlag implemented UNITY in the Boyer-Moore theorem prover \[\text{Gol90a, Gol90b}\]. Implementations of UNITY in HOL have been developed by Andersen \[\text{And92}\] and Prasetya \[\text{Pra95}\]. Prasetya also implemented an extra set of properties to support the verification of self-stabilising algorithms. In his thesis \[\text{Pra95}\], he uses HOL to verify a proof of the hierarchical routing algorithms of Lentfert and Swierstra.

Next, we take a closer look at the UNITY framework. The framework consists of a programming language and a programming logic.

Programs in the UNITY framework, denoted by \(F, G\) and \(H\), consist of several parts that are called sections. We only consider a subset of UNITY programs, namely, those that are made up of the following sections:

- The \textit{initially-section}, containing a predicate that characterises the set of possible initial states. This predicate is denoted by \(\text{init}(F)\) and is sometime interpreted as a set. In the sequel, we assume that \(\text{init}(F) \neq \text{false}\), i.e. there exists at least one state that satisfies \(\text{init}(F)\). If \(\text{init}(F) = \text{true}\), then the section is omitted.

- The \textit{assign-section}, containing a finite, non-empty set of statements on the UNITY state space. We interpret these statements as weakest precondition predicate transformers and use the language of the refinement calculus, as given in the previous section. Statements in the \textit{assign-section} must be deterministic. The set of statements is denoted by \(\text{assign}(F)\) and statements are separated by the symbol \(|\).

UNITY programs have the following operational interpretation. Execution of a UNITY program starts in an arbitrary state satisfying the \textit{initially-section} of the program. At each execution step a statement of the \textit{assign-section} is chosen and executed. Furthermore, an execution is \textit{fair}, that is, each statement is chosen infinitely often. Statements in a UNITY program are always enabled; if the condition of a guarded statement evaluates to false, execution of that statement is equivalent to that of \textit{skip}. 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 the program, one can view this state as the result of the computation. A sequence of states that can occur during execution is called an execution sequence of the program.

For example, the execution of program \(F\) as given in example 2.1 starts in a state in which the value of \(x\) is non-negative. Then, repeatedly one of its statements is executed.
Example 2.1.

Execution of the first statement of $F$ decreases $x$, which will happen as long as $x$ is positive. If $x$ is non-positive, execution of this statement results in a $skip$. Execution of the second statement of $F$ increases $x$ if $x$ is negative and, since $x$ will never be negative, always results in $skip$. So, execution of program $F$ decreases the value of $x$ until the fixed point $x = 0$ is reached.

In the UNITY framework, parallel composition of programs is modelled by the union operator $\parallel$.

**Definition 2.6** For UNITY programs $F$ and $G$, the union $F \parallel G$ is defined by

\[
\begin{align*}
\text{init}(F \parallel G) & \quad \overset{\text{def}}{=} \quad (\text{init}(F) \land \text{init}(G)), \\
\text{assign}(F \parallel G) & \quad \overset{\text{def}}{=} \quad (\text{assign}(F) \cup \text{assign}(G)).
\end{align*}
\]

The union operator is associative, symmetric and idempotent. If a program $F$ is composed with program $G$ by the union operator, then $F$ is called an environment of $G$ and $G$ is an environment of $F$. Program union models parallelism by interleaving the actions of both programs. For example, program $F$ of example 2.1 can be put in parallel with program $G$ given in example 2.2. This results in program $F \parallel G$ as given in example 2.2. Note that we can only compose two programs with the union operator if the initially-sections of both programs are consistent, since we only allow programs with non-empty initially-sections.

---

Example 2.2.
Now, we give two logics for UNITY programs. We start with the standard UNITY logic given by Chandy and Misra [CM88]. This logic is based on three properties: unless, ensures and $\rightarrow$ (leadsto). These properties are attached to an entire program and are defined in terms of the set of statements. Because we introduce more logics later, we subscript the properties by $CM$.

**Definition 2.7 (Chandy and Misra's Logic)** For UNITY program $F$, properties of $F$ are defined by

1. **unless property:**
   \[ p \text{ unless}_{CM} q \overset{\text{def}}{=} (\forall S : S \in \text{assign}(F) : [(p \land \neg q) \Rightarrow S(p \lor q)]). \]

2. **ensures property:**
   \[ p \text{ ensures}_{CM} q \overset{\text{def}}{=} p \text{ unless}_{CM} q \land (\exists S : S \in \text{assign}(F) : [(p \land \neg q) \Rightarrow S(q)]). \]

3. **leadsto property:** $\rightarrow_{CM}$ is defined as the smallest binary relation $Prop$ between predicates satisfying the following conditions:
   
   (a) If $p \text{ ensures}_{CM} q$, then $p \text{ Prop } q$.
   
   (b) If $p \text{ Prop } r$ and $r \text{ Prop } q$, then $p \text{ Prop } q$.
   
   (c) For any set $W$, if $(\forall w : w \in W : p_w \text{ Prop } q)$, then $(\exists w : w \in W : p_w) \text{ Prop } q$.

When it is not clear from the context to which program properties refer, the program is mentioned explicitly, for example $p \text{ unless}_{CM} q$ in $F$. In [CM88], the UNITY properties are extensively studied and a lot of theorems about these properties are derived. We give an overview of these theorems in appendix A. Using the basic properties other properties can be defined,

\[ \text{stable}_{CM} p \overset{\text{def}}{=} p \text{ unless}_{CM} \text{ false}, \]
\[ \text{invariant}_{CM} p \overset{\text{def}}{=} [\text{init}(F) \Rightarrow p] \land \text{stable}_{CM} p, \]
\[ p \text{ until}_{CM} q \overset{\text{def}}{=} p \text{ unless}_{CM} q \land p \rightarrow_{CM} q. \]

Like the $\rightarrow$ property, property until can also be defined directly as a closure of ensures properties.

**Theorem 2.8** Property until$_{CM}$ is the smallest relation $Prop$ that satisfies the following conditions:

1. If $p \text{ ensures}_{CM} q$, then $p \text{ Prop } q$. 

2. If \( p \text{ Prop} (r \lor q) \) and \( r \text{ Prop} q \), then \( (p \lor r) \text{ Prop} q \).

3. If for any set \( W \) \( \forall w : w \in W : p_w \text{ Prop} q \), then \( \exists w : w \in W : p_w \text{ Prop} q \).

**Proof:** First, the implication \( p \text{ Prop} q \Rightarrow p \text{ until}_CM q \) is proved by induction on the proof of \( p \text{ Prop} q \), using that all inference steps of \( \text{Prop} \) also hold for the \( \text{until}_CM \) property. Second, by induction we can prove that every \( \Rightarrow_{CM} \) implies a \( \text{Prop} \) property:

\[
p \Rightarrow_{CM} q \Rightarrow (\exists r :: (p \lor r) \text{ Prop} q).
\]

By induction we can prove that the conjunction rule holds for \( \text{Prop} \). Then,

\[
p \Rightarrow_{CM} q
\]

\[
= \{\text{definition}\}
\]

\[
p \Rightarrow_{CM} q \land p \text{ unless}_CM q
\]

\[
\Rightarrow \{\text{above}\}
\]

\[
(\exists r :: (p \lor r) \text{ Prop} q) \land p \text{ unless}_CM q
\]

\[
\Rightarrow \{\text{conjunction}\}
\]

\[
(\exists r :: p \text{ Prop} q)
\]

\[
= \{\text{predicate calculus}\}
\]

\[
p \text{ Prop} q
\]

\[\square\]

The following theorem shows that some properties are *compositional* in the sense that properties of a union can be split into properties of its components.

**Theorem 2.9 (Union)** For UNITY programs \( F \) and \( G \), and predicates \( p \) and \( q \),

\[
p \text{ unless}_CM q \text{ in } F | G \]

\[
= p \text{ unless}_CM q \text{ in } F \land p \text{ unless}_CM q \text{ in } G,
\]

\[
p \text{ ensures}_CM q \text{ in } F | G \]

\[
= p \text{ ensures}_CM q \text{ in } F \land p \text{ unless}_CM q \text{ in } G \land p \text{ unless}_CM q \text{ in } G \lor p \text{ ensures}_CM q \text{ in } G \land p \text{ unless}_CM q \text{ in } F.
\]

Besides the definition of the properties, Chandy and Misra postulate the substitution axiom, an inference rule for deriving properties of closed systems. The *substitution axiom* states that if \( \text{invariant}_CM a = b \) holds, then \( a \) may be substituted for \( b \) in every property of the program. This axiom has caused a lot of confusion, especially when used in combination with the union theorem. Therefore, we do not consider the substitution axiom to be part of the UNITY logic.

Sanders clarified the status of the substitution axiom by modifying the UNITY logic \([San91b]\). She redefined the UNITY properties in such a way that the substitution axiom can be derived as a theorem. The following definition gives (a characterisation of) the properties of the logic of Sanders. We subscript these properties by \( S \).
Definition 2.10 (Sanders’s Logic) For UNITY program $F$ and predicates $p$ and $q$, properties of $F$ are defined by

$$p \text{ unless}_S q \overset{\text{def}}{=} \langle \exists r : \text{invariant}_CM r : (r \land p) \text{ unless}_CM q \rangle,$$
$$p \text{ ensures}_S q \overset{\text{def}}{=} \langle \exists r : \text{invariant}_CM r : (r \land p) \text{ ensures}_CM q \rangle,$$
$$p \mapsto_S q \overset{\text{def}}{=} \langle \exists r : \text{invariant}_CM r : (r \land p) \mapsto CM q \rangle.$$

The properties $\text{stable}_S$, $\text{invariant}_S$ and $\text{until}_S$ can be defined in a similar way as before, but now using Sanders’s properties. Properties $\mapsto_S$ and $\text{until}_S$ can also be defined as closures of the $\text{ensures}_S$ property as in definition 2.7 and theorem 2.8.

Definition 2.10 restricts properties to reachable states since the strongest invariant of a program corresponds to the set of its reachable states. This has four important consequences. First, the substitution principle holds (and can be derived as a theorem). Second, as shown by Pachl [Pac92] and formally stated in lemma 3.18, for Sanders’s logic the interpretation of the properties can be given in terms of execution sequences:

1. $p \text{ unless}_S q$ in $F$ holds if and only if for every state in every execution sequence of $F$, if $p \land \neg q$ holds, then $p \lor q$ holds in its successor state.

2. $p \mapsto_S q$ in $F$ holds if and only if for every state in every execution sequence of $F$, if $p$ holds, $q$ will hold later on.

This relation between UNITY properties, which are properties of programs, and temporal properties, which are properties of a program execution, makes the Sanders’s logic useful for specifying systems. Third, Sanders’s properties are weaker than Chandy and Misra’s properties. Last, the logic is not suitable for compositional reasoning for two different reasons. First, property $\mapsto_S$ is not compositional (neither are $\mapsto CM$, $\text{until}_{CM}$, and $\text{until}_S$), which is illustrated by program $F$ and $G$ as given in examples 2.1 and 2.2. Although both $|x| = 1 \mapsto_S x = 0$ in $F$ and $|x| = 1 \text{ unless}_S x = 0$ in $G$ hold, the property $|x| = 1 \mapsto_S x = 0$ does not hold in $F[G]$. Program $G$ can change the sign of $x$ in such a way that each statement of $F$ is only executed when the guard is false. Second, compositionality is lost by restricting the properties to reachable states. This is also illustrated by programs $F$ and $G$. Since $\text{invariant}_{CM} x \geq 0$ is a property of $F$, $|x| = 1 \text{ ensures}_S x = 0$ in $F$ holds. Although $|x| = 1 \text{ unless}_{CM} x = 0$ is a property of $G$, property $|x| = 1 \text{ ensures}_S x = 0$ does not hold in the composition $F[G]$ since $\text{invariant}_{CM} x \geq 0$ does not hold in $F[G]$. 
Chapter 3
Program Refinement in UNITY

In this chapter we focus on the semantics of UNITY programs and program refinement. We define six different semantic models for UNITY programs. Program refinement corresponds to the preservation of semantic properties, so the different models yield different notions of program refinement. In this study, two aspects play an important role: compositionality and the way programs are observed.

Programs can be observed in different ways. For UNITY programs we consider two observation criteria. The first is an operational view based on the idea that during execution of a program only the state of the program can be observed. This is a natural criterion used, for example, in temporal logic [MP92], TLA [Lam91] and action systems [BvW94]. It is modelled by a semantics $O$ that gives the set of all stutter-free sequences of states that can occur during program execution. Then, program refinement corresponds to set-inclusion: program $F$ is refined by program $G$ if and only if every execution sequence of $G$ is an execution sequence of $F$. The second observation criterion is about the UNITY properties satisfied by a program. Refinement preserves the set of UNITY properties that the program satisfies, in other words, a program $F$ is refined by a program $G$ if and only if every property of $F$ is also a property of $G$. A specification in the UNITY framework is a set of UNITY properties that must be satisfied by the implementation. So, a program $F$ is refined by a program $G$ if and only if $G$ satisfies every specification that is satisfied by $F$. We examine four different models based on different kinds of UNITY properties.

Compositionality is about the behaviour of programs operating in some environment. For UNITY programs, environments are formed using the union operator $|$. Now, we define what we mean by a compositional model and a best compositional model.

**Definition 3.1** A model $M_1$ is compositional with respect to a model $M_2$ if for all UNITY programs $F, G$ and $H$,

\[
(M_1[F] = M_1[G]) \Rightarrow (M_2[F|H] = M_2[G|H]).
\]
A model $M_1$ is the best compositional model with respect to a model $M_2$ if for all UNITY programs $F$ and $G$,

$$\left(M_1[F] = M_1[G]\right) = \left(\forall H : M_2[F][H] = M_2[G][H]\right).$$

A model that is compositional with respect to itself is called compositional. A best compositional model is also called a fully abstract model.

![Semantic models for UNITY](image)

Figure 3.1. Semantic models for UNITY

For future reference, figure 3.1 gives an overview of the hierarchy of the models that are introduced in this chapter. On the left-hand side, two operational models based on sequences are given. Model $O$ is based on execution sequences, and model $C$ is a compositional model based on so-called extended sequences. On the right-hand side, four property based models are given. These models are based on the initially-section of the program and use unless properties for modelling the safety properties. For modelling progress, models $IUE_{CM}$ and $IUE_{S}$ use ensures properties, while models $IUL_{CM}$ and $IUL_{S}$ are based on leadsto properties. The subscripts $CM$ and $S$ indicate whether a model is based on properties as defined by Chandy and Misra (definition 2.7) or by Sanders (definition 2.10). Models $C$ and $IUE_{CM}$ are compositional models. Arrows in the figure indicate abstractions, i.e. an arrow from model $M_1$ to model $M_2$ states that $(M_1[F] = M_1[G]) \Rightarrow (M_2[F] = M_2[G])$. A thick arrow from model $M_1$ to model $M_2$ states that $M_1$ is the best compositional model with respect to $M_2$.

We want to examine program refinement and we use the semantic models to define notions of refinement for UNITY programs. For each model $M$ we define a refinement relation $\sqsubseteq_M$. This is done in such a way that compositional models give compositional notions of refinement and compositionality of a refinement relation is defined as follows.

**Definition 3.2** A refinement relation $\sqsubseteq_1$ is compositional with respect to refinement relation $\sqsubseteq_2$ if for all UNITY programs $F$, $G$, and $H$,

$$(F \sqsubseteq_1 G) \Rightarrow (F[H] \sqsubseteq_2 G[H]).$$
A refinement relation $\sqsubseteq_1$ is the best compositional refinement relation with respect to refinement relation $\sqsubseteq_2$ if for all UNITY programs $F$ and $G$,

$$(F \sqsubseteq_1 G) = (\forall H :: F[H] \sqsubseteq_2 G[H]).$$

A refinement relation that is compositional with respect to itself is called compositional. The arrows in figure 3.1 also indicate abstraction on the level of refinement, i.e. an arrow from model $M_1$ to model $M_2$ implies that $(F \sqsubseteq_{M_1} G) \Rightarrow (F \sqsubseteq_{M_2} G)$. The thick arrows indicate best compositional notions of refinement. We show that $\sqsubseteq_C$ is the best compositional notion of refinement with respect to $\sqsubseteq_O$, and that $\sqsubseteq_{\text{IUUCM}}$ is the best compositional notion of refinement with respect to the refinement in all property based models.

In contrast to the situation in, for example, many process algebras, where modelling deadlock is the main source of difficulty, for UNITY the main problem is to deal with fairness. Looking for best compositional models we encounter the following problems:

1. finding the balance between abstraction from stutterings versus the need to observe whether fairness requirements are satisfied.

2. the restrictive means of building environments; UNITY programs have only a finite set of deterministic statements.

It is interesting to observe that the following combination of design choices in UNITY allows for (relatively) simple best compositional models:

1. every statement is always enabled: if the guard of a statement is false, then it is still executed and is equivalent to a skip statement,

2. every statement is executed an infinite number of times, and

3. every statement is deterministic.

This chapter is organised as follows. In section 3.1, the left side of figure 3.1, containing the operational models, is studied. The semantic models $O$ and $C$, based on sequences, are introduced and motivated. Both models are used to define a notion of program refinement and it is proved that refinement based on $C$ is the best compositional notion of refinement with respect to $O$. Section 3.2 deals with the property based models. These models give notions of refinement and it is proved that refinement based on $\text{IUUCM}$ is the best compositional notion of refinement with respect to the other property based models. In section 3.3, the two categories of models are compared and are shown to be different.
3.1 Operational Semantics

Chandy and Misra [CM88] define an execution model for UNITY programs based on sequences of tuples. Each tuple consists of a state and (a label of) a statement that is executed in that state. Each element of a sequence models an execution step of the program and a sequence denotes a possible run. In this section we define a more abstract operational semantics for UNITY based on so-called execution sequences. We assume that during the execution of a program only the state of the program is observable. We abstract from (labels of) statements that are executed and from stutterings, execution steps that do not change the state.

For dealing with execution sequences, we introduce some notation. Infinite sequences of states are modelled as functions from natural numbers to states, \((z \in \text{Seq}) \triangleq \mathbb{N} \rightarrow \Sigma\). We use \((\langle \cdot \rangle)\)-brackets for denoting sequences. To abstract from stutterings, we use the operator \(\downarrow : \text{Seq} \rightarrow \text{Seq}\) (see [AL88]) that replaces all maximal finite segments of identical states by the single state. A sequence \(z\) is stutter-free, denoted by \(sf(z)\), if \(z(z) = z\). A stutter-free sequence may contain stutterings but only as an infinite suffix. So,

\[
sf(z) \triangleq \langle \forall i : z(i) = z(i + 1) : \langle \forall j : j \geq i : z(i) = z(j)\rangle\rangle.
\]

By \(\sigma \xrightarrow{S} \rho\) we denote that execution of statement \(S\) in state \(\sigma\) may result in state \(\rho\). This can be defined in terms of predicate transformers by \(\sigma \xrightarrow{S} \rho \triangleq [p_\sigma \Rightarrow S^c(p_\rho)]\), where \(p_\sigma\) and \(p_\rho\) are point predicates that only hold in state \(\sigma\) respectively state \(\rho\). Since statements in a UNITY program are deterministic \((S = S^c)\), this corresponds to \(\sigma \xrightarrow{S} \rho \triangleq [p_\sigma \Rightarrow S(p_\rho)]\).

Now, an execution sequence of a program is a stutter-free sequence of states that may occur during execution of the program. The operational semantics of a UNITY program is defined as the set of all execution sequences.

**Definition 3.3** The semantic model \(O : \text{UNITY} \rightarrow \mathcal{P}(\text{Seq})\) yields for UNITY program \(F\) the set of sequences \(O[F]\) such that a sequence \(z\) is an element of \(O[F]\) if and only if

1. \(z(0) \in \text{init}(F)\);
2. \(z\) is stutter-free: \(sf(z)\);
3. all state changes in \(z\) can be done by a statement of the program \(F\):
   \[
   \langle \forall i : (\exists S : S \in \text{assign}(F) : z(i) \xrightarrow{S} z(i + 1))\rangle;
   \]
4. every statement of \(F\) can be executed infinitely often in \(z\):
   \[
   \langle \forall S, i : S \in \text{assign}(F) : (\exists j : j > i : z(j) \xrightarrow{S} z(j) \lor z(j) \xrightarrow{S} z(j + 1))\rangle.
   \]
Note that we do not require that every statement is executed infinitely often. We require that the effect of each statement occurs infinitely often (as a state transition or as a stuttering). Since UNITY programs only have a finite number of statements, these notions of fairness are equivalent, so our fairness requirement is equivalent to UNITY fairness. In UNITY there is no notion of enabledness of a statement; a statement whose guard is *false* behaves as a stuttering, i.e. as *skip*. Since we abstract from stutterings, our notion of fairness is equivalent to weak fairness [Fra86, MP92]: which says that every statement whose guard is *true* continuously is eventually executed.

Model $O$ gives the possible behaviours of a program when it executes in isolation. So, it does not say anything about the behaviour of the program in states that are not reachable by the program in isolation, but that may be reachable by the program in some environment. Therefore, the model is not compositional.

Now, we construct a semantic model that is compositional. It is inspired by the compositional model in [BKPR91], but we need different closure conditions. The first step towards the model is to allow for interleaving. This is done by introducing extended sequences. *Extended sequences* are infinite sequences of pairs of states, i.e. $(v \in) ES\text{eq} \overset{\text{def}}{=} \mathbb{N} \to \Sigma \times \Sigma$, and with each extended sequence $v$ we associate two sequences $v.1, v.2 \in Seq$ that project on the first and the second elements of the pairs of states. The intuition is that each element in an extended sequence denotes an execution step of the program when it operates in an environment. Each pair of states $(\sigma_1, \sigma_2)$ of an extended sequence models the execution of a statement of the program starting in state $\sigma_1$ and resulting in state $\sigma_2$. In the *hole* between two successive pairs, the environment may change the states. Two states $(\sigma_1, \sigma_2)$ and $(\rho_1, \rho_2)$ are called *connected* if $\sigma_2 = \rho_1$. If two successive pairs are not connected, the hole between these pairs is called a *gap* and an environment must bridge the gap. *Stutterings* in extended sequences are pairs of identical states, and for reasons we explain later, we abstract from connected stutterings, i.e. stutterings in an extended sequence that are connected with one of their neighbours. We examine so-called connected-stutter-free sequences, i.e. extended sequences $v$ for which $csf(v)$ holds, where

$$csf(v) \overset{\text{def}}{=} \langle \forall i : \langle \exists \sigma : v(i) = (\sigma, \sigma) \land (v.2(i - 1) = \sigma \lor v.1(i + 1) = \sigma) \rangle \rangle : \langle \forall j : j \geq i : v(i) = v(j) \rangle \rangle.$$

A connected-stutter-free sequence may contain connected stutterings, but only as an infinite suffix. The operator $\sharp$ is lifted to extended sequences as the operator that removes connected stutterings, i.e. a maximal finite segments of stuttering $v(k), \ldots, v(l)$ is replaces by the single stuttering $v(k)$ if the segment is not connected at one of its ends ($(v.2(k - 1) \neq v.1(k)) \land (v.2(l) \neq v.1(l + 1)))$, otherwise it is removed. For example,

$$\sharp(\langle((1,2),(2,2),(0,0),(0,0),(1,1),(1,1),(1,2),\ldots)\rangle) = \langle((1,2),(0,0),(1,2),\ldots)\rangle.$$
Then, sequence \( v \) is connected-stutter-free if \( v = \#(v) \).

Extended sequences are used to define a compositional model for UNITY programs in the following way. We first give the definition of the compositional semantics and motivate this definition afterwards.

**Definition 3.4** The semantic model \( C : \text{UNITY} \rightarrow (\mathcal{P}(\Sigma) \times \mathcal{P}(\text{ESeq})) \) yields for UNITY program \( F \) the pair \( C[F] = (\text{init}(F), V) \), where \( V \) is the set of extended sequences such that an extended sequence \( v \) is an element of \( V \) if and only if

1. \( v \) contains no connected stutterings: \( \text{csf}(v) \);

2. every pair in \( v \) is either a stuttering or is the execution of a statement of \( F \):

\[
(\forall i : v.1(i) \neq v.2(i) : (\exists S : S \in \text{assign}(F) : v.1(i) \xrightarrow{S} v.2(i)))\]

3. every statement of \( F \) is executed fairly in \( v \):

\[
(\forall S, i : S \in \text{assign}(F) \\
\quad : (\exists j : j > i : \text{step}(S, v(j)) \lor \text{inbetween}(S, v(j - 1), v(j))))\]

where

\[
\text{step}(S, (\sigma_1, \sigma_2)) \triangleq \sigma_1 \xrightarrow{S} \sigma_1 \lor \sigma_1 \xrightarrow{S} \sigma_2 \lor \sigma_2 \xrightarrow{S} \sigma_2,
\]

and

\[
\text{inbetween}(S, (\sigma_1, \sigma_2), (\sigma_3, \sigma_4)) \triangleq \sigma_2 \neq \sigma_3 \land (\forall \rho : \rho \neq \sigma_3 : \text{step}(S, (\rho, \sigma_3))).
\]

An extended sequence models the contribution of a program when it executes in an environment. The environment is supposed to fill the gaps. From extended sequences, normal execution sequences can be derived by considering connected sequences, i.e. sequences in which all pairs are connected. Connected sequences do not contain gaps that need to be filled by an environment. Execution sequences can be derived by checking the initial state and removing redundant information.

**Theorem 3.5 (Correctness)** For a UNITY program \( F \) with \( C[F] = (I, V) \),

\[
\mathcal{O}[F] = \{ v.1 \mid (v \in V) \land (v.1(0) \in I) \land (\forall i : v.2(i) = v.1(i + 1)) \}.
\]
Proof: We only give an outline of the proof. First, we show that for a connected extended sequence \( v \in V \) such that \( v.1(0) \in I \), sequence \( v.1 \in O[F] \). The fact that \( v \) is connected has three consequences. First, \( v \) is connected-stutter-free, so \( v.1 \) is stutter-free. Second, since any pair in \( v \) is a step of a statement in \( F \), each transition in \( v.1 \) is a transition of \( F \). Third, the inbetween part of the fairness requirement in definition 3.4 never holds, so the effect of each statement occurs infinitely often as a step in \( v \) and as a transition or stuttering in \( v.1 \). Consequently, \( v.1 \in O[F] \).

The other direction, for every sequence \( z \in O[F] \) we can construct the extended sequence \( v = \langle (z(0), z(1)), (z(1), z(2)), \cdots \rangle \) such that \( z = v.1 \), \( v \) connected and connected-stutter-free, and \( v.1(0) \in I \) and \( v \in V \).

---

**Program** \( F \)

*init* \( x \in \{0, 1, 2\} \)

*assign* \( x := -x \mod 3 \)

\[
\begin{array}{l}
| x := 0 \\
\end{array}
\]

*end\{F\}*

---

**Example 3.2.**

The compositional model abstracts from connected stutterings, but not from all stutterings, as the operational model does. If all stutterings in an extended sequence were removed, then theorem 3.5 would not hold. This is shown by program \( F \) in example 3.2. The extended sequence \( \langle (1, 2), (2, 1), (0, 0), (1, 2), (2, 1), (0, 0), \cdots \rangle \) (identifying states by the value of \( x \)) models a fair execution of program \( F \). The stutterings \((0, 0)\) correspond to executions of statement \( x := 0 \). Removing all stutterings results in a connected sequence, which corresponds to the execution sequence \( \langle 1, 2, 1, 2, 1, 2, \cdots \rangle \). However, this sequence is not an execution sequence of \( F \) because it does not satisfy the fairness criterion that statement \( x := 0 \) is executed infinitely often.

Although the removal of connected stutterings makes the model more abstract, it is not sufficient to obtain a best composition model. This can be seen from programs \( F \) and \( G \) in example 3.3. Every pair in the extended sequence \( \langle (0, 1), (2, 2), (3, 4), \cdots \rangle \) corresponds to the execution of a statement of program \( F \). This is not the case for \( G \) since no statement of \( G \) causes the stuttering \((2, 2)\). However, both programs have the same operational semantics and there is no environment that can detect this difference, i.e. \( O[F \mid H] = O[G \mid H] \) for all programs \( H \). By adding arbitrary stuttering steps, our model is made more abstract. This is done by the second item in definition 3.4 which allows for stuttering.

The last item of definition 3.4 concerns fairness. It expresses that the effect of each
Program $F$

assign $x := x + 1$

$| x := x$

end\{$F$\}

Program $G$

assign $x := x + 1$

end\{$G$\}

Example 3.3.

statement occurs infinitely often. The term $\text{step}(\cdots)$ expresses that the effect of statement $S$ occurs as a state transition or as a stuttering (due to the fact that we removed connected stutterings, we allow possible stutterings in other pairs). The term $\text{inbetween}(\cdots)$ states that the effect of the statement becomes visible when the gap between two successive pairs of states is filled. The need for the second conjunct is shown by the programs $F$ and $G$ in example 3.4. These programs are given together with their labelled state transition diagrams. It is straightforward to verify that $\mathcal{O}[F|H] = \mathcal{O}[G|H]$ holds for each UNITY program $H$. If we omit the $\text{inbetween}(\cdots)$ part, the extended sequence $\llangle(1,2),(1,2),\cdots\rrangle$ is a sequence of $F$ and not of $G$.

Example 3.4.

Model $C$ is compositional and the semantic equivalent of program union $\mid$ is defined
as follows. We use the standard fair interleaving (denoted in the sequel by \textit{merge}) on extended sequences.

\textbf{Definition 3.6} Let \( F \) and \( G \) be UNITY programs, such that \( C[F] = (I_F, V_F) \) and \( C[G] = (I_G, V_G) \). Then,

\[
C[F] \parallel C[G] \overset{\text{def}}{=} (I_F \cap I_G, \{z(v) \mid \exists v_F, v_G : v_F \in V_F \land v_G \in V_G : v \in \text{merge}(v_F, v_G)\}) \text{.}
\]

Compositionality of \( C \) is expressed by the following theorem. The proof of this theorem is straightforward.

\textbf{Theorem 3.7 (Compositionality)} For UNITY programs \( F \) and \( G \),

\[
C[F \parallel G] = C[F] \parallel C[G] \text{.}
\]

\textbf{Proof:} We only give an outline of the proof. Let \( C[F] = (I_F, V_F) \), \( C[G] = (I_G, V_G) \), and \( C[F \parallel G] = (I_{FG}, V_{FG}) \). The correspondence between the initial parts is easy to verify, here we focus on the correspondence between the sets of extended sequences. First we show that for extended sequences \( v_F \in V_F, v_G \in V_G \) and \( v \in \text{merge}(v_F, v_G) \), sequence \( z(v) \) is an element of \( V_{FG} \). It is straightforward to see that all transitions in \( v \) are stutterings or can be done by a statement in \( F \parallel G \), so the same holds for the transitions in \( z(v) \). Now, we show that \( v \) satisfies the fairness requirement of \( F \parallel G \). Let \( S \in \text{assign}(F \parallel G) \) be a statement from \( F \) (for statements from \( G \), the situation is similar).

If the effect of \( S \) occurs as a \textit{step} in \( v_F \), then it also occurs as a step in \( v \). Suppose, the effect of \( S \) occurs in \( v_F \) as \textit{inbetween}(\( S, v_F(i-1), v_F(i) \)) for a certain \( i \). If no pairs are merged between \( v_F(i-1) \) and \( v_F(i) \), the \textit{inbetween} holds for the corresponding pairs in \( v \). Otherwise a segment \( v_G(k), \ldots, v_G(m) \) is merged between \( v_F(i-1) \) and \( v_F(i) \), and let \( l \) be the least index such that \( \forall j : l < j \leq m : v_G(j) = (v_F.1(i), v_F.1(i)) \). Then, we have one of the following situations:

- If \( l < k \), then the effect of \( S \) occurs as \textit{inbetween}(\( S, v_F(i-1), v_G(k) \)) in \( v \).

- If \( v_G.2(l) \neq v_F.1(i) \), then the effect of \( S \) occurs in \( v \) as \textit{inbetween}(\( S, v_G(l), v_F(i) \)) if \( l = m \), or as \textit{inbetween}(\( S, v_G(l), v_G(l+1) \)) if \( l < m \).

- If \( v_G(l) \) is a non-stuttering \( (\rho, v_F.1(i)) \), then the effect of \( S \) occurs as \textit{step}(\( S, v_G(l) \)) for the corresponding pair in \( v \).

So, \( v \), and therefore also \( z(v) \), are fair in \( F \parallel G \) and \( z(v) \in V_{FG} \).

For the other direction, an extended sequence \( v \in V_{FG} \) can be “divided” into sequences \( v_F \) and \( v_G \) such that \( z(v_F) \in V_F \), \( z(v_G) \in V_G \) and \( z(v') = v \) for a sequence \( v' \in \text{merge}(z(v_F), z(v_G)) \). Pairs of \( v \) are assigned to one of the sequences \( v_F, v_G \) as follows.
If a pair \( v(i) = (\sigma, \rho) \) is a transition of \( F \) and not of \( G \), then the pair is put in \( v_F \). If \( v(i) \) is important for the fairness of a statement \( S \) from \( G \), then a stuttering is added to \( v_G \), i.e. the stuttering \((\sigma, \sigma)\) if \( \sigma \xrightarrow{S} \sigma \) or \( \text{inbetween}(S, v(i - 1), v(i)) \), and the stuttering \((\rho, \rho)\) if \( \rho \xrightarrow{S} \rho \). These stutterings will be connected stutterings in the merged sequence and can be removed. For pairs with transitions from \( G \) and not from \( F \), the situation is similar. Pairs that are transitions of statements from both programs have to be divided equally over \( v_F \) and \( v_G \), which can be done since the number of statements in a program is finite. In this way, sequences \( v_F \) and \( v_G \), and also \( \mathcal{t}(v_F) \) and \( \mathcal{t}(v_G) \), are (fair) extended sequences of \( F \) and \( G \) respectively.

3.1.1 Refinement

A general notion of refinement is the reduction of nondeterminism. In terms of execution sequences, this corresponds to a reduction of the number of possible execution sequences [Lam91].

**Definition 3.8** For UNITY programs \( F \) and \( G \), refinement of \( F \) by \( G \) in \( \mathcal{O} \) is defined by

\[
F \sqsubseteq_{\mathcal{O}} G \overset{\text{def}}{=} \mathcal{O}[F] \supseteq \mathcal{O}[G].
\]

Model \( \mathcal{O} \) is not compositional; neither is the notion of refinement in \( \mathcal{O} \). Refinement in \( \mathcal{C} \) corresponds to reduction of the set of extended sequences and reduction of the set of initial states.

**Definition 3.9** For UNITY programs \( F \) and \( G \), refinement of \( F \) by \( G \) in \( \mathcal{C} \) is defined by

\[
F \sqsubseteq_{\mathcal{C}} G \overset{\text{def}}{=} I_F \supseteq I_G \land V_F \supseteq V_G,
\]

where \( \mathcal{C}[F] = (I_F, V_F) \) and \( \mathcal{C}[G] = (I_G, V_G) \).

As is shown in the next section, this is the best compositional notion of refinement with respect to refinement in \( \mathcal{O} \).

3.1.2 Best Compositional Refinement

Theorem 3.7 shows that \( \mathcal{C} \) is compositional, and theorem 3.5 states that \( \mathcal{C} \) is an abstraction of \( \mathcal{O} \). A similar relation holds for the the notions of refinement in the models. We show that refinement in \( \mathcal{C} \) is the best notion of compositional notion of refinement in \( \mathcal{O} \).

**Theorem 3.10** The refinement notion \( \sqsubseteq_{\mathcal{C}} \) is the best compositional notion of refinement with respect to the refinement relation \( \sqsubseteq_{\mathcal{O}} \).
**Proof:** We have to prove that

\[ F \subseteq_c G = (\forall H : F[H] \subseteq\circ G[H]). \]

The implication from left to right follows from theorem 3.5 and the definition of composition \( \mid \) in \( C \). Here, we focus on the implication from right to left:

\[ F \subseteq_c G \iff (\forall H : F[H] \subseteq\circ G[H]). \]

We prove this by contraposition. For UNITY programs \( F \) and \( G \) such that \( F \not\subseteq_c G \), we construct a UNITY program \( H \) such that \( F[H] \not\subseteq\circ G[H] \), i.e. there exists an execution sequence \( z \in \mathcal{O}[G[H]] \) that is not in \( \mathcal{O}[F[H]] \).

If \( \text{init}(F) \not\supseteq \text{init}(G) \), the program \( H \) with initial condition true and an \textit{assign}-section only containing \textit{skip} will do. Now, assume that there exists an extended sequence \( v \) in \( C[G] \) that is not in \( C[F] \). So, \( v \) does not satisfy one of the conditions of definition 3.4 for program \( F \). We have to examine the following cases:

1. There exists index \( i \) and \( v(i) = (\sigma_1, \sigma_2) \) such that

\[ \sigma_1 \neq \sigma_2 \land (\forall S : S \in \text{assign}(F) : \neg (\sigma_1 \xrightarrow{S} \sigma_2)). \]

Since \( v \) is an extended sequence of \( G \),

\[ (\exists S : S \in \text{assign}(G) : \sigma_1 \xrightarrow{S} \sigma_2). \]

To make state \( \sigma_1 \) reachable, construct \( H \) as follows (here \( \text{state} := \sigma_1 \) is a multiple assignment that sets the values of all the variables to their values in \( \sigma_1 \)):

\[
\begin{align*}
\text{Program } & H \\
\text{assign } & \text{state} := \sigma_1 \\
\text{end} \{ H \}
\end{align*}
\]

Now, construct a sequence \( z \) by first executing the only step of \( H \) and then making the transition to \( \sigma_2 \). Then, \( z \in \mathcal{O}[G[H]] \) and \( z \not\in \mathcal{O}[F[H]] \).

2. There is a difference in fairness: \( \text{fair}(G, v) \) and \( \neg \text{fair}(F, v) \). Hence, there is a statement \( S \in \text{assign}(F) \) and an index \( i \) such that

\[ (\forall j : j \geq i : \neg \text{step}(S, v(j)) \land \neg \text{inbetween}(S, v(j - 1), v(j))). \]

Consider the cases that the sequence \( v \) is connected from \( i \) onwards, or not.
(a) Suppose \( v \) is connected from \( i \) onwards, i.e.

\[
\langle \forall j : j \geq i : v.2(j) = v.1(j+1) \rangle.
\]

In that case, we do not need an environment to fill the gap; we only have to take care that this subsequence is reachable from an initial state. If there exists an index \( k \geq i \) such that \( v.1(k) \in \text{init}(G) \) then the sequence

\[
\langle v.1(k), v.1(k+1), v.1(k+2), \ldots \rangle
\]

is a sequence of \( \mathcal{O}[G[I]] \), but not of \( \mathcal{O}[F[I]] \), where \( I \) is the program with \( \text{init}(I) = \text{true} \) and \( \text{assign}(I) = \{\text{skip}\} \). In case no state in the sequence is an initial state of \( G \) we construct the program \( H \):

```
Program H
assign init(G) \rightarrow state := v.1(i)
end{H}
```

Again, by first executing the only step of \( H \) and continuing with \( v \) from \( i \) we obtain a connected extended sequence of \( C[G[H]] \) but not of \( C[F[H]] \). Since the statement of \( H \) is not enabled in any state of \( v \), it is executed fairly.

(b) Suppose \( v \) is not connected from \( i \). Then we can pick an index \( k \geq i \) such that \( v.2(k-1) \neq v.1(k) \) and choose \( l \) such that for each statement \( T \in \text{assign}(G) \),

\[
\langle \exists j : k < j \leq l : \text{step}(T, v(j)) \lor \text{inbetween}(T, v(j-1), v(j)) \rangle.
\]

This is possible because \( \text{fair}(G, v) \).

Construct \( v' \) as follows:

\[
v' = \langle v(k), \ldots, v(l), v(k), \ldots, v(l) \ldots \rangle.
\]

Since \( v.2(k-1) \neq v.1(k) \) and \( \neg(\text{inbetween}(S, v(k-1), v(k))) \), we have that \( \neg(\text{inbetween}(S, v(l), v(k))) \). So, for the new extended sequence \( \text{fair}(G, v') \) and \( \neg\text{fair}(F, v') \) hold.

Now, we construct a UNITY program \( H \) that can fill all gaps in \( v' \) in such a way that the effect of statement \( S \) never occurs. Define

\[
\text{fill}(S, (\sigma_1, \sigma_2), (\sigma_3, \sigma_4)) \overset{\text{def}}{=} \sigma_2 \neq \sigma_3 \land \sigma_2 \not\rightarrow^S \sigma_3,
\]

which states that \( S \) fills the gap between \( (\sigma_1, \sigma_2) \) and \( (\sigma_3, \sigma_4) \). For every gap that can be filled by statement \( S \), we have to provide an alternative way to fill the gap. Because \( \langle \forall j : \neg\text{inbetween}(S, v'(j-1), v'(j)) \rangle \), we can find for every \( j \) such that

\[
\text{fill}(S, v'(j-1), v'(j))
\]
an intermediate state $\sigma_j$ such that

$$\neg (\text{step}(S, (\sigma_j, v'.1(j)))).$$

Then, the environment $H$ is constructed as follows:

$$\text{Program } H$$

$$\begin{align*}
\text{assign} \\
\langle [j : j \leq l - k : \text{state} := v'.1(j)] \ |
\langle [j : j \leq l - k \land \text{fill}(S, v'(j - 1), v'(j)) : \text{state} := \sigma_j] \\
\end{align*}$$

$$\text{end}\{H\}$$

This program can fill the gap between the pairs $v'_j$ and $v'_{j+1}$ with the pairs

$$\left\{\begin{array}{ll}
(v'.2(j), \sigma_j), (\sigma_j, v'.1(j + 1)) & \text{if } \text{fill}(S, v'(j), v'(j + 1)) \\
(v'.2(j), v'.1(j + 1)) & \text{otherwise}.
\end{array}\right.$$

This does not introduce the effect of statement $S$: if $\text{fill}(S, v'(j), v'(j + 1))$ then

$$\neg (v'.2(j) \xrightarrow{S} \sigma_j)$$

because $S$ is deterministic.

\hfill \square

A direct consequence of this theorem is that $C$ is the best compositional model with respect to $O$.

**Corollary 3.11** Model $C$ is the best compositional model with respect to $O$.

### 3.2 Property based Semantics

In this section we examine the right hand side of figure 3.1. We define semantic models for UNITY programs in terms of UNITY properties. The meaning of a program is the set of all its properties. Each model consists of three parts: a predicate giving the set of initial states, a set of safety properties (unless), and a set of progress properties. We can build a number of different models in this way by varying the kinds of properties. We can choose between the original properties as defined by Chandy and Misra and the properties as given by Sanders. We can also choose between ensures properties and \to properties for modelling progress. This results in four different models. Let $P$ be the property domain, i.e. the domain of pairs of predicates: $P \overset{\text{def}}{=} \text{Pred} \times \text{Pred}$. We use the domain $\text{Spec} \overset{\text{def}}{=} \text{Pred} \times \mathcal{P}(P) \times \mathcal{P}(P)$ containing initial states, and two sets of properties.
Definition 3.12 The property based models for UNITY programs are functions in \( \text{UNITY} \to \text{Spec} \) and are defined by

\[
\begin{align*}
\text{IU} & \text{CM} [F] \overset{\text{def}}{=} (I, U_{CM}, E_{CM}), \\
\text{IU} & \text{L} [F] \overset{\text{def}}{=} (I, U_{CM}, L_{CM}), \\
\text{IU} & \text{E} [F] \overset{\text{def}}{=} (I, U_{S}, E_{CM}), \\
\text{IU} & \text{L} [F] \overset{\text{def}}{=} (I, U_{S}, L_{S}),
\end{align*}
\]

where

\[
\begin{align*}
I & \overset{\text{def}}{=} \text{init}(F), \\
U_{CM} & \overset{\text{def}}{=} \{ (p, q) \mid p \text{ unless}_{CM} q \text{ in } F \}, \\
E_{CM} & \overset{\text{def}}{=} \{ (p, q) \mid p \text{ ensures}_{CM} q \text{ in } F \}, \\
L_{CM} & \overset{\text{def}}{=} \{ (p, q) \mid p \leftrightarrow_{CM} q \text{ in } F \}, \\
U_{S} & \overset{\text{def}}{=} \{ (p, q) \mid p \text{ unless}_{S} q \text{ in } F \}, \\
E_{S} & \overset{\text{def}}{=} \{ (p, q) \mid p \text{ ensures}_{S} q \text{ in } F \}, \\
L_{S} & \overset{\text{def}}{=} \{ (p, q) \mid p \leftrightarrow_{S} q \text{ in } F \}.
\end{align*}
\]

As is shown in figure 3.1, model \( \text{IU} \text{CM} \) is the strongest model, which captures all ensures properties of a program, so leadsto properties can be derived. Furthermore, it captures the initial section and all unless properties (and therefore all stable properties). Hence, invariants and properties in Sanders’s logic can be derived.

Model \( \text{IU} \text{CM} \) is the only compositional model. Its compositionality is a direct consequence of the UNITY union theorem 2.9 which states that unless\(_{CM}\) and ensures\(_{CM}\) can be expressed in terms of these properties for the components. The other models are not compositional since they are based on the leadsto property, which is non-compositional, or since they take invariants into account. These sources of non-compositionality were discussed in section 2.2.

### 3.2.1 Refinement

We gave four semantic models based on properties. These models are now used to define notions of refinement as preservation of properties. The UNITY framework supports refinement of specifications and specifications are sets of properties that must be satisfied by the implementation. This notion can be lifted to refinement of programs. A program \( F \) is refined by a program \( G \) if \( G \) satisfies every specification satisfied by \( F \); in other words, if every property of \( F \) is also a property of \( G \). First, we formalise the refinement ordering in the domain of specifications \( \text{Spec} \).

Definition 3.13 For specifications \( S_1, S_2 \in \text{Spec} \) we define the ordering

\[
S_1 \sqsubseteq \text{Spec} S_2 \overset{\text{def}}{=} I_1 \supseteq I_2 \land U_1 \subseteq U_2 \land E_1 \subseteq E_2.
\]
where \( S_1 = (I_1, U_1, E_1) \) and \( S_2 = (I_2, U_2, E_2) \).

This ordering is used to define notions of refinement for the property based models given in definition 3.12.

**Definition 3.14** For UNITY programs \( F \) and \( G \), we define refinement in property based models by:

\[
\begin{align*}
F \sqsubseteq_{\text{UE}_{\text{CM}}} G & \quad \text{def} \quad \text{UE}_{\text{CM}}[F] \sqsubseteq_{\text{Spec}} \text{UE}_{\text{CM}}[G], \\
F \sqsubseteq_{\text{UL}_{\text{CM}}} G & \quad \text{def} \quad \text{UL}_{\text{CM}}[F] \sqsubseteq_{\text{Spec}} \text{UL}_{\text{CM}}[G], \\
F \sqsubseteq_{\text{UE}_{s}} G & \quad \text{def} \quad \text{UE}_{s}[F] \sqsubseteq_{\text{Spec}} \text{UE}_{s}[G], \\
F \sqsubseteq_{\text{UL}_{s}} G & \quad \text{def} \quad \text{UL}_{s}[F] \sqsubseteq_{\text{Spec}} \text{UL}_{s}[G].
\end{align*}
\]

### 3.2.2 Best Compositional Refinement

The model \( \text{UE}_{\text{CM}} \) is compositional and is stronger than the other property based models. We prove that refinement in \( \text{UE}_{\text{CM}} \) is the best compositional notion of refinement with respect to \( \text{UL}_{s} \).

**Theorem 3.15** The refinement notion \( \sqsubseteq_{\text{UE}_{\text{CM}}} \) is the best compositional notion of refinement with respect to the refinement relation \( \sqsubseteq_{\text{UL}_{s}} \).

**Proof:** We have to prove that

\[
F \sqsubseteq_{\text{UE}_{\text{CM}}} G \quad \Leftrightarrow \quad \langle \forall H :: F[H] \sqsubseteq_{\text{UL}_{s}} G[H] \rangle.
\]

Since \( \text{UL}_{s} \) is an abstraction of \( \text{UE}_{\text{CM}} \) and the latter model is compositional, we only have to prove that for UNITY programs \( F \) and \( G \)

\[
F \sqsubseteq_{\text{UE}_{\text{CM}}} G \quad \Leftrightarrow \quad \langle \forall H :: F[H] \sqsubseteq_{\text{UL}_{s}} G[H] \rangle.
\]

We prove this by contraposition. For UNITY programs \( F \) and \( G \) such that \( F \not\sqsubseteq_{\text{UE}_{\text{CM}}} G \), we construct a UNITY program \( H \) such that \( F[H] \not\sqsubseteq_{\text{UL}_{s}} G[H] \), i.e. there exists a property in \( \text{UL}_{s}[F[H]] \) that is not a property of \( \text{UL}_{s}[G[H]] \). We distinguish three cases:

1. Program \( F \) and \( G \) differ in initially-properties: \( \neg[\text{init}(F) \iff \text{init}(G)] \). Take any UNITY program \( H \) such that \( \text{init}(H) = \text{true} \). Then

\[

\neg[\text{init}(F[H]) \iff \text{init}(G[H])]
\]

and hence, \( F[H] \not\sqsubseteq_{\text{UL}_{s}} G[H] \).
2. Suppose there exist predicates $p$ and $q$ such that property $p \text{ unless}_{CM} q$ in $F$ holds and property $p \text{ unless}_{CM} q$ in $G$ does not. We construct a program $H$ such that $p \text{ unless}_{CM} q$ in $F|H$ holds and $p \text{ unless}_{CM} q$ in $G|H$ does not. Since $p \text{ unless}_{CM} q$ in $G$ does not hold, there exists a state $\sigma$ such that

$$\exists S : S \in \text{assign}(G) : (p \land \neg q)(\sigma) \land \neg S(p \lor q)(\sigma)).$$

We construct program $H$ that makes the state $\sigma$ reachable.

$$\text{Program } H$$
$$\text{assign } \text{state} := \sigma$$
$$\text{end}\{H\}$$

Then:

$$\neg(p \lor q)(\sigma)$$
$$\Rightarrow \{\text{assignment}\}$$
$$[(\text{state} := \sigma)(p \lor q)]$$
$$\Rightarrow \{\text{predicate calculus}\}$$
$$[(p \land \neg q) \Rightarrow (\text{state} := \sigma)(p \lor q)]$$
$$\Rightarrow \{\text{definition}\}$$
$$p \text{ unless}_{CM} q \text{ in } H$$

Consequently, $p \text{ unless}_{CM} q$ in $F|H$, and $p \text{ unless}_{S} q$ in $F|H$. However, since the state $\sigma$ is reachable in $G|F$, for every predicate $r$ such that $\text{invariant}_{CM} r$ in $G|H$, $r(\sigma)$ holds, so property $p \text{ unless}_{S} q$ in $G|H$ does not hold.

3. Programs $F$ and $G$ differ in $\text{ensures}_{CM}$ properties: there are state predicates $p$ and $q$ such that $p \text{ ensures}_{CM} q$ in $F$ holds and $p \text{ ensures}_{CM} q$ in $G$ does not. We construct a program $H$ such that $p \Rightarrow q$ in $F|H$ holds and $p \Rightarrow q$ in $G|H$ does not. Due to the previous cases we may assume that $\text{init}(F) = \text{init}(G)$ as well as $p \text{ unless}_{CM} q$ in $F = p \text{ unless}_{CM} q$ in $G$. Then:

$$\neg(p \text{ ensures}_{CM} q \text{ in } G)$$
$$= \{ \text{ definition } \text{ensures}_{CM} \}$$
$$\neg(p \text{ unless}_{CM} q \text{ in } G)$$
$$\lor \left( \forall S : S \in \text{assign}(G) : \exists \sigma : \sigma \in \Sigma : (p \land \neg q)(\sigma) \land \neg S(q)(\sigma) \right)$$
$$= \{ p \text{ unless}_{CM} q \text{ in } F \text{ and assumption } \}$$
$$\forall S : S \in \text{assign}(G) : (\exists \sigma : \sigma \in \Sigma : (p \land \neg q)(\sigma) \land \neg S(q)(\sigma))$$
Pick such a state $\sigma_S$ for every statement $S$ in $G$ and construct $H$ as follows:

Program $H$

assign $\{ |S : S \in \text{assign}(G) : \text{state} := \sigma_S \}$
end$\{H\}$

Since for every $S \in \text{assign}(G)$ $(p \land \neg q)(\sigma_S)$, property $p$ unless$_{CM}$ $q$ in $H$ holds. Consequently, $p$ ensures$_{CM}$ $q$ in $F[H]$, and by definition of leadsto, $p \iff_{CM}$ $q$ in $F[H]$, and $p \iff_S q$ in $F[H]$ hold. However, $p \iff_S q$ in $G[H]$ does not hold. This is shown by constructing a sequence $z$ of $G[H]$ in which $(p \land \neg q)$ holds continuously, so $p$ holds, but $q$ is never reached (lemma 3.18). This sequence is constructed by repeatedly executing (in a fair way) a statement $\text{state} := \sigma_S$ of $H$ followed by the corresponding statement $S$ of $G$. After the execution of $\text{state} := \sigma_S$, state $\sigma_S$ is reached in which $p \land \neg q$ holds. Since $p$ unless$_{CM}$ $q$ in $G$ holds, execution of $S$ reaches a state in which $p \lor q$. Furthermore, $\neg S(q)(\sigma_S)$, and hence, $S^q(\neg q)$, execution of $S$ reaches a state in which $p \land \neg q$ holds.

So, $F[H] \not\subseteq_{\text{IU} L_S} G[H]$. 

\[ \square \]

Theorem 3.15 has two important consequences. First, since all property based models are abstractions of $\text{IU} \mathcal{E}_{CM}$, refinement in $\text{IU} \mathcal{E}_{CM}$ is the best compositional notion of refinement with respect to all property based models.

Corollary 3.16 Refinement relation $\subseteq_{\text{IU} \mathcal{E}_{CM}}$ is the best compositional refinement relation with respect to the refinement relations $\subseteq_{\text{IU} \mathcal{L}_{CM}}$, $\subseteq_{\text{IU} \mathcal{E}_S}$, and $\subseteq_{\text{IU} \mathcal{L}_S}$.

Furthermore, the model $\text{IU} \mathcal{E}_{CM}$ is the best compositional model with respect to the other property based models.

Corollary 3.17 Model $\text{IU} \mathcal{E}_{CM}$ is the best compositional model with respect to $\text{IU} \mathcal{L}_{CM}$, $\text{IU} \mathcal{E}_S$, and $\text{IU} \mathcal{L}_S$.

3.3 Sequences versus Properties

In the previous sections we described two categories of semantic models for UNITY programs: in section 3.1 two models based on execution sequences and in section 3.2 four models based on UNITY properties. In this section we examine the relation between these categories of models.

First, we examine the relation between $\mathcal{O}$ and $\text{IU} \mathcal{L}_S$. These models give the notion of observability. As already mentioned in section 2.2, properties in Sanders’s logic have a nice interpretation in terms of execution sequences. This is expressed by the following lemma [UK93b].
**Lemma 3.18** For a UNITY program $F$ and state predicates $p$ and $q$,

\[ p \text{ unless}_S q \text{ in } F = (\forall z, i : z \in \mathcal{O}[F] : (p \land \neg q)(z(i)) \Rightarrow (p \lor q)(z(i + i))), \]

\[ p \rightarrow_S q \text{ in } F = (\forall z, i : z \in \mathcal{O}[F] : p(z(i)) \Rightarrow (\exists j : j \geq i : q(z(j)))). \]

As a direct consequence, refinement in $\mathcal{UL}_S$ can be derived from refinement in $\mathcal{O}$.

**Theorem 3.19** For UNITY programs $F$ and $G$,

\[ F \subseteq_{\mathcal{O}} G \Rightarrow F \subseteq_{\mathcal{UL}_S} G. \]

Theorem 3.15 states that refinement in $\mathcal{UL}_{CM}$ is the best compositional notion of refinement with respect to refinement in $\mathcal{UL}_S$. Combined with lemma 3.18 we may conclude that execution sequences are more expressive than properties; programs that have the same set of extended sequences satisfy the same properties.

**Theorem 3.20** For UNITY programs $F$ and $G$,

\[ F \subseteq_{\mathcal{C}} G \Rightarrow F \subseteq_{\mathcal{UL}_{CM}} G. \]

---

**Example 3.5.**

The converse of theorem 3.19 does not hold; $\text{unless}_S$ and $\rightarrow_S$ are not powerful enough to characterise sets of execution sequences. Consider programs $F$ and $G$ as given in example 3.5. Using the relation between execution sequences and properties in Sanders’s logic (lemma 3.18), it is easy to prove that both programs have the same $\text{unless}_S$ and $\rightarrow_S$ properties. (Both programs also have the same $\rightarrow_{CM}$ properties, but this is more difficult to prove.) Programs $F$ and $G$ have different sets of execution sequences. For example, sequence $\langle 0, 1, 2, 3, \cdots \rangle$ (identifying states by the value of $x$) is an execution sequence of $G$. Since UNITY fairness is assumed, this sequence is not an execution sequence of $F$. The last statement of $F$ is executed infinitely often, so each sequence of $F$ contains a step in which the value of $x$ is increased by two. So, a combination of $\text{unless}_S$ and $\rightarrow_S$ properties is not powerful enough to characterise sets of execution sequences.

There seem to be two ways to resolve the difference between execution sequences and ($\text{unless}$ and $\rightarrow$) properties of a UNITY program.
1. Use stronger properties; use \(\text{ensures}_S\) properties instead of \(\rightarrow_S\) properties as suggested by Misra in [Mis90a]. The programs given above differ in ensures properties:

\[
\text{true \, ensures}_S \left( (x \mod 4 = 0) \lor (x \mod 4 = 1) \right)
\]

is a property of \(F\), but not of \(G\). (This example is due to Lambert Meertens.)

2. Change the set of possible execution sequences by adopting a different notion of fairness; for example strong fairness or compassion [MP92]: it is not the case that the condition of a statement holds infinitely often, and that it is only executed a finite number of times. Assuming this notion of fairness, \(F\) and \(G\) have the same execution sequences. We could also choose a number of different notions of fairness based on the conditions of statements (enabledness) [Fra86]. However, the next example, which only has unconditional statements, shows that these notions are not useful for our goal.

---

**Program** \(H\)

- **init**
  \(x \in \{0, 1, 2\}\)

- **assign**
  - \(a\) \(x := x \mod 3\)
  - \(b\) \(x := (x + 1) \mod 3\)
  - \(c\) \(x := (x + 2) \mod 3\)

**Program** \(K\)

- **init**
  \(x \in \{0, 1, 2\}\)

- **assign**
  - \(d\) \(x := -x \mod 3\)
  - \(e\) \(x := (1 - x) \mod 3\)
  - \(f\) \(x := (2 - x) \mod 3\)

---

**Example 3.6.**

The following programs show that both ways do not work. Consider programs \(H\) and \(K\) as given, together with their state transition diagrams, in example 3.6. First, we show
that both programs have the same properties. It is clear that the initially-sections of both programs are the same. Since their state transition diagrams are the same (statement labels cannot be observed), the programs have the same unless\(_\text{CM}\) properties. Moreover, the ensures\(_\text{CM}\) properties are the same: in table 3.7 all combinations of predicates \((p \land \neg q)\) and \(q\) are listed, and for each combination is indicated whether or not a statement exists such that \([ (p \land \neg q) \Rightarrow S(q) ]\) holds. A one in the table indicates that such a statement exists, a zero denotes that such a statement does not exist, and — indicates that the combination of predicates is not possible, i.e. when \(\langle \exists \sigma :: (p \land \neg q)(\sigma) \land q(\sigma) \rangle\) implying \(\langle \exists \sigma :: q(\sigma) \land \neg q(\sigma) \rangle\), which is impossible. The predicates \((p \land \neg q)\) and \(q\) are given by the set of values for \(x\) on which they hold. It is easy to check that the table holds for both programs \(H\) and \(K\). From this table and the fact that the unless\(_\text{CM}\) properties of both programs are the same, it can be concluded that the ensures\(_\text{CM}\) properties of both programs are the same. Consequently, the UNITY properties of both programs are the same. However, the execution sequences of \(H\) and \(K\) differ. For example, the sequence \(\langle 0,1,2,0,1,2,\cdots \rangle\) is an execution sequence of \(K\) but not of \(H\). This sequence of states is not fair in \(H\). To produce this sequence, the last statement of program \(H\) should be ignored forever.

To summarise, programs \(H\) and \(K\) have the same unless\(_\text{CM}\) and ensures\(_\text{CM}\) properties but they differ in their execution sequences. Moreover, since no statement has a condition, it does not matter whether we adopt UNITY fairness or one of the stronger notions of fairness. So, UNITY properties are not expressive enough to characterise execution sequences. Since extended sequences are more expressive than execution sequences, UNITY properties are not expressive enough to characterise extended sequences either. Therefore, the converse of theorem 3.20 does not hold.

<table>
<thead>
<tr>
<th>(p \land \neg q)</th>
<th>(q)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>01</th>
<th>02</th>
<th>12</th>
<th>012</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\emptyset)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>—</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>02</td>
<td>0</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>012</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 3.7. \(\exists S :: [(p \land \neg q) \Rightarrow S(q)]\), for both programs \(H\) and \(K\).

From Temporal Logic it is known that the use of auxiliary location variables may
increase the expressive power of properties. This method is of no help here since the flow of control in a UNITY program is very simple: there is in fact only one location.

This example shows that reduction of execution sequences, as used in temporal logics [AL88] and actions systems [BvW94], and preservation of properties, as proposed by Sanders [San90] and Singh [Sin93], are different notions of refinement.

3.4 Conclusions

In this chapter we presented several semantic models for UNITY programs. We described two categories of models: operational models based on execution sequences, and models based on UNITY properties. For both categories we gave a best compositional model. We showed how the models can be used to define notions of refinement of UNITY programs. The compositional models gave the best compositional notion of refinement with respect to refinement notions induced by the other models. We related the different semantic models for UNITY programs and discussed different observation criteria. The differences between the models were shown by some examples.
The union operator of UNITY framework is a basic operation without much structure: components communicate by shared variables, and all variables of a component are shared variables. Most parallel and distributed programming languages support more sophisticated ways of program composition. A component may have local variables, or variables that can only be read or only be written by an environment, and often there are other means of communication than communication via shared variables. In the UNITY design process, these aspects must be handled by mappings from UNITY programs to the target implementation.

We introduce the ImpUNITY framework, an extension of the UNITY framework in which more structure is added to program composition. ImpUNITY supports the following features about the way programs are composed and how they communicate with each other.

- Interference: restrictions on how an environment can interact with a program.
- Observability: restrictions on how the state space can be observed.
- Procedures: offering an alternative way of communication between components.

We will introduce the ImpUNITY framework in three steps. In chapter 4 we introduce interference, chapter 5 is about observability and chapter 7 deals with procedures. In chapter 6 and 8 we present two case studies in the ImpUNITY formalism.
Chapter 4

Interference

If two programs are composed, they interfere with each other. The standard union operator in UNITY allows programs to interfere freely. In this chapter we propose a way to restrict interference. We introduce so-called modifiers, predicate transformers expressing in which way the state can be changed. Section 4.1 deals with the notion of modifiers in the refinement calculus. In section 4.2 it is shown how modifiers are incorporated into the ImpUNITY programming language. Section 4.3 gives the ImpUNITY logic. This logic takes into account that an environment cannot interfere freely. Section 4.4 gives a set of program transformation rules.

4.1 Modifiers

In this section we introduce modifiers, statements that can be seen as lower bounds of the allowed behaviour. In the ImpUNITY framework we will use modifiers to restrict the interference of an environment. Due to the structure of ImpUNITY programs, we cannot force an environment to interfere at a specific point and we cannot force the environment to interfere by only executing a bounded number of statements. Therefore, we define modifiers as reflexive and idempotent statements.

Definition 4.1 A statement $M$ is a modifier iff

\begin{align*}
M \leq \text{skip}, \\
M; M = M.
\end{align*}

Alternatively, we could have defined that a statement $M$ is a modifier iff $M = M^*$. Hence, for a statement $S$, the transitive and reflexive closure $S^*$ is a modifier. Furthermore, a conjunctive equivalence command is a special kind of modifier. For example, the command $[=_{x}]$, for a set of variables $X$, is the modifier stating that the values of variables in $X$ cannot be changed.
A modifier gives a restriction on transitions. If a command satisfies the restriction induced by a modifier, then we say that it obeys the modifier. Although we mostly use modifiers as a restriction on statements, we define the notion of obeying modifier for the more general notion of commands. Since a modifier is a lower bound of the allowed command, a command obeys a modifier if it is a refinement of the modifier.

**Definition 4.2** For command $A$ and modifier $M$, by $A \text{ obey } M$ we denote that $A$ obeys $M$ which is defined by

$$A \text{ obey } M \overset{\text{def}}{=} M \leq A.$$

**Lemma 4.3** For modifier $M$ and command $A$,

$$(M \leq A) = (M \leq A^*).$$

**Proof:**

$$M \leq A^*$$

$$\Rightarrow \{A^* \leq A\}$$

$$M \leq A$$

$$\Rightarrow \{\text{refinement calculus}\}$$

$$M^* \leq A^*$$

$$= \{M \text{ is a modifier}\}$$

$$M \leq A^*$$

\[\square\]

As a corollary, we have that if a modifier allows some interference, it allows that interference repeatedly.

**Corollary 4.4** For modifier $M$ and command $A$,

$$(A \text{ obey } M) = (A^* \text{ obey } M).$$

The refinement ordering is used to express that a command obeys a modifier; the ordering also has a useful interpretation for comparing modifiers. Refinement of modifiers corresponds to restricting interference. For modifiers $M$ and $M'$, such that $M \leq M'$, $M$ allows more state changes than $M'$. The modifier $\text{skip}$ is the most restrictive modifier, allowing no changes at all. The following lemma expresses an (anti-) monotonicity property of obey.
Lemma 4.5 Let $M$ and $M'$ be modifiers, then for any command $A$

\[ M \leq M' \implies (A \text{ obey } M' \implies A \text{ obey } M). \]

That a command $A$ obeys modifiers $M_1$ and $M_2$ can be expressed by $M_1 \lor M_2 \leq A$. However, in general the command $M_1 \lor M_2$ will not be a modifier since it is not always conjunctive and therefore not a statement. Now, we define the statement $M_1 + M_2$ as the strongest modifier that is a refinement of $M \lor M'$. Then, for a modifier $M$ we have that $M \text{ obey } (M_1 + M_2)$ expresses that $M$ obeys both modifiers.

Definition 4.6 Let $M_1$ and $M_2$ be modifiers, and let $T$ be the set of modifiers $T \overset{\text{def}}{=} \{ M \mid M_1 \lor M_2 \leq M \}$. Then,

\[ M_1 + M_2 \overset{\text{def}}{=} \land_{M \in T} M. \]

It is easy to verify that $\land_{M \in T} M = (\land_{M \in T} M)^*$, so command $M_1 + M_2$ is also a modifier. First, we prove that $M_1 + M_2$ is a refinement of $M_1 \lor M_2$.

Lemma 4.7 For modifiers $M_1$ and $M_2$,

\[ M_1 \lor M_2 \leq M_1 + M_2. \]

Proof: Let $T \overset{\text{def}}{=} \{ M \mid M_1 \lor M_2 \leq M \}$, then

\[
M_1 \lor M_2 \leq M_1 + M_2 \\
= \{ \text{definition} \} \\
M_1 \lor M_2 \leq \land_{M \in T} M \\
= \{ \text{refinement calculus} \} \\
(\forall M : M \in T : M_1 \lor M_2 \leq M) \\
= \{ \text{definition } T \} \\
true
\]

\[ \square \]

Second, we prove that every statement (especially every modifier) that is a refinement of $M_1 \lor M_2$, is also a refinement of $M_1 + M_2$.

Lemma 4.8 Let $M_1$ and $M_2$ be modifiers. Then, for every statement $S$,

\[ (M_1 \lor M_2 \leq S) = (M_1 + M_2 \leq S). \]
**Proof:** The implication from right to left follows directly from lemma 4.7. Here we focus on the implication from left to right. Let $T \overset{\text{def}}{=} \{ M \mid M_1 \lor M_2 \leq M \}$. Suppose $M_1 \lor M_2 \leq S$, then

$$M_1 \lor M_2 \leq S \Rightarrow \{ \text{refinement calculus} \}$$

$$(M_1 \lor M_2)^* \leq S^* \Rightarrow \{ M^* \leq (M \lor M')^* \}$$

$M_1^* \lor M_2^* \leq S^* = \{ \text{definitions} \}$$

$M_1 \lor M_2 \leq S^*$

and consequently, $S^* \in T$. Furthermore,

$$M_1 + M_2 = \{ \text{definition} \}$$

$$\bigwedge_{M \in T} M$$

$$\leq \{ S^* \in T \}$$

$$S^* \leq \{ \text{refinement calculus} \}$$

$$S \quad \Box$$

This results in the following theorem.

**Theorem 4.9** For statement $S$ and modifiers $M_1$ and $M_2$,

$$(S \text{ obey } M_1 \land S \text{ obey } M_2) = S \text{ obey } (M_1 + M_2).$$

The following lemma shows how to calculate the strongest modifier for a pair of equivalence commands.

**Lemma 4.10** For sets of variables $X$ and $Y$,

$$[=X] + [=Y] = [=X \cup Y].$$
4.2 Modifiers in ImpUNITY

In this section we introduce a part of the ImpUNITY programming language. Like program in the UNITY formalism, an ImpUNITY program has an initially and an assign-section. ImpUNITY programs operate in an overall state space $\Sigma_U$ based on the set of all ImpUNITY program variables $U$. The initially-section contains a predicate on $\Sigma_U$ giving the possible initial states of the program. We require that at least one initial state exists. The assign-section of an ImpUNITY program consists of a set of statements on $\Sigma_U$. We do not require that the statements are deterministic. Furthermore, to restrict interference, an ImpUNITY program also has the following additional section:

- The external-section, containing a modifier specifying in which way an environment is allowed to change the state. This modifier is denoted by $\text{external}(F)$.

The external-section models a restriction on the environment. Definition 4.2 states what it means for a statement to obey a modifier. This idea can easily be lifted to ImpUNITY programs.

**Definition 4.11** For ImpUNITY program $F$ and modifier $M$, define

$$F \text{ obey } M \overset{\text{def}}{=} (\forall S : S \in \text{assign}(F) : S \text{ obey } M).$$

Then, an environment of $F$ is defined as a program that obeys the interference restriction of $F$. A context of $F$ is an environment whose interference restriction is obeyed by $F$.

**Definition 4.12** For ImpUNITY programs $F$ and $H$, by $F \text{ env } H$ we denote that $H$ is an environment of $F$ which is defined by

$$H \text{ env } F \overset{\text{def}}{=} H \text{ obey } \text{external}(F).$$

By $H \text{ cont } F$ we denote that $H$ is a context of $F$ which is defined

$$H \text{ cont } F \overset{\text{def}}{=} F \text{ env } H \land H \text{ env } F.$$

Modifier $[\cdots_U]$ is the least restrictive modifier in the sense that it allows all interference, since $U$ is the set of all ImpUNITY programming variables. So, if the external-section contains the modifier $[\cdots_U]$, the program corresponds to a UNITY program. In that case, the external-section is not mentioned explicitly. If the external-section contains the modifier skip, no interference is allowed and the program can be seen as a closed system.

For example, consider ImpUNITY program $F$ in example 4.1. The external-section contains the modifier $[\cdots_U]$ and specifies that an environment is not allowed to change variable $y$. Since variable $y$ can be read by an environment, $y$ may be called an output variable [UHK94] or a private variable [CWB94] of $F$. However, modifiers can express
Program $F$

external $\{ y \}$

init $y = 0$

assign $x, y := x + 1, 10$

[$y > 0 \rightarrow y := y - 1$

end{$F$}

Example 4.1.

more general forms of interference. This will be shown in chapter 6 in which a register component is specified and refined.

External interference of a program can be further restricted by the $ext$ operator that adds to the modifier in the $external$-section of the program.

**Definition 4.13** For ImpUNITY program $F$ and modifier $M$, ImpUNITY program $ext(M : F)$ only differs from $F$ in the $external$-section:

$$external(ext(M : F)) \overset{\text{def}}{=} M + external(F).$$

Consider program $F$ in example 4.1. The $external$-section of the program $ext(\{x\} : F)$ is the modifier $\{x, y\}$, stating that neither variable $x$ nor variable $y$ can be changed by an environment.

The union operator of ImpUNITY programs is an extension of the standard union operator in UNITY.

**Definition 4.14** For ImpUNITY programs $F$ and $G$, the union $F \parallel G$ is defined by

$$external(F \parallel G) \overset{\text{def}}{=} (external(F) + external(G)),$$

$$init(F \parallel G) \overset{\text{def}}{=} (init(F) \land init(G)),$$

$$assign(F \parallel G) \overset{\text{def}}{=} (assign(F) \cup assign(G)).$$

The $external$-section of the union is constructed from the $external$-sections of both components in such a way that an environment of the union satisfies the interference restrictions of both components. The following lemma states that it is possible to examine all components separately when checking the environment conditions.

**Lemma 4.15** For ImpUNITY programs $F$, $G$ and $H$,

$$H env F \parallel G = H env F \land H env G,$$

$$F \parallel G env H = F env H \land G env H.$$
Preservation of \textit{unless} and \( \rightarrow_S \) properties is expressed in the notion of observable refinement which is defined as follows.

**Definition 4.16** For ImpUNITY programs \( F \) and \( G \), \( G \) is an observable refinement of \( F \), denoted by \( F \sqsubseteq_o G \), if for all predicates \( p \) and \( q \),

\[
\begin{align*}
p \text{ unless}_S q \text{ in } F \Rightarrow p \text{ unless}_S q \text{ in } G, \\
p \rightarrow_S q \text{ in } F \Rightarrow p \rightarrow_S q \text{ in } G.
\end{align*}
\]

Note that the \textit{external}-section does not play a role in observational refinement. Programs \( F \) and \( G \) that only differ in their \textit{external}-sections are observable refinements of each other, i.e. \( F \sqsubseteq_o G \) and \( G \sqsubseteq_o F \).

Since we want to have a compositional notion of refinement, we define refinement of ImpUNITY programs as observable refinement in any environment.

**Definition 4.17** For ImpUNITY programs \( F \) and \( G \), by \( F \sqsubseteq G \) we denote that \( G \) is a refinement of \( F \), which is defined by

\[
F \sqsubseteq G \overset{\text{def}}{=} (\forall H : H \text{ env } F \land H \sqsubseteq_o G[H]).
\]

The notion of refinement is compositional in the following sense.

**Lemma 4.18** For ImpUNITY programs \( F \), \( G \) and \( H \), such that \( H \text{ env } F \),

\[
F \sqsubseteq G \Rightarrow F[H] \sqsubseteq G[H].
\]

**Proof:**

\[
\begin{align*}
F \sqsubseteq G \\
\Rightarrow & \quad \{\text{definition}\} \\
(\forall H, H' : H[H'] \text{ env } F : H[H'] \sqsubseteq_o G[H[H']]) \\
= & \quad \{\text{lemma 4.15}\} \\
(\forall H, H' : H \text{ env } F \land H' \text{ env } F : H \text{ env } G \land H' \text{ env } G \land F[H[H'] \sqsubseteq_o G[H[H']]) \\
= & \quad \{\text{predicate calculus}\} \\
(\forall H : H \text{ env } F : (\forall H' : H' \text{ env } F : H \text{ env } G) \\
\land (\forall H' : H' \text{ env } F : H' \text{ env } G \land F[H[H'] \sqsubseteq_o G[H[H']]) \\
\Rightarrow & \quad \{\exists H' :: H' \text{ env } F\}, \text{ predicate calculus} \\
(\forall H : H \text{ env } F : H \text{ env } G \\
\land (\forall H' : H' \text{ env } F \land H' \text{ env } H : H' \text{ env } G \land F[H[H'] \sqsubseteq_o G[H[H']]) \\
\}
\end{align*}
\]
\[
\forall H : H \text{ env } F : \quad H \text{ env } G \\
\land (\forall H' : H' \text{ env } F H : H' \text{ env } G H \land F[H]H' \subseteq_o G[H]H')
\]

\[
\forall H : H \text{ env } F : H \text{ env } G \land F[H] \subseteq G[H]
\]

4.3 The ImpUNITY logic

In this section we introduce the ImpUNITY logic, a logic for ImpUNITY programs. We follow the approach of Sanders, but our main concern is to preserve the compositionality of the properties. We start from the basic properties in the Chandy and Misra logic and take “invariants” into account. However, instead of using invariants of the program itself, we use invariants of programs in an environment. Then, we define a new property to reason about progress. Like $\Rightarrow$ and until, the property is defined as a closure of ensures, but by using some extra conditions on the closure a compositional property is obtained.

The first step in defining the ImpUNITY logic is the use of local invariants. A local invariant of a program is an invariant that cannot be falsified by an environment. For an ImpUNITY program $F$, we model the restriction on interference of an environment by the modifier $\text{inter}(F)$. For the moment, the external-section is the only section concerning interference and $\text{inter}(F) \overset{\text{def}}{=} \text{external}(F)$. In chapter 7, where we introduce procedures, we extend $\text{inter}(F)$. To distinguish properties in the ImpUNITY logic from the other properties, these properties are subscripted by $\ast$.

**Definition 4.19** For ImpUNITY program $F$ and predicates $p$ and $q$, properties of $F$ are defined by

\[
\begin{align*}
\text{linvariant}_\ast r \text{ in } F & \overset{\text{def}}{=} [r \Rightarrow \text{inter}(F)(r)] \land \text{invariant}_{CM} r \text{ in } F, \\
p \text{ unless}_\ast q \text{ in } F & \overset{\text{def}}{=} (\exists r : \text{linvariant}_\ast r \text{ in } F : (p \land r) \text{ unless}_{CM} q \text{ in } F), \\
p \text{ ensures}_\ast q \text{ in } F & \overset{\text{def}}{=} (\exists r : \text{linvariant}_\ast r \text{ in } F : (p \land r) \text{ ensures}_{CM} q \text{ in } F), \\
p \rightarrow_\ast q \text{ in } F & \overset{\text{def}}{=} (\exists r : \text{linvariant}_\ast r \text{ in } F : (p \land r) \Rightarrow_{CM} q \text{ in } F).
\end{align*}
\]

As in the UNITY logics, these properties can be used to define additional properties in the standard way:

\[
\begin{align*}
\text{stable}_\ast p & \overset{\text{def}}{=} p \text{ unless}_\ast \text{false}, \\
\text{invariant}_\ast p & \overset{\text{def}}{=} ([\text{init}(F) \Rightarrow p] \land \text{stable}_\ast p), \\
p \text{ until}_\ast q & \overset{\text{def}}{=} (p \text{ unless}_\ast q \land p \Rightarrow_\ast q).
\end{align*}
\]
4.3. The ImpUNITY logic

Definition 4.19 is a generalisation of the UNITY logic. For programs that allow free interference, then, the ImpUNITY properties coincide with the properties in the standard UNITY logic of Chandy and Misra (see lemma 4.30). For closed programs, for which an environment cannot interfere, i.e., programs with \( \text{inter}(F) = \text{skip} \), all invariants of the program can be taken into account and the ImpUNITY properties specialise to the properties of the logic of Sanders (see lemma 4.31).

The properties defined above are UNITY-like properties in the sense that they can be used in a similar way as the standard UNITY properties. All theorems derived in [CM88] for properties of a single program also hold for the ImpUNITY properties. Furthermore, we can derive a kind of substitution theorem for local invariants.

**Theorem 4.20** Let \( F \) be an ImpUNITY program and let \( r \) be a predicate such that \( \text{linvariant}_r \text{ in } F \). If for predicates \( p, p', q \) and \( q' \) both \( [r \Rightarrow (p = p')] \) and \( [r \Rightarrow (q = q')] \) hold, then

\[
\begin{align*}
    p \text{ unless } q \text{ in } F & = p' \text{ unless } q' \text{ in } F, \\
    p \text{ ensures } q \text{ in } F & = p' \text{ ensures } q' \text{ in } F, \\
    p \mapsto q \text{ in } F & = p' \mapsto q' \text{ in } F.
\end{align*}
\]

The following lemma expresses that a local invariant of a program cannot be falsified by an environment.

**Lemma 4.21** Let \( F \) and \( H \) be ImpUNITY programs. If \( H \text{ env } F \), then

\[
\begin{align*}
    \text{stable}_{CM} r \text{ in } H & \iff \text{linvariant}_r \text{ in } F, \\
    \text{linvariant}_r \text{ in } F|H & \iff \text{linvariant}_r \text{ in } F.
\end{align*}
\]

**Proof:** First,

\[
\begin{align*}
    \text{linvariant}_r \text{ in } F & \\
    \Rightarrow & \quad \{ \text{definition} \} \\
    [r \Rightarrow \text{inter}(F)(r)] & \\
    \Rightarrow & \quad \{ \text{H env F} \} \\
    \langle \forall S : S \subseteq \text{assign}(H) : [r \Rightarrow S(r)] \rangle & \\
    = & \quad \{ \text{definitions} \} \\
    \text{stable}_{CM} r \text{ in } H
\end{align*}
\]
Second,

\[\text{linvariant}_s r \text{ in } F\]

\[\Rightarrow \quad \{\text{definitions and above}\}\]

\[r \Rightarrow \text{inter}(F)(r) \land [\text{init}(F) \Rightarrow r] \land \text{stable}_{CM} r \text{ in } F \land \text{stable}_{CM} r \text{ in } H\]

\[\Rightarrow \quad \{\text{union, theorem 2.9}\}\]

\[r \Rightarrow \text{inter}(F)(r) \land [\text{init}(F | H) \Rightarrow r] \land \text{stable}_{CM} r \text{ in } F | H\]

\[\Rightarrow \quad \{\text{external}(F) \leq \text{external}(F) + \text{external}(H)\}\]

\[r \Rightarrow \text{inter}(F | H)(r) \land [\text{init}(F | H) \Rightarrow r] \land \text{stable}_{CM} r \text{ in } F | H\]

\[= \quad \{\text{definitions}\}\]

\[\text{linvariant}_s r \text{ in } F | H\]

\[\square\]

ImpUNITY properties are based on local invariants. So, as a consequence of lemma 4.21, ImpUNITY properties are \textit{compositional} in the sense that properties of a union can be split into properties of its components.

**Lemma 4.22** Let \(F\) and \(H\) be ImpUNITY programs. If \(H\) env \(F\), then

\[p \text{ unless}_s q \text{ in } F | H \quad \iff \quad p \text{ unless}_s q \text{ in } F \land p \text{ unless}_{CM} q \text{ in } H,\]

\[p \text{ ensures}_s q \text{ in } F | H \quad \iff \quad p \text{ ensures}_s q \text{ in } F \land p \text{ unless}_{CM} q \text{ in } H.\]

**Proof:**

\[p \text{ unless}_s q \text{ in } F \land p \text{ unless}_{CM} q \text{ in } H\]

\[= \quad \{\text{definitions}\}\]

\[\langle \exists r : \text{linvariant}_s r \text{ in } F : (p \land r) \text{ unless}_{CM} q \text{ in } F \rangle \land p \text{ unless}_{CM} q \text{ in } H\]

\[= \quad \{\text{predicate calculus}\}\]

\[\langle \exists r : \text{linvariant}_s r \text{ in } F : (p \land r) \text{ unless}_{CM} q \text{ in } F \land p \text{ unless}_{CM} q \text{ in } H\rangle\]

\[\Rightarrow \quad \{\text{lemma 4.21, conjunction (A.3) and consequence weakening (A.2)}\}\]

\[\langle \exists r : \text{linvariant}_s r \text{ in } F | H : (p \land r) \text{ unless}_{CM} q \text{ in } F \land (p \land r) \text{ unless}_{CM} q \text{ in } H\rangle\]

\[\Rightarrow \quad \{\text{union, theorem 2.9}\}\]

\[\langle \exists r : \text{linvariant}_s r \text{ in } F | H : (p \land r) \text{ unless}_{CM} q \text{ in } F | H\rangle\]

\[= \quad \{\text{definition}\}\]

\[p \text{ unless}_s q \text{ in } F | H\]
4.3. The ImpUNITY logic

The proof for \textit{ensures} is similar. \hfill\Box

In a similar way we can prove that ImpUNITY properties are preserved in contexts.

\textbf{Lemma 4.23} Let $F$ and $H$ be ImpUNITY programs. If $H$ cont $F$, then

\begin{align*}
p \text{ unless}_s q \text{ in } F|H & \iff p \text{ unless}_s q \text{ in } F \land p \text{ unless}_s q \text{ in } H, \\
p \text{ ensures}_s q \text{ in } F|H & \iff p \text{ ensures}_s q \text{ in } F \land p \text{ unless}_s q \text{ in } H.
\end{align*}

\textbf{Example 4.2.}

To get a feeling for the ImpUNITY properties, consider program $F$ as given in example 4.2. For this program we check some properties. Program $F$ can set variable $x$ to $-1$ (by the second statement) and to $1$ (by the last statement). So, in the standard UNITY logic, which assumes that the environment can interfere freely, neither $\text{stable}_{CM} x \geq 0 \text{ in } F$ nor $\text{stable}_{CM} x \leq 0 \text{ in } F$ hold. When $F$ runs in isolation it does not change the value of $x$ as expressed by property $\text{invariant}_{CM} y \geq 0 \land x = 0 \text{ in } F$. Using this invariant $\text{stable}_S x \geq 0 \text{ in } F$ and $\text{stable}_S x \leq 0 \text{ in } F$ can be derived. Program $F$ does not have to run in isolation, nor does it have to run in an arbitrary environment. The external-section of $F$ requires that an environment does not change the value of variable $y$. This restriction implies that in any execution of $F$ (in any environment) the value of $y$ will always be at least zero. Property $\text{invariant}_S y \geq 0 \text{ in } F$ holds and $F$ cannot set $x$ to $-1$, so, $\text{stable}_S x \geq 0 \text{ in } F$. However, an environment is allowed to change the value of $x$. Then, program $F$ may change $x$ from some negative value to $1$, and consequently, $\text{stable}_S x \leq 0 \text{ in } F$ does not hold.

The second step in defining the ImpUNITY logic is to define a new progress property. For closed systems, properties $\rightarrow_S$ and \textit{until}_S in Sanders’s logic have nice temporal interpretations as expressed by lemma 3.18. Property $\textit{ensures}_S$ is only used for defining $\rightarrow_S$. In the standard UNITY logic, property $\textit{ensures}_{CM}$ is more important: it can be used to reason about progress of programs in a compositional way using the union theorem 2.9. For ImpUNITY programs we define a new progress property by incorporating
the restriction on interference of the external-section. This property is called internal until and is denoted by \( \rightarrow_s \). It is a property between \( \text{ensures}_s \) and \( \text{until}_s \) that can be used to reason about progress of components. We define the internal until property in two steps. First, we define property \( \rightarrow_{CM} \). Like the \( \text{until}_{CM} \) property, \( \rightarrow_{CM} \) is a closure of \( \text{ensures}_{CM} \) properties. However, by being more restrictive in taking the closure, a compositional progress property is obtained. Then, we define the \( \rightarrow_s \) property by taking local invariants into account. The \( \rightarrow_s \) property can be used as an alternative for \( \text{ensures}_s \) in the sense that it can serve as a base for the definition of the leadsto property and all the standard UNITY theorems of \( \text{ensures}_{CM} \) properties [CM88] also hold for \( \rightarrow_s \).

First, we give the formal definitions of the \( \rightarrow_s \) properties and prove that the properties are compositional. Then we give the intuition behind the closure conditions used in the definition.

**Definition 4.24** Let \( F \) be an ImpUNITY program. Property \( \rightarrow_{CM} \) is the smallest relation \( \text{Prop} \) that satisfies the following conditions:

1. If \( p \text{ ensures}_{CM} q \) in \( F \), then \( p \text{ Prop } q \) in \( F \).
2. If \( p \text{ Prop } (r \lor q) \) in \( F \), \( r \text{ Prop } q \) in \( F \), and \( [(r \land \neg q) \Rightarrow \text{inter}(F)(\neg p \lor r \lor q)] \), then \( (p \lor r) \text{ Prop } q \) in \( F \).
3. If for any set \( W \) \( (\forall w : w \in W : p_w \text{ Prop}_{CM} q \text{ in } F) \), and
   \[
   \langle \forall w : w \in W : [(p_w \land \neg q) \Rightarrow \text{inter}(F)(\langle \forall i : i \in W : \neg p_i \rangle \lor p_w \lor q)] \rangle,
   \]
then \( (\exists w : w \in W : p_w) \text{ Prop } q \) in \( F \).

Then, property \( \rightarrow_s \) is defined by

\[
p \rightarrow_s q \text{ in } F \overset{\text{def}}{=} (\exists r : \text{linvariant}_s r \text{ in } F : (p \land r) \rightarrow_{CM} q \text{ in } F).
\]

The extra conditions in the second and third item are posed in order to obtain a compositional progress property. In fact, the definition is constructed such that the following lemma holds (compare with lemma 4.22).

**Lemma 4.25** Let \( F \) and \( H \) be ImpUNITY programs. If \( H \text{ env } F \), then

\[
p \rightarrow_{CM} q \text{ in } F|H \iff p \rightarrow_{CM} q \text{ in } F \land p \text{ unless}_{CM} q \text{ in } H,
\]

\[
p \rightarrow_s q \text{ in } F|H \iff p \rightarrow_s q \text{ in } F \land p \text{ unless}_s q \text{ in } H.
\]

If \( H \text{ cont } F \), then

\[
p \rightarrow_s q \text{ in } F|H \iff p \rightarrow_s q \text{ in } F \land p \text{ unless}_s q \text{ in } H.
\]
4.3. The ImpUNITY logic

**Proof:** Here we only prove the compositionality of $\rightarrow_{CM}$. We prove this lemma using induction on the proof of $p \rightarrow_{CM} q$ in $F$.

**Base:** The base case of the induction follows directly from the compositionality of the $\textit{unless}_{CM}$ and $\textit{ensures}_{CM}$ properties of theorem 2.9.

**Trans:** Suppose $p \rightarrow_{CM} q$ in $F$ has been proven by transitivity. Then, there exist two state predicates $p'$ and $r$ such that $[p = (p' \lor r)]$, $p' \rightarrow_{CM} (r \lor q)$ in $F$, $r \rightarrow_{CM} q$ in $F$, and $[(r \land \neg q) \Rightarrow \text{inter}(F)(\neg p' \lor r \lor q)]$ hold. We have to prove $(p' \lor r) \rightarrow_{CM} q$ in $F\mid H$.

From property $p$ $\textit{unless}_{CM} q$ in $H$, we can derive two safety properties of $H$. First, $p' \textit{unless}_{CM} (r \lor q)$ in $H$ follows directly from $[p = (p' \lor r)]$ and from theorems for $\textit{unless}$ (A.4, A.9). Second, $r$ $\textit{unless}_{CM} q$ in $H$ follows from the condition of transitivity:

$$[(r \land \neg q) \Rightarrow \text{inter}(F)(\neg p' \lor r \lor q)]$$

$$\Rightarrow \quad \text{predicate calculus}$$

$$[(r \land \neg q \land (p' \lor r)) \Rightarrow \text{inter}(F)(\neg (p' \lor r) \lor r \lor q)]$$

$$\Rightarrow \quad \{H \text{ obey } \text{inter}(F)\}$$

$$\forall S : S \in \text{assign}(H) : [(r \land \neg q \land (p' \lor r)) \Rightarrow S(\neg (p' \lor r) \lor r \lor q)]$$

$$= \quad \{\text{definition}\}$$

$$r \textit{unless}_{CM} (p' \lor r) \lor q \text{ in } H$$

$$\Rightarrow \quad \{\text{conjunction (A.3) with } p' \lor r \textit{unless}_{CM} q \text{ in } H\}$$

$$r \textit{unless}_{CM} q \text{ in } H$$

By the induction hypothesis we conclude that both $p' \rightarrow_{CM} (q \lor r)$ in $F\mid H$ and $r \rightarrow_{CM} q$ in $F\mid H$. Then, $(p' \lor r) \rightarrow_{CM} q$ in $F\mid H$ follows from the transitivity of $\rightarrow_{CM}$ using that $\text{inter}(F) \leq \text{inter}(F\mid H)$.

**Disj:** Suppose $p \rightarrow_{CM} q$ in $F$ has been proven by applying distributivity. Then, there exists a set of state predicates $\{p_w : w \in W\}$ such that $[p = (\exists w : w \in W : p_w)]$, $p_w \rightarrow_{CM} q$ in $F$, and $[(p_w \land \neg q) \Rightarrow \text{inter}(F)((\forall i : i \in W : \neg p_i) \lor p_w \lor q)]$ hold. We have to prove $(\exists w : w \in W : p_w) \rightarrow_{CM} q$ in $F\mid H$. From $p$ $\textit{unless}_{CM} q$ in $H$, which is equivalent to $(\exists w : w \in W : p_w)$ $\textit{unless}_{CM} q$ in $H$, we derive a set of safety properties of $H$ using the condition of disjunctivity:

$$[(p_w \land \neg q) \Rightarrow \text{inter}(F)((\forall i : i \in W : \neg p_i) \lor p_w \lor q)]$$

$$\Rightarrow \quad \text{predicate calculus}$$

$$[(p_w \land \neg q \land (\exists i : i \in W : p_i)) \Rightarrow \text{inter}(F)((\forall i : i \in W : \neg p_i) \lor p_w \lor q)]$$

$$\Rightarrow \quad \{H \text{ obey } \text{inter}(F)\}$$
This concludes the proof of the first item of the lemma. The proofs of the second and third item are similar to the proofs of lemma 4.22 and 4.23. 

Property $\rightarrow_*$ is defined as the transitive and disjunctive closure of $\text{ensures}_*$. Both closure steps contain a condition in terms of the possible external interference by an environment. These conditions are posed to preserve compositionality of the property. Next, we give the intuition behind the definition and the role of the conditions in the definition.

Transitivity is used to express that a program takes care of progress in a sequence of steps. For example, from states in $p$, a state in $q$ is reached via intermediate states in $r$. The condition for transitivity is $[(r \land \neg q) \rightarrow \text{inter}(F)(\neg p \lor r \lor q)]$ and expresses that $r \land \neg q$ cannot be falsified by an environment without establishing $\neg(p \lor r) \lor q$. This implies the following property of the environment $H$: $r \text{ unless } (p \lor r) \lor q \text{ in } H$. In other words, as soon as the intermediate state in $r$ is reached, an environment cannot go back to $p$, so the environment cannot disturb the progress expressed by $\rightarrow_*$. This is illustrated by program $F$ in example 4.3. Program $F$ sets $x$ to 0 in two steps. The first statement sets $b$ to $true$ and then the second statement sets $x$ to 0. An environment could prevent $F$ from setting $x$ to 0 by resetting $b$ to $false$. However, this is forbidden by the external-section of $F$. We can derive: $\neg b \text{ ensures}_*(b \lor x = 0) \text{ in } F$, $b \text{ ensures}_* x = 0 \text{ in } F$ and $[(b \land x \neq 0) \Rightarrow [=_{F}](b \lor x = 0)]$. Then, $true \rightarrow_* x = 0 \text{ in } F$.

Disjunctivity is about the case that a number of statements is responsible for some progress. For example, one statement makes the step from $p_1$ to $q$ while another makes the transition from $p_2$ to $q$. Together they are responsible for the progress from $(p_1 \lor p_2)$ to $q$. The condition for disjunctivity is

$$\langle \forall w : w \in W : [p_w \land \neg q \Rightarrow \text{inter}(F)(\langle \forall i : i \in W : \neg p_i \lor p_w \lor q])] \rangle,$$

and expresses that $p_w \land \neg q$ cannot be falsified by an environment without establishing $\langle \forall i : i \in W : \neg p_i \rangle \lor q$. This implies the following properties of the environment $H$:
4.3. The ImpUNITY logic

$p_w$ unless $(\forall i : i \in W : \neg p_i) \lor p_w \lor q$ in $H$ for any $w \in W$. This reflects the idea that an environment cannot disturb the progress by flipping between predicates $p_w$. This is illustrated by program $G$ in example 4.3. Since the environment cannot change the value of $b$, it cannot prevent $G$ from setting $x$ to 0, so, true $\rightarrow_* x = 0$ in $G$.

<table>
<thead>
<tr>
<th>Program $F$</th>
<th>Program $G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>external $[={b}]$</td>
<td>external $[={b}]$</td>
</tr>
<tr>
<td>assign $b := true$</td>
<td>assign $b \rightarrow b, x := false, 0$</td>
</tr>
<tr>
<td></td>
<td>$\neg b \rightarrow b, x := true, 0$</td>
</tr>
<tr>
<td>end{$F$}</td>
<td>end{$G$}</td>
</tr>
</tbody>
</table>

Example 4.3.

Now that we have some intuition about the $\rightarrow_*$ property, we compare the property with other properties. First, the $\rightarrow_*$ property is a property between ensures and $\rightarrow_*$ and it implies unless.

Lemma 4.26 For ImpUNITY program $F$,

\[
\begin{align*}
p \text{ ensures}_* q \text{ in } F & \Rightarrow p \rightarrow_* q \text{ in } F, \\
p \rightarrow_* q \text{ in } F & \Rightarrow p \rightarrow_* q \text{ in } F, \\
p \rightarrow_* q \text{ in } F & \Rightarrow p \text{ unless}_* q \text{ in } F.
\end{align*}
\]

Since $\rightarrow_*$ is a property between ensures and $\rightarrow_*$, it can serve as a base for defining $\rightarrow_*$.
All $\rightarrow_*$ properties can be derived from $\rightarrow_*$ properties in the same way as they can be derived from the ensures properties.

Lemma 4.27 For ImpUNITY program $F$, property $\rightarrow_*$ is the smallest relation Prop that satisfies the following conditions:

1. If $p \rightarrow_* q$, then $p \text{ Prop } q$.

2. If $p \text{ Prop } r$ and $r \text{ Prop } q$, then $p \text{ Prop } q$.

3. If for any set $W$ $(\forall w : w \in W : p_w \text{ Prop } q)$, then $(\exists w : w \in W : p_w) \text{ Prop } q$.

Furthermore, strengthening the interference restriction preserves the ImpUNITY properties. This is expressed by the following lemma.

Lemma 4.28 For ImpUNITY programs $F$ and $G$, that only differ in their external-sections and that satisfy $\text{external}(G) \leq \text{external}(F)$,
\[ p \text{ unless}_* q \text{ in } F \Rightarrow p \text{ unless}_* q \text{ in } G, \]
\[ p \Rightarrow_* q \text{ in } F \Rightarrow p \Rightarrow_* q \text{ in } G. \]

From the way the ImpUNITY logic is constructed, it is easy to see that the ImpUNITY properties are between the properties in the logic of Chandy and Misra and the properties in the modified logic of Sanders.

**Lemma 4.29** For ImpUNITY program \( F \),

\[ p \text{ unless}_* CM q \text{ in } F \Rightarrow p \text{ unless}_* q \text{ in } F, \]
\[ p \text{ ensures}_* CM q \text{ in } F \Rightarrow p \Rightarrow_* q \text{ in } F, \]
\[ p \Rightarrow CM q \text{ in } F \Rightarrow p \Rightarrow_* q \text{ in } F, \]

and

\[ p \text{ unless}_* q \text{ in } F \Rightarrow p \text{ unless}_S q \text{ in } F, \]
\[ p \Rightarrow_* q \text{ in } F \Rightarrow p \text{ until}_S q \text{ in } F, \]
\[ p \Rightarrow_* q \text{ in } F \Rightarrow p \Rightarrow q \text{ in } F. \]

In the extreme case, the implications turn out to be equivalences, meaning that we do not get unexpected results. This is expressed in the following two lemmas. For a program \( F \) that allows the environment to interfere freely, the program corresponds to a program in the standard UNITY framework. This means that no invariants are taken into account and that the \( \text{unless}_* \) and \( \Rightarrow_* \) reduce to their counterparts in the logic of Chandy and Misra. For the \( \Rightarrow_* \) property the condition for transitivity reduces to \([r \Rightarrow q] \lor [(p \land \neg r) \Rightarrow q] \), and the condition for disjunctivity, if we take \( W = \{1, 2\} \), reduces to \([p_1 \Rightarrow q] \lor [p_2 \Rightarrow q] \). These conditions imply that no real closure is taken and that \( \Rightarrow_* \) reduces to the \( \text{ensures}_CM \) property.

**Lemma 4.30** For ImpUNITY program \( F \) with \( \text{inter}(F) = [\times_U] \),

\[ p \text{ unless}_* q \text{ in } F = p \text{ unless}_CM q \text{ in } F, \]
\[ p \text{ ensures}_* q \text{ in } F = p \text{ ensures}_CM q \text{ in } F, \]
\[ p \Rightarrow_* q \text{ in } F = p \Rightarrow CM q \text{ in } F, \]
\[ p \Rightarrow_* q \text{ in } F = p \text{ ensures}_CM q \text{ in } F. \]

**Proof:** The first three items are of the lemma are straightforward to prove. Here we focus on the last item. The implication \( p \Rightarrow_* q \iff p \text{ ensures}_* q \) holds by definition of \( \Rightarrow_* \). The other implication is proven by induction on the proof of \( p \Rightarrow_* q \).

**Base:** Follows directly from the observation above.

**Trans:** Suppose \( p \Rightarrow_* q \) has been proven by transitivity using \([p = (p' \lor r)]\), \( p' \Rightarrow_* (r \lor q) \text{ in } F \), \( r \Rightarrow_* q \text{ in } F \), and \([(r \land \neg q) \Rightarrow \text{inter}(F)(\neg p' \lor r \lor q)] \). Since \( \text{inter}(F) = [\times_U] \), the condition for transitivity is \([r \Rightarrow q] \lor [(\neg r \land p') \Rightarrow q] \). Then,
\[
p' \Rightarrow \gamma_s (r \land q) \land r \Rightarrow \gamma_q
\]
\[\Rightarrow \{\text{induction hypothesis}\}
\]
\[
p' \text{ ensures}_{\text{CM}} (r \land q) \land r \text{ ensures}_{\text{CM}} q
\]
\[\Rightarrow \{\text{disjunction (A.16) with } r \text{ and } \neg r \land p'\}
\]
\[
(p' \lor r) \text{ ensures}_{\text{CM}} (r \land q) \land (p' \lor r) \text{ ensures}_{\text{CM}} (q \lor (\neg r \land p'))
\]
\[\Rightarrow \{\text{consequence weakening (A.13) using } [r \Rightarrow q] \text{ resp. } [(\neg r \land p') \Rightarrow q]\}
\]
\[
(p' \lor r) \text{ ensures}_{\text{CM}} q
\]

Disj: Suppose \( p \Rightarrow \gamma q \) has been proven by disjunctivity using \([p = (\exists w : w \in W : p_w)], p_w \Rightarrow \gamma q \) in \( F \), and \([(p_w \land \neg q) \Rightarrow \text{inter}(F)(\forall i : i \in W : \neg p_i) \lor p_w \land q)] \). The condition for disjunctivity for \( \text{inter}(F) = [\text{skip}] \) reduces to

\[
(\forall w : w \in W : [p_w \Rightarrow q] \lor [\neg q \Rightarrow (\exists i : i \in W : p_w) = p_w]).
\]

Then,

\[
(\forall w : w \in W : p_w \Rightarrow \gamma q)
\]
\[\Rightarrow \{\text{induction hypothesis}\}
\]
\[
(\forall w : w \in W : p_w \text{ ensures}_{\text{CM}} q)
\]
\[\Rightarrow \{\text{predicate calculus}\}
\]
\[
(\forall w : w \in W : [p_w \Rightarrow q] \lor p_w \text{ ensures}_{\text{CM}} q)
\]
\[
= \{\text{A.20 using condition disjunctivity}\}
\]
\[
(\forall w : w \in W : [p_w \Rightarrow q] \lor (\exists w : w \in W : p_w) \text{ ensures}_{\text{CM}} q)
\]
\[= \{\text{predicate calculus}\}
\]
\[
(\forall w : w \in W : [p_w \Rightarrow q]) \lor (\exists w : w \in W : p_w) \text{ ensures}_{\text{CM}} q
\]
\[
= \{\text{predicate calculus}\}
\]
\[
[(\exists w : w \in W : p_w) \Rightarrow q] \lor (\exists w : w \in W : p_w) \text{ ensures}_{\text{CM}} q
\]
\[\Rightarrow \{\text{implication (A.17)}\}
\]
\[
(\exists w : w \in W : p_w) \text{ ensures}_{\text{CM}} q
\]
\[
\square
\]

For a program \( F \) with \( \text{inter}(F) = \text{skip} \), an environment cannot interfere and \( F \) can be seen as a closed system. In this case, the closure conditions in the definition of \( \Rightarrow_{\text{CM}} \) are always satisfied and definition 4.24 coincides with the characterisation of the \text{until}_{\text{CM}} property as given in theorem 2.8. Furthermore, every invariant is a local invariant, so definition 4.19 corresponds to definition 2.10. This gives the following lemma.
Lemma 4.31  For ImpUNITY program $F$ with $\text{inter}(F) = \text{skip}$,

\[
\begin{align*}
    p \text{ unless}_* q \text{ in } F &= p \text{ unless}_S q \text{ in } F, \\
    p \text{ ensures}_* q \text{ in } F &= p \text{ ensures}_S q \text{ in } F, \\
    p \Rightarrow_S^* q \text{ in } F &= p \Rightarrow_S q \text{ in } F, \\
    p \Rightarrow^* q \text{ in } F &= p \text{ until}_S q \text{ in } F.
\end{align*}
\]

For standard UNITY programs we have shown that preservation of $\text{unless}_{CM}$ and $\text{ensures}_{CM}$ properties is the best compositional notion of refinement (see theorem 3.15). Since environments are restricted, this notion is too strong for ImpUNITY programs. Properties introduced in this section are weaker, and the following theorem expresses that it is sufficient that each $\text{unless}_{CM}$ property implies an $\text{unless}_*$ property in the refined program, and that each $\text{ensures}_{CM}$ property implies an $\Rightarrow_*$ property.

Theorem 4.32  Let $F$ and $G$ be ImpUNITY programs. If for all predicates $p$ and $q$,

\[
\begin{align*}
    \text{init}(F) &\subseteq \text{init}(G), \\
    p \text{ unless}_{CM} q \text{ in } F &\Rightarrow p \text{ unless}_* q \text{ in } G, \\
    p \text{ ensures}_{CM} q \text{ in } F &\Rightarrow p \Rightarrow_* q \text{ in } G,
\end{align*}
\]

then $F \subseteq G$.

The following lemma expresses that preservation of properties is equivalent to the preservation of $\text{unless}_*$ and $\Rightarrow_*$ properties.

Lemma 4.33  For ImpUNITY programs $F$ and $G$, such that $[\text{init}(F) \subseteq \text{init}(G)]$, 

\[
\begin{align*}
    \forall p, q :: p \text{ unless}_{CM} q \text{ in } F &\Rightarrow p \text{ unless}_* q \text{ in } G \\
    p \text{ ensures}_{CM} q \text{ in } F &\Rightarrow p \Rightarrow_* q \text{ in } G
\end{align*}
\]

Proof:  The implication from right to left follows directly from the fact that properties in the Chandy and Misra logic are stronger than the corresponding properties in the ImpUNITY logic as expressed in lemma 4.29. The other implication follows from the fact that preservation of the $\text{unless}_{CM}$ implies preservation of local invariants. For the $\Rightarrow_*$ property a standard induction proof can be used. \hfill \Box

A direct consequence is that preservation of $\text{unless}_*$ and $\Rightarrow_*$ results in refinement.
Corollary 4.34 Let $F$ and $G$ be ImpUNITY programs. If for all predicates $p$ and $q$,

\[
\begin{align*}
\text{init}(F) & \subseteq \text{init}(G), \\
p \text{ unless}_s q \text{ in } F & \Rightarrow p \text{ unless}_s q \text{ in } G, \\
p \rightarrow_s q \text{ in } F & \Rightarrow p \rightarrow_s q \text{ in } G,
\end{align*}
\]

then $F \subseteq G$.

This corollary suggests to use properties $\text{unless}_s$ and $\rightarrow_s$ to specify components.

4.4 Program Transformation Rules

In this section we give five program transformation rules. The first rule allows the assign-section of an ImpUNITY program to be extended with a new statement if that statement is a refinement of the statements in the assign-section.

Transformation 4.35 (Add statement) Let $F$ and $G$ be ImpUNITY programs that only differ in their assign-sections, and let $T$ be a statement. If

- $(\bigwedge_{S \subseteq \text{assign}(F)} S \land \text{skip}) \leq T$, and
- $\text{assign}(G) = \text{assign}(F) \cup \{T\},$

then $F \subseteq G$.

The second rule is about the combination of statements. Statements that differ only in their guards can be combined into one statement that is enabled if one of its components is enabled.

Transformation 4.36 (Combine statements) Let $F$ and $G$ be ImpUNITY programs that only differ in their assign-sections. Let $U = \{p_i \rightarrow S \mid i \in I\}$ be a subset of $\text{assign}(F)$, and let $T \overset{\text{def}}{=} (\exists i : i \in I : p_i) \rightarrow S$. If

\[
\text{assign}(G) = \text{assign}(F) \setminus U \cup \{T\},
\]

then $F \subseteq G$.

Proof: It is straightforward to prove that the initially-section and the $\text{unless}_{CM}$ properties are preserved. Furthermore,

\[
\begin{align*}
[(p \land \neg q) \Rightarrow ((\exists i : i \in I : p_i) \rightarrow S)(q)] \\
= \{\text{definition}\} \\
[(p \land \neg q \land (\exists i : i \in I : p_i) \Rightarrow S(q)) \land (p \land \neg q \land \neg(\exists i : i \in I : p_i) \Rightarrow q)]
\end{align*}
\]
\[
\{\text{predicate calculus}\}
\]
\[
[(p \land \neg q \land (\exists i : i \in I : p_i) \Rightarrow S(q)) \land (p \land \neg q \Rightarrow (\exists i : i \in I : p_i))]
\]
\[
\{\text{predicate calculus}\}
\]
\[
[(p \land \neg q \Rightarrow S(q)) \land (p \land \neg q \Rightarrow (\exists i : i \in I : p_i))]
\]

\[
[\exists \forall i : i \in I : \{r; S_i \leq S'_i,\} \Rightarrow \forall i : i \in I : [a \land r \land b \Rightarrow S_i((a \land r) \lor b)]]
\]

\[
[\forall i : i \in I : [a \land r \land b \Rightarrow r \land S_i((a \land r) \lor b)]]
\]

So, all \(\text{ensures}_CM\) are preserved. Properties in the Chandy and Misra logic imply ImpUNITY properties (lemma 4.29), and hence, the refinement \(F \sqsubseteq G\) follows from theorem 4.32. □

The third transformation rule is about the use of local invariants. Each statement of the program may be replaced by a different statement as long as the other statement behaves the same if the local invariant holds. This rule is a kind of substitution rule on the level of programs.

**Transformation 4.37 (Local invariant)** Let \(F\) and \(G\) be ImpUNITY programs that only differ in their assign-sections, and let \(\text{assign}(F) = \{S_i : i \in I\}\) and let \(\text{assign}(G) = \{S'_i : i \in I\}\). If for some predicate \(r\)

- \(\text{linvariant}_r \leq F\), and

- \(\{r; \forall i : S_i \leq S'_i,\} \Rightarrow \forall i : S_i((a \land r) \lor b)\)

then \(F \sqsubseteq G\).

**Proof:** First,

\[a \text{ unless}_{CM} b \text{ in } F\]

\[\Rightarrow \{\text{conjunction (A.3) } r \text{ unless}_{CM} \text{ false in } F\} \]

\[(a \land r) \text{ unless}_{CM} b \text{ in } F\]

\[= \{\text{definition}\} \]

\[\forall i : i \in I : [a \land r \land b \Rightarrow S_i((a \land r) \lor b)]\]

\[= \{\text{predicate calculus}\} \]

\[\forall i : i \in I : [a \land r \land b \Rightarrow r \land S_i((a \land r) \lor b)]\]
By taking \( a = r \) and \( b = \text{false} \) and the fact that the \textit{initially} and \textit{external}-sections are not changed, property \( \text{linvariant}_s r \) in \( G \) holds. Furthermore,

\[
\text{a unless}_C M \text{ b in } F
\]

\[
\Rightarrow \quad \{ \text{above} \}
\]

\[
( a \land r ) \text{ unless}_C M \text{ b in } G
\]

\[
\Rightarrow \quad \{ \text{definition, linvariant}_s r \text{ in } G \}
\]

\[
a \text{ unless}_s \text{ b in } G
\]

Preservation of \textit{ensures} properties is shown in a similar way. Then, the refinement \( F \subseteq G \) follows from theorem 4.32.

---

### Example 4.4.

Programs \( F \) and \( G \) in example 4.4 illustrate the use of this program transformation rule. Program \( G \) is the same as program \( F \) except that the first statement is rewritten using that \( \text{linvariant}_s (y \geq 0) \) in \( F \).

The fourth rule is about the strengthening of a guard and is based on the transitivity of the \( \rightarrow_s \) property. Here, we denote by \( F \setminus S \) the program \( F \) from which statement \( S \) is removed from the \textit{assign}-section.
Transformation 4.38 (Strengthening guard) Let $F$ be an ImpUNITY program and let $S = (p ightarrow T)$ be a statement such that $S \in \text{assign}(F)$. Let $G$ be the ImpUNITY program that only differs from $F$ in the assign-section and
\[
\text{assign}(G) = \text{assign}(F) \setminus \{S\} \cup \{S'\},
\]
where $S' = (p \land q \rightarrow T)$. If
\begin{itemize}
  \item $[(q \land p) \Rightarrow \text{inter}(F)(q \lor \neg p)]$,
  \item $q \text{ unless}_{CM} \neg p \text{ in } (F \setminus S)$, and
  \item $p \rightarrow_{CM} (\neg p \lor q) \text{ in } G$,
\end{itemize}
then $F \subseteq G$.

Proof: We use theorem 4.32. We have to show that $\text{unless}_{CM}$ and $\text{ensures}_{CM}$ properties in $F$ imply $\text{unless}_{s}$ and $\rightarrow_{s}$ properties in $G$. It is straightforward to prove that strengthening a guard preserves safety properties. We focus on the progress properties and prove that each $\text{ensures}_{CM}$ property of $F$ implies the corresponding $\rightarrow_{s}$ property of $G$. For $\text{ensures}_{CM}$ properties that are not based on statement $S$, this is trivial. Suppose a $\text{ensures}_{CM}$ $b$ in $F$ holds and $[(a \land \neg b) \Rightarrow S(b)]$. Then, we can derive $[(a \land \neg b) \Rightarrow p]$ (so, $[(a \land \neg p) \Rightarrow b]$) and $[(a \land q \land \neg b) \Rightarrow S'(b)]$. Then,
\[
a \text{ unless}_{CM} b \text{ in } F
\]
\[
\Rightarrow \{ \text{preservation of safety properties, union} \}
\]
\[
a \text{ unless}_{CM} b \text{ in } G \land a \text{ unless}_{CM} b \text{ in } (F \setminus S)
\]
\[
\Rightarrow \{ \text{conjunction } p \rightarrow_{CM} (\neg p \lor q) \text{ in } G \text{ and } q \text{ unless}_{CM} \neg p \text{ in } (F \setminus S) \}
\]
\[
a \land p \rightarrow_{CM} (a \land (\neg p \lor q)) \lor (b \land p) \lor (b \land (\neg p \lor q)) \text{ in } G
\]
\[
\land a \land q \text{ unless}_{CM} (a \land \neg p) \lor (b \land q) \lor (b \land \neg p) \text{ in } (F \setminus S)
\]
\[
\Rightarrow \{ \text{A.21 } [\neg b \Rightarrow (a = a \land p)], \text{ consequence weakening } [(a \land \neg p) \Rightarrow b] \}
\]
\[
a \rightarrow_{CM} (a \land q) \lor b \text{ in } G \land (a \land q) \text{ unless}_{CM} b \text{ in } (F \setminus S)
\]
\[
\Rightarrow \{(a \land q \land \neg b) \Rightarrow S'(b) \}
\]
\[
a \rightarrow_{CM} ((a \land q) \lor b) \text{ in } G \land (a \land q) \text{ ensures}_{CM} b \text{ in } G
\]
\[
\Rightarrow \{ \text{transitivity, see below} \}
\]
\[
a \rightarrow_{s} b \text{ in } G
\]
\[
\Rightarrow \{ \text{lifting} \}
\]
\[
a \rightarrow_{s} b \text{ in } G
\]
In the derivation above, the transitivity of $\rightarrow_{CM}$ is used. We have to check the condition on transitivity. The condition follows from the assumption on predicate $q$:

$$[a \land q \land \neg b \Rightarrow \text{inter}(F)(\neg a \lor (a \land q) \lor b)]$$

$$= \{\text{predicate calculus}\}$$

$$[a \land q \land \neg b \Rightarrow \text{inter}(F)(\neg a \lor q \lor b)]$$

$$= \{(a \land \neg b) \Rightarrow p\}, \text{predicate calculus}$$

$$[q \land p \Rightarrow \text{inter}(F)(q \lor \neg p)]$$

\[ \square \]

**Program $F$**

- **external [$\leftarrow \{ y \}$]**
- **assign** $y \geq 0 \rightarrow x, y := 0, 10$
  - $y > 0 \rightarrow y := y - 1$
- **end** \{ $F$ \}

**Program $G$**

- **external [$\leftarrow \{ y \}$]**
- **assign** $y = 0 \rightarrow x, y := 0, 10$
  - $y > 0 \rightarrow y := y - 1$
- **end** \{ $G$ \}

**Example 4.5.**

Programs $F$ and $G$ in example 4.5 illustrate the use of this program transformation rule. Program $G$ is the same as program $F$ except that the guard of the first statement of $F$ has been strengthened. The fact that $G$ is a refinement of $F$ can be proven with the rule above taking $p = (y \geq 0)$ and $q = (y = 0)$.

The last transformation rule in this section concerns the splitting of a statement.

**Transformation 4.39 (Split statement)** Let $F$ be an ImpUNITY program and let $S = (p \rightarrow T)$ be a statement such that $S \in \text{assign}(F)$. Let \{ $q_i \mid i \in I$ \} be a set of predicates and let $G$ be the ImpUNITY program that only differs from $F$ in the assign-section and

$$\text{assign}(G) = \text{assign}(F) \setminus \{ S \} \cup \{ S_i \mid i \in I \},$$

where $S_i = (p \land q_i \rightarrow T)$ for $i \in I$. If

- $[p \Rightarrow (\exists i : i \in I : q_i)]$,
- $[p \land q_i \Rightarrow \text{inter}(F)(\neg p \lor q_i)]$, for all $i \in I$, and
- $q_i$ unless $\text{CM} \neg p$ in $(G \setminus S_i)$, for all $i \in I$,

then $F \equiv G$. 
Proof: We use theorem 4.32. We have to show that unless\(_{CM}\) properties and ensures\(_{CM}\) properties in \(F\) imply unless\(_{\ast}\) and \(\rightarrow_{\ast}\) properties in \(G\). It is straightforward to prove that splitting a statement preserves all safety properties. We focus on the progress properties and prove that each ensures\(_{CM}\) property of \(F\) implies the corresponding \(\rightarrow_{\ast}\) property of \(G\). For ensures\(_{CM}\) properties that are not based on statement \(S\), this is trivial.

Suppose \(a\) ensures\(_{CM}\) \(b\) in \(F\) and \([(a \land \neg b) \Rightarrow S(b)]\). Then, we can derive \([(a \land \neg p) \Rightarrow p]\) (so, \([(a \land \neg p) \Rightarrow b]\)) and \([(a \land q_{i} \land \neg b) \Rightarrow S_{i}(b)]\). Then,

\[
a \operatorname{unless}_{CM} b \text{ in } F
\]
\[
\Rightarrow \{\text{preservation of safety properties}\}
\]
\[
a \operatorname{unless}_{CM} b \text{ in } G
\]
\[
\Rightarrow \{\text{union, theorem 2.9}\}
\]
\[
\forall i :: a \operatorname{unless}_{CM} b \text{ in } (G \setminus S_{i})
\]
\[
\Rightarrow \{\text{conjunction (A.3) } q_{i} \operatorname{unless}_{CM} \neg p \text{ in } (G \setminus S_{i})\}
\]
\[
\forall i :: (a \land q_{i}) \operatorname{unless}_{CM} ((a \land \neg p) \lor b) \text{ in } (G \setminus S_{i})
\]
\[
= \{\text{consequence weakening (A.2) } [(a \land \neg p) \Rightarrow b]\}
\]
\[
\forall i :: (a \land q_{i}) \operatorname{unless}_{CM} b \text{ in } (G \setminus S_{i})
\]
\[
= \{[(a \land q_{i} \land \neg b) \Rightarrow S_{i}(b)]\}
\]
\[
\forall i :: (a \land q_{i}) \text{ ensures}_{CM} b \text{ in } G
\]
\[
\Rightarrow \{\text{ensures lifting}\}
\]
\[
\forall i :: (a \land q_{i}) \rightarrow_{CM} b \text{ in } G
\]
\[
\Rightarrow \{\text{disjunctivity, see below}\}
\]
\[
(a \land \exists i :: q_{i}) \rightarrow_{CM} b \text{ in } G
\]
\[
\Rightarrow \{[a \land \neg b \Rightarrow p] \text{ and } [p \Rightarrow \exists i :: q_{i}]\}
\]
\[
a \rightarrow_{CM} b \text{ in } G
\]
\[
\Rightarrow \{\text{lifting}\}
\]
\[
a \rightarrow_{\ast} b \text{ in } G
\]

For the disjunctivity step in the proof above, we must check the condition for the disjunctivity of \(\rightarrow_{CM}\). This condition follows from the assumption on predicates \(q_{i}\):
\[ a \land q_i \land \neg b \Rightarrow \text{inter}(F)(\forall w :: \neg(a \land q_w) \lor q_i \lor b) \]

\[ \{\text{predicate calculus}\} \]

\[ a \land q_i \land \neg b \Rightarrow \text{inter}(F)(\neg a \lor (\forall w :: \neg q_w) \lor q_i \lor b) \]

\[ \{(a \land \neg p) \Rightarrow b\} \]

\[ a \land q_i \land \neg b \land p \Rightarrow \text{inter}(F)(\neg a \lor (\forall w :: \neg q_w) \lor q_i \lor b \lor \neg p) \]

\[ \{\text{predicate calculus}\} \]

\[ q_i \land p \Rightarrow \text{inter}(F)(\forall w :: \neg q_w) \lor q_i \lor \neg p \]

\[ \{p \Rightarrow (\exists i :: q_i)\} \]

\[ q_i \land p \Rightarrow \text{inter}(F)(q_i \lor \neg p) \]

Example 4.6.

Programs \( F \) and \( G \) in example 4.6 illustrate the use of this program transformation rule. Program \( G \) is the same as program \( F \) except that the statement of \( F \) has been split into two statements. The refinement \( F \subseteq G \) can be shown by taking the set of predicates \( \{b, \neg b\} \).

### 4.5 Conclusions

In this chapter we introduced the first part of the ImpUNITY framework. ImpUNITY programs are UNITY program extended with an extra section that is used to restrict the interference of an environment. Restriction of interference was modelled by modifiers. This extension of the language led to the ImpUNITY logic, a UNITY like logic that takes interference of environments into account. Properties in the ImpUNITY logic are compositional and can be used for specifying components. A number of programming transformation rules exploiting the restriction of interference were given.
Chapter 5

Observability

Observability is about the way in which the state space can be observed. This can, for example, be used to express that the value of certain variables cannot be observed. In the ImpUNITY framework, it is used to express that some variables of a program are hidden, i.e. they cannot be read by an environment. This has two consequences. First, it restricts the way in which programs are composed, because an environment is not allowed to read hidden variables (but it may write these variables). Second, it results in a new notion of refinement that allows “invisible” behaviour to be changed. Together with the modifiers introduced in the previous chapter, we can express that a variable is local to a program. This yields a notion of data refinement that allows local structures to be replaced.

The chapter is organised as follows. In section 5.1, we examine observability in a general setting using predicate transformers. Section 5.2 introduces observability in the ImpUNITY framework and shows the consequences for the notion of program refinement. In section 5.3, we give some program transformation rules exploiting observability and in section 5.4 we show how data refinement is modelled in the ImpUNITY framework.

5.1 Observability

Observability is about the distinction of different states in the state space. For example, we want to be able to model that different states are observed in the same way because they only differ in variables that cannot be observed. We formalise this with the introduction of a view which is a set of variables that cannot be observed. To reason about views on the level of predicate transformers, we use two equivalence commands based on views: \( \{\#O\} \) and \( [\#O] \). Both commands change the state to an arbitrary, observably equivalent state, i.e. a state that only differs on variables in \( O \). Command \( \{\#O\} \) updates the state in an angelic way, while \( [\#O] \) is demonic. A statement \( S \) is called a stuttering with respect to a view \( O \) if the effect of \( S \) is not visible when variables in \( O \) are not
observed, i.e. if it changes only variables in $O$. This is equivalent to saying that $S$ is a refinement of $[\triangleright O]$, i.e. $[\triangleright O] \leq S$.

We say that a predicate respects a view if it does not depend on the variables in that view, i.e. if it does not yield different truth values for states that are observably equivalent.

**Definition 5.1** For predicate $p$ and view $O$, define

$$p \text{ resp } O \overset{\text{def}}{=} [[\triangleright O](p) \Rightarrow p].$$

Commands $[\triangleright O]$ and $[\triangleright O]$ form an adjoint pair and both commands are equivalence commands. Therefore, $p \text{ resp } O$ can be also stated by $[[\triangleright O](p) = p]$ and $[p = [[\triangleright O](p)]].$

We say that a view $O'$ is more specific than a view $O$ if $O'$ allows a more detailed observation of the state space, i.e. if $O' \subseteq O$, so if less variables are hidden. Predicates that respect a view also respect more specific views. This (anti-)monotonicity property is expressed in the following lemma.

**Lemma 5.2** Let $O$ and $O'$ be views such the $O' \subseteq O$. Then for predicate $p$,

$$p \text{ resp } O \Rightarrow p \text{ resp } O'.$$

**Proof:**

$$p \text{ resp } O \Rightarrow p \text{ resp } O'$$

$$= \{\text{definition}\}$$

$$[[\triangleright O](p) \Rightarrow p] \Rightarrow [[\triangleright O'](p) \Rightarrow p]$$

$$\Leftarrow \{\text{refinement calculus}\}$$

$$\{\triangleright O'\} \leq \{\triangleright O\}$$

$$\Leftarrow \{O' \subseteq O \text{ and lemma 2.5}\}$$

$$true$$

\[\square\]

We want to extend the notion of respecting a view to commands by expressing that a command respects a view if it is only sensitive to differences in states that are observable. Before we give the exact definition, we give the following lemma.

**Lemma 5.3** For command $A$ and view $O$, the following three expressions are equivalent,

1. $\forall p : p \text{ resp } O : A(p) \text{ resp } O$,

2. $\{\triangleright O\}; A; \{\triangleright O\} = A; \{\triangleright O\}$,

3. $A \leq_{\{\triangleright O\}} A$. 
5.1. Observability

Proof:

\[ \langle \forall p : p \text{ resp } O : A(p) \text{ resp } O \rangle \]

= \{ \text{definition} \}
\[ \langle \forall p : (\Diamond_{O}(p) = p : (\Diamond_{O}; A)(p) = A(p)) \rangle \]

= \{ \text{predicate calculus} \}
\[ \langle \forall p :: (\Diamond_{O}; A; \Diamond_{O})(p) = (A; \Diamond_{O})(p) \rangle \]

= \{ \text{definition} \}
\[ (\Diamond_{O}; A; \Diamond_{O}) = A; \Diamond_{O} \]

= \{ \text{skip } \leq \{ \Diamond_{O} \} \}
\[ (\Diamond_{O}; A; \Diamond_{O}) \leq A; \Diamond_{O} \]

= \{ \Rightarrow: \text{skip } \leq \{ \Diamond_{O} \}, \Leftarrow: \{ \Diamond_{O} \} \text{ and } \{ \Diamond_{O} \} = \{ \Diamond_{O} \}; \{ \Diamond_{O} \} \}
\[ (\Diamond_{O}; A \leq A; \Diamond_{O}) \]

= \{ \text{definition} \}
\[ A \leq \{ \Diamond_{O} \} A \]

\[ \square \]

The first expression is the easiest one to interpret: for a predicate \( p \) that does not depend on variables in \( O \), \( A(p) \) does not depend on variables in \( O \) either. In other words, for predicates respecting the view \( O \), command \( A \) does not yield observably different results. So, it expresses that command \( A \) is independent of a view \( O \), or that view \( O \) is auxiliary to \( A \). The last expression is a well-known expression in the refinement calculus saying that command \( A \) is a data refinement through \( \{ \Diamond_{O} \} \) of itself. It is known that a set \( O \) may be auxiliary to a command \( A \) while a subset \( O' \subseteq O \) is not. For example, command \( x := y \) is independent of the set of variables \( \{ x, y \} \), i.e. \( x := y \newcommand{\cong}{\equiv} (x, y) \cong (y, y) \) \( x := y \). If neither variable \( x \) nor variable \( y \) can be observed, the command \( x := y \) does not have any observable effect. However, the command is not independent of variable \( y \), since \( x := y \newcommand{\cong}{\equiv} (x, y) \cong (y, y) \) \( x := y \), does not hold. When \( x \) can be observed the effect of \( x := y \) is observable.

We want to model the idea that a statement is not sensitive to differences of states that are not observable, i.e. it does not refer to variables that are not observable. This notion is monotonic in views in the sense that if a command respects some view, then it also respects all more specific views. We require monotonicity explicitly in the following definition.
**Definition 5.4** For command $A$ and view $O$, define

$$A \text{ resp } O \overset{\text{def}}{=} \langle \forall O' : O' \subseteq O : A \leq_{\{\text{def}_{O'}\}} A \rangle.$$ 

Our notion of respecting a view is weaker than the “intuitive” notion that a command does not read the variables in a view. For example, the command $x := x + 1$ respects the view $\{x\}$. This notion is strong enough for our purposes.

Monotonicity is expressed by the following lemma.

**Lemma 5.5** Let $O$ and $O'$ be views such that $O' \subseteq O$. Then for command $A$,

$$A \text{ resp } O \implies A \text{ resp } O'.$$

The following lemma states that a command respects two views if and only of it respects the union of both views.

**Lemma 5.6** For views $O_1$ and $O_2$, and command $A$,

$$A \text{ resp } O_1 \land A \text{ resp } O_2 = A \text{ resp } (O_1 \cup O_2).$$

**Proof:** The implication from right to left follows directly from lemma 5.5. Here we focus on the implication from left to right.

\[
A \text{ resp } O_1 \land A \text{ resp } O_2
\]

\[
= \quad \{\text{definition}\}
\]

\[
\langle \forall O_1' : O_1' \subseteq O_1 : A \leq_{\{\text{def}_{O_1'}\}} A \rangle \land \langle \forall O_2' : O_2' \subseteq O_2 : A \leq_{\{\text{def}_{O_2'}\}} A \rangle
\]

\[
= \quad \{\text{predicate calculus}\}
\]

\[
\langle \forall O_1', O_2' : O_1' \subseteq O_1 \land O_2' \subseteq O_2 : A \leq_{\{\text{def}_{O_1'}, \text{def}_{O_2'}\}} A \land A \leq_{\{\text{def}_{O_1'}, \text{def}_{O_2'}\}} A \rangle
\]

\[
\implies \quad \{\text{refinement calculus}\}
\]

\[
\langle \forall O_1', O_2' : O_1' \subseteq O_1 \land O_2' \subseteq O_2 : A \leq_{\{\text{def}_{O_1'}, \{\text{def}_{O_2'}\}\}} A \rangle
\]

\[
= \quad \{\text{set theory}\}
\]

\[
\langle \forall O' : O' \subseteq (O_1 \cup O_2) : A \leq_{\{\text{def}_{O'}\}} A \rangle
\]

\[
= \quad \{\text{definition}\}
\]

$$A \text{ resp } (O \cup O')$$

\[\square\]
The following lemmas can be used to prove that a command respects a view.

**Lemma 5.7** An assignment statement \( x := E \) respects a view \( O \) if the expression \( E \) does not refer to variables in \( O \).

A predicate respecting a view \( O \) can be lifted to a guard command or to an assert command. These commands also respect the view \( O \).

**Lemma 5.8** For predicate \( p \) that respects view \( O \),

\[
\{ p \} \text{ resp } O \quad \text{and} \quad [ p ] \text{ resp } O.
\]

**Proof:** For every view \( O' \) such that \( O' \leq O \),

\[
\{ p \} \leq_{\{\text{resp}\}} \{ p \}
\]

= \{ definition \}

\[
\langle \forall q :: \{ \Box O' \}; \{ p \}(q) \Rightarrow \{ p \}; \{ \Box O' \}(q) \rangle
\]

= \{ definition \}

\[
\langle \forall q :: \{ \Box O' \}(p \land q) \Rightarrow (p \land \{ \Box O' \}(q)) \rangle
\]

= \{ p \text{ resp } O \}

\[
\langle \forall q :: \{ \Box O' \}((\{ \Box O \})(p) \land q) \Rightarrow ((\{ \Box O \})(p) \land \{ \Box O' \}(q)) \rangle
\]

= \{ lemma 2.3 \}

\[
true
\]

The proof for \([ p ]\) is similar. \( \square \)

Furthermore, commands that are constructed using sequential, demonic or angelic composition respect a view if the components respect that view.

**Lemma 5.9** Let \( O \) be a view. For commands \( A_1 \) and \( A_2 \), and set of commands \( \{ A_i \mid i \in I \} \) such that \( A_1 \text{ resp } O \), \( A_2 \text{ resp } O \), and \( A_i \text{ resp } O \) for all \( i \in I \)

\[
(A_1; A_2) \text{ resp } O,
\]

\[
(\land_{i \in I} A_i) \text{ resp } O,
\]

\[
(\lor_{i \in I} A_i) \text{ resp } O.
\]
5.2 Observability in ImpUNITY

In this section we show how observability is incorporated in the ImpUNITY framework. We extend the ImpUNITY programming language and use views to express which variables can be observed by the program and which variables are observable by an environment. We modify the notion of environment in such a way that it takes observability into account. We also incorporate the notion of observability into the notion of program refinement.

We extend the ImpUNITY programming language with the following section:

- The hide-section, containing a view specifying which part of the state space can be observed by an environment. This view is denoted by $\text{hide}(F)$ and each environment must respect this view. We use the convention that if $\text{hide}(F)$ is the empty set of variables, then the hide-section is not mentioned explicitly.

The hide-section models a restriction on the environment using a view. Definition 5.1 and 5.4 state what it means for predicates and commands to respect a view. This idea can be lifted to ImpUNITY programs: a program respects a view if the initially-section as well as the statements in the assign-section respect the view.

**Definition 5.10** For ImpUNITY program $F$ and view $O$, define

$$F \text{ resp } O \overset{\text{def}}{=} (\text{init}(F) \text{ resp } O) \land (\forall S : S \in \text{assign}(F) : S \text{ resp } O).$$

Now, an environment of $F$ is defined as a program that satisfies both the interference restriction and the observability restriction of $F$.

**Definition 5.11** For ImpUNITY programs $F$ and $H$, by $F \text{ env } H$ we denote that $H$ is an environment of $F$ which is defined by

$$H \text{ env } F \overset{\text{def}}{=} H \text{ obey external}(F) \land H \text{ resp } \text{hide}(F).$$

By $H \text{ cont } F$ we denote that $H$ is a context of $F$ which is defined

$$H \text{ cont } F \overset{\text{def}}{=} F \text{ env } H \land H \text{ env } F.$$

Consider program $F$ in example 5.1. The hide-section states that variable $y$ cannot be observed by an environment.

Next, we introduce the hide operator which is defined as follows.

**Definition 5.12** For ImpUNITY program $F$ and view $O$, program $\text{hide}(O : F)$ is defined as the program that only differs from $F$ in the hide-section and

$$\text{hide}(\text{hide}(O : F)) \overset{\text{def}}{=} \text{hide}(F) \cup O.$$
Program $F$
\begin{align*}
\text{hide } y \\
\text{init } y &= 0 \\
\text{assign } x, y &: = 0, 10
\end{align*}
\begin{align*}
\quad | y > 0 & \rightarrow y &: = y - 1 \\
\text{end}\{F\}
\end{align*}

Example 5.1.

For example, for program $F$ in example 5.1, program hide($\{x\} : F$) is the program in which neither $x$ nor $y$ can be observed by any environment.

Now, we extend definition 4.14 of program union.

**Definition 5.13** For ImpUNITY programs $F$ and $G$, the union $F | G$ is defined by

\begin{align*}
\text{hide}(F | G) & \overset{\text{def}}{=} (\text{hide}(F) \cup \text{hide}(G)), \\
\text{external}(F | G) & \overset{\text{def}}{=} (\text{external}(F) + \text{external}(G)), \\
\text{init}(F | G) & \overset{\text{def}}{=} (\text{init}(F) \land \text{init}(G)), \\
\text{assign}(F | G) & \overset{\text{def}}{=} (\text{assign}(F) \cup \text{assign}(G)).
\end{align*}

As a consequence of lemma 5.6 and the way programs are composed, we can extend lemma 4.15 to programs with observability restrictions. The following lemma states that it is possible to examine all components separately when checking the environment conditions.

**Lemma 5.14** For ImpUNITY programs $F$, $G$ and $H$,

\begin{align*}
H \text{ env } F | G & = H \text{ env } F \land H \text{ env } G, \\
F | G \text{ env } H & = F \text{ env } H \land G \text{ env } H.
\end{align*}

The hide-section of a program specifies that a part of the state space is not visible for any environment. This has consequences for the properties of programs and the notion of refinement. We follow Zhou et al. [ZGK93] and define a notion of observable properties. For a program $F$, observable properties are properties that do not refer to the hidden state space of the program. This is done by renaming the hidden variables of the program. Observable properties are subscripted by $O$.

**Definition 5.15** For ImpUNITY program $F$ and predicates $p$ and $q$, observable properties of $F$ are defined by

\begin{align*}
p \text{ unless}_O q \text{ in } F & \overset{\text{def}}{=} p \text{ unless}_S q \text{ in } F', \\
p \text{ ensures}_O q \text{ in } F & \overset{\text{def}}{=} p \text{ ensures}_S q \text{ in } F', \\
p \leftrightarrow_O q \text{ in } F & \overset{\text{def}}{=} p \leftrightarrow_S q \text{ in } F'.
\end{align*}
where $F'$ is a program that is similar to $F$ except that hidden variables of $F$ are renamed such that both $p \text{ resp hide}(F')$ and $q \text{ resp hide}(F')$.

Program $F$ given in example refex:hide hides variable $y$. Observable properties of $F$ with predicates that referring to $y$ can be calculated by renaming $y$ by some variable that occurs neither in $F$ nor in the predicates. Since the predicates of the property do not refer to the new variables, the specific choice of the new variables does not matter. For example, property $x = y \text{ unless}_S x = 0$ in $F$ can be proven by renaming $y$ by $y'$ and prove $x = y \text{ unless}_S x = 0$ in $F[y'/y]$.

Earlier we defined refinement of ImpUNITY programs as preservation of all $\text{unless}_S$ and $\rightarrow_S$ in any environment. The purpose of the introduction of observability is to weaken the refinement relation: we do not want to preserve all properties, we are only interested in preservation of observable properties. This is formalised in the following notion of observable refinement, which is a modification of definition 4.16.

**Definition 5.16** For ImpUNITY programs $F$ and $G$, by $F \sqsubseteq G$ we denote that $G$ is an observable refinement of $F$, denoted by $F \sqsubseteq_0 G$, if for all predicates $p$ and $q$,

\[
\begin{align*}
\text{p unless}_S q \text{ in } F & \quad \Rightarrow \quad \text{p unless}_0 q \text{ in } G, \\
\text{p } \not\rightarrow_S q \text{ in } F & \quad \Rightarrow \quad \text{p } \not\rightarrow_0 q \text{ in } G.
\end{align*}
\]

Again, since we want to have a compositional notion of refinement, we define refinement of ImpUNITY programs as observable refinement in any environment.

**Definition 5.17** For ImpUNITY programs $F$ and $G$, by $F \sqsubseteq G$ we denote that $G$ is a refinement of $F$ which is defined by

\[
F \sqsubseteq G \overset{\text{def}}{=} (\forall H : H \text{ env } F : H \text{ env } G \land F[H] \sqsubseteq_0 G[H]).
\]

The compositionality result of lemma 4.18 can be extended too.

**Lemma 5.18** For ImpUNITY programs $F$, $G$ and $H$, such that $H \text{ env } F$,

\[
F \sqsubseteq G \quad \Rightarrow \quad F[H] \sqsubseteq G[H].
\]

### 5.3 Program Transformation Rules

In this section we give some program transformation rules that exploit observability. This is done by introducing the idea of refinement through an abstraction. First we give a theorem stating that the preservation of ImpUNITY properties through certain kinds of abstractions implies refinement. In subsection 5.3.1 we show how this can be used to reschedule stutterings of a program, and in subsection 5.3.2 we show how statements can be refined through an abstraction.
For the refinement of ImpUNITY programs with observability restrictions, we do not want to preserve all ImpUNITY properties; it is sufficient to preserve observable properties. This idea is used in the idea of refinement through an abstraction. First, we define what we mean by an abstraction.

**Definition 5.19** A command $A$ is called an abstraction if $A$ is universally disjunctive and terminating.

Then, the following theorem states that preservation of properties through an abstraction implies refinement as long as the abstraction is not observable.

**Theorem 5.20** Let $F$ and $G$ be ImpUNITY programs that only differ in their initially and assign-sections. Let $A$ be an abstraction such that $A \subseteq \{\text{hide}(F)\}$. If for every program $H$ such that $H \text{ env } F$, also $H \text{ env } G$ holds and for all predicates $p$ and $q$,

$$
\begin{align*}
A(\text{init}(F)) & \subseteq \text{init}(G), \\
p \text{ unless}_CM q \text{ in } F & \Rightarrow A(p) \text{ unless}_* A(q) \text{ in } G, \\
p \text{ ensures}_CM q \text{ in } F & \Rightarrow A(p) \rightarrow_* A(q) \text{ in } G,
\end{align*}
$$

and if $p \text{ unless}_CM q \text{ in } F$ then

$$
\begin{align*}
p \text{ unless}_CM q \text{ in } H & \Rightarrow A(p) \text{ unless}_* A(q) \text{ in } H, \\
p \text{ ensures}_CM q \text{ in } H & \Rightarrow A(p) \rightarrow_* A(q) \text{ in } H,
\end{align*}
$$

then $F \sqsubseteq G$.

**Proof:** Let $H$ be an environment of $F$. Since $F$ and $G$ do not differ in the external and the hide-sections, $H$ is also an environment of $G$. First, we show that properties of $F[H$ are preserved through abstraction $A$. For $\text{unless}_CM$ properties:

$$
\begin{align*}
p \text{ unless}_CM q \text{ in } F[H \\
= & \{\text{union, theorem 2.9}\} \\
= & p \text{ unless}_CM q \text{ in } F \land p \text{ unless}_CM q \text{ in } H \\
\Rightarrow & \{\text{assumptions}\} \\
A(p) \text{ unless}_* A(q) \text{ in } G \land A(p) \text{ unless}_* A(q) \text{ in } H \\
= & \{\text{union, lemma 4.22}\} \\
A(p) \text{ unless}_* A(q) \text{ in } G[H
\end{align*}
$$
In a similar way, using compositionality of \( \rightarrow_s \) expressed by lemma 4.25, we can derive

\[
p \text{ ensures}_{CM} q \text{ in } F | H \quad \Rightarrow \quad A(p) \rightarrow_s A(q) \text{ in } F | H.
\]

Then, by induction on the prove of the \( \rightarrow_{CM} \) property it is proved that

\[
p \rightarrow_{CM} q \text{ in } F | H \quad \Rightarrow \quad A(p) \rightarrow_s A(q) \text{ in } F | H.
\]

Furthermore, the initially-section of the composition is preserved:

\[
A(\text{init}(F | H)) \Leftarrow \text{init}(G | H)
\]

= \{union\}

\[
A(\text{init}(F) \land \text{init}(H)) \Leftarrow \text{init}(G) \land \text{init}(H)
\]

= \{H resp hide(F), lemma 2.3\}

\[
A(\text{init}(F)) \land \text{init}(H) \Leftarrow \text{init}(G) \land \text{init}(H)
\]

= \{assumption\}

\[
\text{true}
\]

Invariants are preserved too:

\[
invariant_{CM} r \text{ in } F | H
\]

= \{definition\}

\[
[\text{init}(F | H) \Rightarrow r] \land r \text{ unless}_{CM} \text{false in } F | H
\]

\[
\Rightarrow \quad \{\text{monotonicity}\}
\]

\[
[A(\text{init}(F | H)) \Rightarrow A(r)] \land r \text{ unless}_{CM} \text{false in } F | H
\]

\[
\Rightarrow \quad \{\text{above}\}
\]

\[
[A(\text{false}) = \text{false}]
\]

\[
[\text{init}(G | H) \Rightarrow A(r)] \land A(r) \text{ unless}_{s} A(\text{false}) \text{ in } G | H
\]

\[
\Rightarrow \quad \{A(\text{false}) = \text{false}\}
\]

\[
[\text{init}(G | H) \Rightarrow A(r)] \land A(r) \text{ unless}_{s} \text{false in } G | H
\]

= \{definition\}

\[
invariant_{s} A(r) \text{ in } G | H
\]

Then, for property \( \text{Prop} \in \{\text{unless, } \rightarrow\} \) and predicates \( p \) and \( q \) such that \( p \text{ resp hide}(F) \) and \( q \text{ resp hide}(F) \),
Also observable properties of $F|H$ are preserved. Let $Y$ be the lists of hidden variables of $F|H$ and let $Z$ be a list of variables such that $F, G, H, p$ and $q$ respect $Z$.

\[ p \text{ Prop}_S q \text{ in } F|H \Rightarrow p \text{ Prop}_O q \text{ in } G|H \]

\[ = \quad \{ \text{definition} \} \]

\[ p \text{ Prop}_S q \text{ in } ((F|H)[Z/Y]) \Rightarrow p \text{ Prop}_S q \text{ in } ((G|H)[Z/Y]) \]

\[ = \quad \{ \text{renaming} \} \]

\[ p[Z/Y] \text{ Prop}_S q[Y/Z] \text{ in } F|H \Rightarrow p[Z/Y] \text{ Prop}_S q[Y/Z] \text{ in } G|H \]

\[ \Leftarrow \quad \{ \text{above} \} \]

\[ \text{true} \]

So, every observable unless and $\Rightarrow$ property of $F|H$ is an observable property of $G|H$ and $F \subseteq G$.

In chapter 4 we only dealt with interference and gave some program transformation rules. These transformation rules are instantiations of theorem 4.32. By taking the abstraction $A \overset{\text{def}}{=} \text{skip}$, theorem 5.20 specialises to theorem 4.32. Therefore, all program transformation rules given in section 4.4 can also be used for ImpUNITY programs with observability restrictions. Furthermore, lemma 4.33 expressed that preservation of Chandy and Misra properties corresponds to preservation of properties in the new logic. This lemma can be lifted to preservation of properties through some abstraction if the abstraction is conjunctive.
5.3.1 Rescheduling Stutterings

In this subsection we show that theorem 5.20 can be used to reschedule stutterings in a program. Before we give the transformation rule we give three lemmas. The first two lemmas show that for well-chosen predicates and a special abstraction basic UNITY properties are preserved through this abstraction.

For the preservation of properties of a program through some abstraction \( A \), we prove the following lemma.

**Lemma 5.21** Let \( F \) be an ImpUNITY program and let \( A \) be an abstraction such that \( \text{skip} \leq A \). For all predicates \( p \) and \( q \) such that \( \neg A(q) \Rightarrow (p = A(p)) \):

\[
\begin{align*}
p \text{ unless}_{CM} q & \text{ in } F \quad \Rightarrow \quad A(p) \text{ unless}_{*} A(q) \text{ in } F, \\
p \text{ ensures}_{CM} q & \text{ in } F \quad \Rightarrow \quad A(p) \Rightarrow_{*} A(q) \text{ in } F.
\end{align*}
\]

**Proof:** For \( \text{Prop} \in \{\text{unless}_{CM}, \text{ensures}_{CM}\} \) we derive

\[
p \text{ Prop } q
\]

\[
\Rightarrow \quad \{ \text{consequence weakening (A.2,A.13) skip} \leq A \}
\]

\[
p \text{ Prop } A(q)
\]

\[
= \quad \{ \text{(A.8,A.20) assumption on } p \text{ and } q \}
\]

\[
A(p) \text{ Prop } A(q)
\]

Then, the lemma follows from lemma 4.29, which lifts properties in Chandy and Misra’s logic to ImpUNITY properties. \( \square \)

The next lemma shows that for an abstraction \( A \), statements may be modified such that properties are preserved through \( A \).

**Lemma 5.22** Let \( S \) and \( T \) be statements and let \( A \) be an abstraction such that \( \text{skip} \leq A \) and \( A_R \leq T \). For all predicates \( p \) and \( q \) such that \( \neg A(q) \Rightarrow (p = A(p)) \):

\[
\begin{align*}
p \text{ unless}_{CM} q & \text{ in } S \quad \Rightarrow \quad A(p) \text{ unless}_{*} A(q) \text{ in } (S; T), \\
p \text{ ensures}_{CM} q & \text{ in } S \quad \Rightarrow \quad A(p) \Rightarrow_{*} A(q) \text{ in } (S; T).
\end{align*}
\]

**Proof:**

\[
p \text{ unless}_{CM} q \text{ in } S
\]

\[
= \quad \{ \text{definition} \}
\]

\[
[(p \land \neg q) \Rightarrow S(p \lor q)]
\]

\[
\Rightarrow \quad \{ \text{adjoints } \text{skip} \leq (A_R; A) \}
\]
\[(p \land \lnot q) \Rightarrow S; A_R; A(p \lor q)\]
\[\Rightarrow \{\text{skip } \leq A, \text{ disjunctivity}\}\]
\[[(p \land \lnot A(q)) \Rightarrow (S; A_R)(A(p) \lor A(q))]\]
\[= \{\text{assumptions}\}\]
\[[(A(p) \land \lnot A(q)) \Rightarrow (S; T)(A(p) \lor A(q))]\]
\[= \{\text{definition}\}\]
\[A(p) \text{ unless}_{CM} A(q) \text{ in } (S; T)\]
\[\Rightarrow \{\text{lemma 4.29}\}\]
\[A(p) \text{ unless}_{A} A(q) \text{ in } (S; T)\]

The proof for \(\text{ensures}_{CM}\) is similar. \(\Box\)

The following lemma gives a simpler way to prove the conditions on \(A\) as given in the previous lemmas.

**Lemma 5.23** Let \(A\) be an abstraction such that \(\text{skip } \leq A\). For all predicates \(p\) and \(q\),
\[A(p \land \lnot q) \Rightarrow p \lor A(q)\]  \(=\)  \[-A(q) \Rightarrow (p = A(p))\] .

**Proof:**
\[-A(q) \Rightarrow (p = A(p))\]
\[= \{\text{skip } \leq A\}\]
\[-A(q) \Rightarrow (A(p) \Rightarrow p)\]
\[= \{\text{predicate calculus}\}\]
\[A(p) \Rightarrow (p \lor A(q))\]
\[= \{\text{disjunctivity}\}\]
\[(A(p \land \lnot q) \lor A(p \land q)) \Rightarrow (p \lor A(q))\]
\[= \{\text{monotonicity}\}\]
\[A(p \land \lnot q) \Rightarrow (p \lor A(q))\]
\(\Box\)

These lemmas result in the following transformation rule that allows stutterings to be rescheduled. Recall that a statement \(T\) is a stuttering with respect to a view \(O\) if \([\llcorner O \lrcorner] \leq T\). Statement \(T\) is a stuttering of a program \(F\) if it is a stuttering with respect to the hidden variables of the program, i.e. \([\llcorner \text{hide}(F) \lrcorner] \leq T\), and it is a transition of the program, i.e. \((\wedge_{\text{assign}(F)} S \land \text{skip}) \leq T\).
Chapter 5. Observability

Transformation 5.24 (Reschedule stutterings) Let $F$ and $G$ be ImpUNITY programs that only differ in their assign-sections and let $\text{assign}(F) = \{ S_i \mid i \in I \}$ and $\text{assign}(G) = \{ S'_i \mid i \in I \}$. If for some statement $T$,

- $(\bigwedge_{i \in I} S_i \land \text{skip}) \leq T$,
- $[\bigtriangleup_{\text{hide}(F)}] \leq T$,
- $[(T^*)_L(\text{init}(F)) \Leftarrow \text{init}(G)]$, and
- $(S_i; T^*) \leq S'_i$ for all $i \in I$,

then $F \sqsubseteq G$.

**Proof:** Let $A \overset{\text{def}}{=} (T^*)_L$. Then, $A$ is an abstraction and

\[
A \leq \{ [\bigtriangleup_{\text{hide}(F)}] \}
\]

= \{ definition \}

\[
(T^*)_L \leq \{ [\bigtriangleup_{\text{hide}(F)}] \}
\]

= \{ adjoints \}

\[
[\bigtriangleup_{\text{hide}(F)}] \leq T^*
\]

= \{ lemma 4.3, $[\bigtriangleup_{\text{hide}(F)}]$ is a modifier \}

\[
[\bigtriangleup_{\text{hide}(F)}] \leq T
\]

= \{ assumption \}

true

For all predicates $p$ and $q$, we can prove by induction that $[p \land \neg q \Rightarrow T(p \lor q)]$ implies that $[(T_L)^i(p \land \neg q) \Rightarrow p \lor A(q)]$, for any $i \in \mathbb{N}$. Then, for all predicates $p$ and $q$,

\[
[\neg A(q) \Rightarrow (p = A(p))]
\]

$\Leftarrow$ \{lemma 5.23\}

\[
[A(p \land \neg q) \Rightarrow p \lor A(q)]
\]

= \{ $A = \lor_{i \in I} (T_L)^i$ \}

\[
[\lor_{i \in I} (T_L)^i(p \land \neg q) \Rightarrow p \lor A(q)]
\]

= \{ refinement calculus \}

\[
(\forall i : i \in I : [(T_L)^i(p \land \neg q) \Rightarrow p \lor A(q)]
\]

$\Leftarrow$ \{ induction \}
5.3. Program Transformation Rules

\[ p \land \neg q \Rightarrow T(p \lor q) \]
\[ \{ \text{assumption } (\forall i \in I \ S_i \land \text{skip}) \leq T \} \]
\[ [p \land \neg q \Rightarrow (\forall i \in I \ S_i \land \text{skip})(p \lor q)] \]
\[ = \{ \text{predicate calculus} \} \]
\[ [p \land \neg q \Rightarrow (\forall i \in I \ S_i)(p \lor q)] \]
\[ = \{ \text{definitions} \} \]
\[ p \text{ unless}_{CM} q \text{ in } F \]

Then, by the assumptions and lemma 5.22 we have for all predicates \( p \) and \( q \)

\[ A(\text{init}(F)) \Leftarrow \text{init}(G), \]
\[ p \text{ unless}_{CM} q \text{ in } F \Rightarrow A(p) \text{ unless}_{*} A(q) \text{ in } G, \]
\[ p \text{ ensures}_{CM} q \text{ in } F \Rightarrow A(p) \rightarrow_{*} A(q) \text{ in } G. \]

Furthermore, for every program \( H \) and for all predicates \( p \) and \( q \), if \( p \text{ unless}_{CM} q \text{ in } F \) holds, then \( \neg A(q) \Rightarrow (p = A(p)) \) holds and by by lemma 5.21

\[ p \text{ unless}_{CM} q \text{ in } H \Rightarrow A(p) \text{ unless}_{*} A(q) \text{ in } H, \]
\[ p \text{ ensures}_{CM} q \text{ in } H \Rightarrow A(p) \rightarrow_{*} A(q) \text{ in } H. \]

Since \( F \) and \( G \) do not differ external and hide-sections, every environment of \( F \) is also an environment of \( G \), the refinement \( F \sqsubseteq G \) follows from lemma 5.20. \( \square \)

\begin{center}
\begin{tabular}{ll}
Program \( F \) & Program \( G \) \\
hide & hide \y \\
init \( y = 0 \) & assign \( x := 0 \parallel y :\in \{i \mid 0 \leq i \leq 10\} \)
\ assign \( x, y := 0, 10 \) & \ | \( y > 0 \rightarrow y := y - 1 \) \\\n| \( y > 0 \rightarrow y := y - 1 \) & end\{G\} \\\nend\{F\} & \\
\end{tabular}
\end{center}

Example 5.2.

Programs \( F \) and \( G \) in example 5.2 show how transformation 5.24 can be used to reschedule stutterings. Take for \( T \) the second statement of program \( F \), such that \( T^* \Rightarrow y > 0 \rightarrow y :\in \{i \mid 0 \leq i \leq y\} \). This statement is scheduled after the first statement of \( F \), and this results in program \( G \). So, \( G \) is a refinement of \( F \).

In the action system formalism rescheduling of stuttering is explicitly coded into the data refinement or simulation rule and the correctness is proven on the level of
the execution sequences of the program [Wri92, BvW94]. Since UNITY properties are insensitive to stuttering, we can prove the correctness of re-scheduling of stutterings within the formalism.

Transformation 5.24 gives a way to rewrite statements using stutterings, but it does not allow stutterings to be removed from a program. This rewriting of statements corresponds to the idea of weak simulation of Back and van Wright [BvW94]. Back and von Wright give a notion of simulation that does allow stutterings to be removed. This can also be modelled in our framework but we have to be more careful since we need to preserve fairness properties. Before we give this rule, we prove the following lemma.

**Lemma 5.25** Let $S$ and $T$ be statements and let $A \overset{\text{def}}{=} (\text{skip} \lor T_L)$. For all predicates $p$ and $q$ such that $[(p \land \neg q) \Rightarrow T(q)]$:

$$p \text{ unless}_CM q \text{ in } S \Rightarrow A(p) \Rightarrow A(q) \text{ in } (S; T).$$

**Proof:** It is straightforward to prove $\text{skip} \leq A$ and $[\neg A(q) \Rightarrow (p = A(p))]$. Then, by lemma 5.21 we have

$$p \text{ unless}_CM q \text{ in } S \Rightarrow A(p) \text{ unless}_CM A(q) \text{ in } S$$

and furthermore,

$$p \text{ unless}_CM q \text{ in } S$$

$$= \{\text{definition}\}$$

$$[(p \land \neg q) \Rightarrow S(p \lor q)]$$

$$\Rightarrow \{\text{adjoints } \text{skip} \leq T; T_L\}$$

$$[(p \land \neg q) \Rightarrow S; T; T_L(p \lor q)]$$

$$= \{\text{disjunctivity}\}$$

$$[(p \land \neg q) \Rightarrow S; T(T_L(p \land \neg q) \lor T_L(q))]$$

$$\Rightarrow \{\text{assumption}\}$$

$$[(p \land \neg q) \Rightarrow S; T(q \lor T_L(q))]$$

$$= \{\text{definition}\}$$

$$[(p \land \neg q) \Rightarrow S; T; A(q)]$$

$$\Rightarrow \{\text{skip} \leq A\}$$

$$[(p \land \neg A(q)) \Rightarrow S; T; A(q)]$$

$$= \{[\neg A(q) \Rightarrow (p = A(p))]\}$$
\[ (A(p) \land \neg A(q)) \Rightarrow S; T; A(q) \]
\[ = \{ A(p) \ \text{unless}_{CM} \ A(p) \ \text{in} \ S; T \ \text{and definition} \} \]
\[ A(p) \ \text{ensures}_{CM} A(p) \ \text{in} \ S; T \]
\[ \Rightarrow \{ \text{lemma 4.29} \} \]
\[ A(p) \rightarrow_{*} A(p) \ \text{in} \ S; T \]

This lemma results in the following transformation rule that allows stuttering statements to be removed.

**Transformation 5.26 (Remove stuttering)** Let \( F \) be an ImpUNITY program, and let \( S \) and \( T \) be statements in \( \text{assign}(F) \) such that \( \lceil \text{hide}(F) \rceil \leq T \). Let \( G \) be the ImpUNITY program that only differs from \( F \) in the assign-section:

\[ \text{assign}(G) = \text{assign}(F) \setminus T \cup \{ S; T \}. \]

Then, \( F \subseteq G \).

**Proof:** Let \( A \overset{\text{def}}{=} (\text{skip} \lor T_L) \), then this transformation is a direct consequence of lemmas 5.21, 5.25, and 5.20.

---

**Program** \( F \)
- hide \( y \)
- init \( y = 0 \)
- assign \( x := 0 \mid y \in \{ i \mid 0 \leq i \leq 10 \} \mid y > 0 \rightarrow y := y - 1 \)
- end\( \{ F \} \)

**Program** \( G \)
- hide \( y \)
- assign \( x := 0 \mid y \in \{ i \mid 0 \leq i \leq 9 \} \)
- end\( \{ G \} \)

---

**Example 5.3.**

Consider the programs \( F \) and \( G \) in example 5.3. Program \( G \) is a refinement of \( F \) by transformation 5.26 in which the second statement of \( F \) is scheduled after the first statement of \( F \).

**5.3.2 Refining Statements**

In this subsection we show that theorem 5.20 can be used to refine statements of a program through an abstraction.

As a stepping stone to the transformation rule we prove that refinement of statements through some abstraction implies refinement of properties through that abstraction.
Lemma 5.27 Let $F$ and $G$ be ImpUNITY programs such that $\text{assign}(F) = \{ S_i \mid i \in I \}$ and $\text{assign}(G) = \{ S'_i \mid i \in I \}$. Let $A$ be an abstraction such that

$$S_i \leq_A S'_i, \text{ for all } i \in I,$$

then

$$p \text{ unless}_{CM} q \text{ in } F \Rightarrow A(p) \text{ unless}_* A(q) \text{ in } G,$$

$$p \text{ ensures}_{CM} q \text{ in } F \Rightarrow A(p) \rightarrow_* A(q) \text{ in } G.$$  

Proof:

$$p \text{ unless}_{CM} q \text{ in } F$$

$$= \{ \text{definition} \}$$

$$= \{ \forall i : i \in I : [(p \land \neg q) \Rightarrow S_i(p \lor q)] \}$$

$$= \{ \text{predicate calculus} \}$$

$$= \{ \forall i : i \in I : [p \Rightarrow q \lor S_i(p \lor q)] \}$$

$$= \{ \text{monotonicity} \}$$

$$= \{ \forall i : i \in I : [A(p) \Rightarrow A(q \lor S_i(p \lor q))] \}$$

$$= \{ \text{disjunctivity} \}$$

$$= \{ \forall i : i \in I : [A(p) \Rightarrow A(q) \lor A; S_i(p \lor q)] \}$$

$$= \{ \text{assumption } S_i \leq_A S'_i \}$$

$$= \{ \forall i : i \in I : [A(p) \Rightarrow A(q) \lor S'_i A(p \lor q)] \}$$

$$= \{ \text{predicate calculus, disjunctivity} \}$$

$$= \{ \forall i : i \in I : [(p \land \neg A(q) \Rightarrow S'_i A(p \lor A))] \}$$

$$= \{ \text{definition} \}$$

$$= \{ \text{lemma 4.29} \}$$

$$= \{ A(p) \text{ unless}_{CM} A(p) \text{ in } G \}$$

$$\Rightarrow \{ A(p) \text{ unless}_* A(p) \text{ in } G \}$$

The proof for $\text{ensures}_{CM}$ is similar. 

Next, we look at $(Y \sim Z)$-abstractions, abstractions that we will use for data refinement in section 5.4. Here we only show that they can be used for refinement through a command. A $(Y \sim Z)$-abstraction is based on a so-called abstraction predicate and is defined as follows.
Definition 5.28 Let $Y$ and $Z$ be sets of variables. A predicate $P$ is called an abstraction predicate on $Y$ if $\{\forall y \in Y \} (P)$, and the command

$$A_P \overset{\text{def}}{=} \{\forall y \in Y \}; \{P\}; \{\forall z \}.$$  

is called the $(Y \sim Z)$-abstraction based on $P$.

It is easy to verify that a $(Y \sim Z)$-abstraction is indeed an abstraction, i.e. it is universally disjunctive and terminating. A $(Y \sim Z)$-abstraction is used to transform statements and programs that do not read and write variables in $Z$ to statements and programs do no read and write variables in $Z$. The abstraction predicate $P$ gives the relation between the old variables $Y$ and the new variables $Z$. For example, the abstraction command $A \overset{\text{def}}{=} A_{\forall y \in Y \}; \{y = -z\}; \{\forall z \}$ can be used to transform the statement $x, y := y, y + 1$ to the statement $x, z := -z, z - 1$ by refinement through $A$, i.e. $x, y := y, y + 1 \leq_A x, z := x, z - 1$. As shown by Back [Bac93b], if a statement preserves the abstraction predicate of a $(Y \sim Z)$-abstraction $A$ and respect the views $Y$ and $Z$, then it is refined through $A$ by itself. This is expressed by the following lemma.

Lemma 5.29 Let $A$ be a $(Y \sim Z)$-abstraction based on $P$ and let $M$ be a modifier such that $[P \Rightarrow M(P)]$. For all statements $S$ such that $(S \text{ obey } M)$ and $(S \text{ resp } Y \cup Z)$,

$$S \leq_A S.$$  

Proof: First, $[P \Rightarrow S(P)]$ follows directly from $[P \Rightarrow M(P)]$ and $S \text{ obey } M$. Then:

$$\{\forall y \}; \{P\}; \{\forall z \}; S$$

$$\leq \{S \text{ resp } Y \cup Z\}$$

$$\{\forall y \}; \{P\}; S; \{\forall z \}$$

$$\leq \{\text{assumption}\}$$

$$\{\forall y \}; \{S(P)\}; S; \{\forall z \}$$

$$= \{\text{refinement calculus}\}$$

$$\{\forall y \}; S; \{P\}; \{\forall z \}$$

$$\leq \{S \text{ resp } Y \cup Z\}$$

$$S; \{\forall y \}; \{P\}; \{\forall z \}$$

Hence, $S \leq_A S$. \hfill \Box$

Now, the following lemma states that preservation of properties through a $(Y \sim Z)$-abstraction implies refinement as long as the variables in $Y$ and $Z$ are hidden and predicate $P$ is preserved by the modifier in the external-section. In action systems, this is known as the non-interference condition. This lemma is a direct consequence of theorem 5.20 and lemma 5.29.
Lemma 5.30 Let $F$ and $G$ be ImpUNITY programs that only differ in their initially and assign-sections. Let $A$ be a $(Y \sim Z)$-abstraction that is based on $P$ such that $(Y \cup Z) \subseteq \text{hide}(F)$ and $[P \Rightarrow \text{external}(F)(P)]$. If for all predicates $p$ and $q$,
\[
A(\text{init}(F)) \quad \Leftarrow \quad \text{init}(G),
\]
\[
p \quad \text{unless}_{CM} \quad q \quad \text{in} \quad F \quad \Rightarrow \quad A(p) \quad \text{unless}_{A} \quad A(q) \quad \text{in} \quad G,
\]
\[
p \quad \text{ensures}_{CM} \quad q \quad \text{in} \quad F \quad \Rightarrow \quad A(p) \quad \Rightarrow_{A} \quad A(q) \quad \text{in} \quad G,
\]
then $F \subseteq G$.

This lemma 5.30 and lemma 5.27 result in the following transformation for refinement through a $(Y \sim Z)$-abstraction based on $P$.

Transformation 5.31 (Abstraction) Let $F$ and $G$ be ImpUNITY programs that only differ in their initially and assign-sections. Let $\text{assign}(F) = \{S_i \mid i \in I\}$ and $\text{assign}(G) = \{S'_i \mid i \in I\}$. If for sets of variables $Y$ and $Z$ such that $Y \cup Z \subseteq \text{hide}(F)$ and for some $(Y \sim Z)$-abstraction $A$ based on $P$,
\begin{itemize}
  \item $[P \Rightarrow \text{external}(F)(P)]$,
  \item $[A(\text{init}(F))] \Leftarrow \text{init}(G)$, and
  \item $S_i \leq_{A} S'_i$, for all $i \in I$,
\end{itemize}
then $F \subseteq G$.

As a special case, this rule can be used to refine auxiliary variables $Z$ of a program. This is done by using the $(\emptyset \sim Z)$-abstraction $A = \{\text{aux}_Z\}$ based on true. In this case, the non-interference condition $[P \Rightarrow \text{external}(F)(P)]$ is always satisfied.

5.4 Data Refinement

In the previous section we gave some tools to modify the local variables in a program by introducing refinement through an abstraction. However, local variables are still not really local in the sense that they are independent from the environment: they cannot be renamed or replaced arbitrarily. Let $F$ and $G$ be ImpUNITY programs that only differ in that one variable is renamed. Then, program $F$ is not refined by program $G$ since an environment $H$ of $F$ that reads this variable is not an environment of $G$. In this section we extend the ImpUNITY framework with a transformation that allows fresh variables to be added and removed. This is done by redefining program union in such a way that it renames local variables in a program composition. This is sufficient to obtain transformation rules for renaming local variables, superposition and data refinement. We give two methods of data refinement, a direct and an indirect method.
First, we formally define what we mean by local and fresh variables of a program. A set of variables is called local if the variables can neither be read nor written by the environment, and a set of variables is called fresh if the variables are not read nor written by the program, and the program does not pose any restrictions on the use of the variables.

**Definition 5.32** A set of variables $Y$ is called local to ImpUNITY program $F$ if

\[
Y \subseteq \text{hide}(F), \\
\text{external}(F) \text{ obey } [=y].
\]

A set of variables $Z$ is called fresh to ImpUNITY program $F$ if

\[
Z \cap \text{hide}(F) = \emptyset, \\
[B_Z] \text{ obey } \text{external}(F), \\
\text{init}(F) \text{ resp } Z, \\
F \text{ obey } [=z].
\]

Then, a local renaming is a renaming of a program where local variables are replaced by fresh variables.

**Definition 5.33** Let $Y$ and $Z$ be sets of variables and let $F$ be an ImpUNITY program. ImpUNITY program $G \overset{\text{def}}{=} F[Z/Y]$ is called a renaming of $F$ if $Y$ is local to $F$ and $Z$ is fresh to $F$.

Next, we give a new definition of program union that renames local variables before the programs are actually composed. This gives the possibility to compose programs that have local variables with the same names. The definition is an extension of definition 5.13.

**Definition 5.34** Let $F$ and $G$ be ImpUNITY programs, let $Y_F$ be the set of all local variables of $F$ and let $Y_G$ be the set of all local variables of $G$. The union $F\llbracket G\rrbracket$ is defined by $F\llbracket G\rrbracket \overset{\text{def}}{=} F'[Z_F/Y_F]$ and $G' \overset{\text{def}}{=} G[Z_G/Y_G]$, and $Z_F$ and $Z_G$ are disjoint sets of variables that are fresh to both $F$ and $G$ and have the same number of elements as $Y_F$ and $Y_G$, respectively.

For simplicity, we always rename all local variables in a composition, even when the local variables of both components are already distinct. Otherwise, we would have an extra condition on the variables of the components. Note that the definition of program union is only unique up to local renaming.

A program $H$ is an environment of a program $F$ if $H$ respects the interference and observability restrictions of $F'$ for a suitable local renaming $F'$ of $F$. 
**Definition 5.35** Let $F$ and $H$ be ImpUNITY programs. By $H_{env}$, $F$ we denote that $H$ is an environment after a suitable renaming of $F$:

$$H_{env}, F \overset{\text{def}}{=} (\exists F' : F' \text{ local renaming of } F : H_{env} F').$$

By $H_{cont}$, $F$ we denote that $H$ is a context after renaming of $F$ which is defined

$$H_{cont}, F \overset{\text{def}}{=} F_{env}, H \land H_{env}, F.$$  

Definition 5.17 defines the notion of observable refinement. Again, this notion is lifted to a compositional notion. This is done using the new definitions of program union and environment.

**Definition 5.36** For ImpUNITY programs $F$ and $G$, by $F \subseteq G$ we denote that $G$ is a refinement of $F$ which is defined by

$$F \subseteq G \overset{\text{def}}{=} (\forall H : H_{env}, F : H_{env}, G \land F[H \subseteq_o G], H).$$

It is straightforward to prove that all program transformation rules are still valid. We can also give a rule for adding and removing new local variables. We assume that every ImpUNITY program only reads and writes a finite number of variables and that the set of ImpUNITY variables is infinite. Based on this assumption, we are able to rename local variables of a program with variables that are fresh both to the program and to the environment. It also gives the possibility to extend the local state space of a program with a set of fresh variables or to remove variables that are not read or written by the program. This is expressed in the following transformation rule.

**Transformation 5.37 (Fresh variables)** Let $F$ be an ImpUNITY program and $Z$ a set of variables that is fresh to $F$. Let $G$ be the ImpUNITY program defined by

$$\begin{align*}
\text{hide}(G) & \overset{\text{def}}{=} \text{hide}(F) \cup Z, \\
\text{external}(G) & \overset{\text{def}}{=} \text{external}(F) + [\equiv Z], \\
\text{init}(G) & \overset{\text{def}}{=} \text{init}(F), \\
\text{assign}(G) & \overset{\text{def}}{=} \text{assign}(F).
\end{align*}$$

Then $F \subseteq G$ and $G \subseteq F$.

**Proof:** Let $H$ be an environment of $F$ and let $Z'$ be a set of variables that is fresh to $H$ and $G$. Let $G' \overset{\text{def}}{=} G[Z'/Z]$ be a local renaming of $G$. Then, $H$ is an environment of $G'$ iff $H$ is an environment of $F$. Furthermore, it is straightforward to prove that $F[H]$, $H$ and $G'[H]$ have the same (observable) properties.  

Now, we are ready to formalise data refinement and simulation in the ImpUNITY framework as derived transformations. Data refinement is about the replacement of a
set of abstract variables by a set of concrete variables. In literature, many formulations of data refinement can be found [Wri94, MV94, Hoa92]. We start with a direct method similar to the one presented by Back et al. for the action system formalism [Wri94, MV94]. This method is based on refinement through an command and the goal is to replace local variables $Y$ by local variables $Z$. In the action systems formalism, the local and global variables of a program are mentioned explicitly and actions in an action system are typed. ImpUNITY programs operate on a global state space and local variables are modelled as variables that cannot be read nor written by an environment. This results in a slightly different formulation of abstraction commands but the main idea is the same: an abstraction command transforms a statement using the abstract variables to a statement using the concrete variables. We use $(Y \sim Z)$-abstractions to replace concrete variables $Y$ by abstract variables $Z$. The abstraction predicate on which the $(Y \sim Z)$-abstraction is based is called an abstraction relation and gives a relation between the concrete variables and the abstract variables.

**Transformation 5.38 (Data refinement)** Let $F$ and $G$ be the following ImpUNITY programs

```
Program $F$
hide $X \cup Y$
external $M + [=_{Y}]$
init $initF$
assign $\langle \{ i : i \in I : S_i \} \rangle$
end\{$F$\}

Program $G$
hide $X \cup Z$
external $M + [=_{Z}]$
init $initG$
assign $\langle \{ i : i \in I : S_i' \} \rangle$
end\{$G$\}
```

such that the set of variables $Z \setminus X$ is fresh to $F$ and the set $Y \setminus X$ is fresh to $G$. If for $(Y \sim Z)$-abstraction $A$ based on $P$,

- $[P \Rightarrow (M + [=_{Y \cup Z}])(P)]$,
- $[A(initF) \Leftarrow initG]$, and
- $S_i \leq_A S_i'$, for all $i \in I$,

then $F \sqsubseteq G$.

**Proof:** Let program $F'$ be the program $F$ extended with local state variables $Z \setminus X$ and let $G'$ be the program $G$ extended with local variables $Y \setminus X$. By transformation 5.37 we have $F \sqsubseteq F'$ and $G' \sqsubseteq G$. Furthermore, since for every statement $T$ of a context $H$ of $F'$ both $T \text{ resp } (Y \cup Z)$ and $T \text{ obey } [=_{Y \cup Z}]$ hold, the refinement $T \leq_A T$ follows from the first assumption and lemma 5.29. Then, the refinement $F' \sqsubseteq G'$ follows from transformation 5.31, and the refinement $F \sqsubseteq G$ follows from transitivity. \qed
The data refinement rules generalises the rule for fresh variables 5.37. The \((\emptyset \sim Z)\)-abstraction based on \textit{true} can be used to add fresh variables \(Z\). The \((Y \sim \emptyset)\)-abstraction based on \textit{true} can be used to remove variables \(Y\). Using the same abstractions, data refinement can be used to add or remove superpositions. If both \(Y \subseteq X\) and \(Z \subseteq X\), data refinement specialises to transformation 5.31.

Also the indirect method for data refinement as given in [MV94] can be used in ImpUNITY. In this method, the abstraction relation \(P\), giving a relation between concrete and abstract variables, is called a \textit{coupling invariant}. Normally, the coupling invariant is a predicate on variables \(Y\) and \(Z\), giving the relation between the abstract local variables and the concrete local variables. We only require non-interference, i.e. \([P \Rightarrow M + \{\exists Z\}(P)]\), that is, the coupling invariant can not be falsified by an environment. The method consists of three steps. First, the concrete variables \(Z\) are added to the program as a superposition in such a way that the coupling invariant \(P\) is established. Second, the coupling invariant is a local invariant that is used to modify the \textit{initially} and the \textit{assign}-sections of the program. Goal of this this transformation is to make the abstract variables \(Y\) auxiliary. Finally, the auxiliary abstract variables are removed. As can be easily verified, all these steps are instantiations of transformation 5.38.

For refinement of action systems, Back and von Wright give a rule for simulation [BvW94]. This rule is a data refinement rule that abstracts from stutterings and allows stutterings to be rescheduled. The rule of simulation is not a proper refinement for ImpUNITY programs since fairness properties are not preserved. However, we can give a similar rule which follows directly from earlier transformation rules.

\textbf{Transformation 5.39 (Weak simulation)} Let \(F\) and \(G\) be the following ImpUNITY programs

\begin{align*}
\text{Program } F & \quad \text{Program } G \\
\text{hide } X \cup Y & \quad \text{hide } X \cup Z \\
\text{external } M + [=Y] & \quad \text{external } M + [=Z] \\
\text{init } \text{init}F & \quad \text{init } \text{init}G \\
\text{assign } \langle [i : i \in I : S_i] \rangle & \quad \text{assign } \langle [i : i \in I : S'_i] \rangle \\
\text{end}\{F\} & \quad \text{end}\{G\}
\end{align*}

such that the set of variables \(Z \setminus X\) is fresh to \(F\) and the set \(Y \setminus X\) is fresh to \(G\). If for \((Y \sim Z)\)-abstraction \(A\) based on \(P\) and for a statement \(T\)

\begin{itemize}
\item \([P \Rightarrow (M + [=Y \cup Z])(P)]\),
\item \((((\land_{i \in I} S_i) \land \text{skip}) \lor [\exists \text{hide}(F)])) \leq T\),
\item \([A; (T^*)_L(\text{init}F) \Leftarrow \text{init}G]\),
\end{itemize}
• $S_i; T^* \leq_A S'_i$, for all $i \in I$, and

• $((\bigwedge_{i \in I} S_i) \land \text{skip}); T^* \leq S''_j$, for all $j \in J$,

then $F \sqsubseteq G$.

**Proof:** First, for every $j \in J$ a copy of the statement $((\bigwedge_{i \in I} S_i) \land \text{skip})$ is added to program $F$ by transformation 4.35. Second, the $\text{initially}$-section is rewritten to $(T^*)_I(\text{init}F)$ and all statements are transformed to $S; T^*$ by transformation 5.24. Then, by application of the data refinement rule (transformation 5.38) program $G$ is obtained. □

The rule for data refinement 5.38 is a special case of weak simulation in which $T = \text{skip}$ and $J = \emptyset$. Furthermore, for $A = \text{skip}$ and $J = \emptyset$, weak simulation specialises to rescheduling of stutterings as given in transformation 5.24, and if $A = \text{skip}$, $T = \text{skip}$ and $J$ consists of one element, the rule for adding a statement 4.35 is obtained.

### 5.5 Refinement of Modules

The notion of refinement as discussed in the sections 4.2 and 5.2 can be used for refinement in an environment. i.e. if $F \sqsubseteq G$, then for every environment $H$ of $F$ we have $F[H] \sqsubseteq G[H]$. We would like to see programs as modules or components that can be refined independently, i.e. for program $F$ and $G$ such that $G \text{ cont } F$, we want that if $F \sqsubseteq F'$ and $G \sqsubseteq G'$, then also the refinement $F[G \sqsubseteq F'][G']$ holds. We conjecture that this property holds for our notion of program refinement, but we did not find a proof yet. We outline the main difficulties. The property does hold if $F'$ is an environment of $G$. In that case, $F'[G \sqsubseteq F'][G']$, and since $G \text{ env } F$ the refinement $F[G \sqsubseteq F'][G]$ holds. A similar result holds if $G'$ is an environment of $F$. However, $F'$ does not have to be an environment of $G$ as can can be seen from example 5.4. The refinement $F \sqsubseteq F'$ follows

<table>
<thead>
<tr>
<th>Program $F$</th>
<th>Program $F'$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>external</strong> $[={y}]$</td>
<td><strong>external</strong> $[={y}]$</td>
</tr>
<tr>
<td><strong>init</strong> $y = 0$</td>
<td><strong>init</strong> $y = 0$</td>
</tr>
<tr>
<td><strong>assign</strong></td>
<td><strong>assign</strong></td>
</tr>
<tr>
<td>$y := y + 1$</td>
<td>$y := y + 1$</td>
</tr>
<tr>
<td>$\mid x := \text{true} \land x := \text{false}$</td>
<td>$\mid y &lt; 0 \rightarrow z_1 := 1$</td>
</tr>
<tr>
<td><strong>end</strong>{$F$}</td>
<td>$\mid x := (z_2 = 10)$</td>
</tr>
<tr>
<td><strong>end</strong>{$F'$}</td>
<td></td>
</tr>
</tbody>
</table>

**Example 5.4.**
from transformation 4.37 using the local invariant $\text{linvariant}_s(y \geq 0)$ in $F$. Let $G$ be an ImpUNITY program with output variable $z_1$, i.e. $[=z_1] \leq \text{external}(G)$, and input variable $z_2$, i.e. $z_2 \in \text{hide}(G)$. Now, $F'$ is not an environment of $G$ for two reasons. First, $F'$ does not obey the interference restriction of $G$ since it contains a statement that may change variable $z_1$. Second, it does not respect to observability restriction of $G$ since it reads variable $z_2$ that is hidden in $G$.

A solution is to extend the ImpUNITY programming language with two extra sections giving additional restrictions on ImpUNITY programs. (In the action system formalism a similar approach is taken: by typing the actions, the set of variables that may be read and written by an action are given explicitly.)

- The **internal-section**, containing a modifier that must be obeyed by the program. For a program $F$, this set is denoted by $\text{internal}(F)$.

- The **read-section**, containing the set of all variables that can be read by the program. For a program $F$, this set is denoted by $\text{read}(F)$. We require that $F$ respects the view $(\text{read}(F))^c$, the complement of the set $\text{read}(F)$ in the set of all ImpUNITY program variables.

Moreover, we need stronger definitions of environment and context. Instead of requiring that an environment respects the set of hidden variables, we require that hidden variables do not appear in the *read-section* of the environment. We also require that the *internal-section* obeys the interference restriction.

**Definition 5.40** For ImpUNITY programs $F$ and $H$, define

- $H \text{ env } F \overset{\text{def}}{=} \text{external}(F) \leq \text{internal}(H) \land \text{hide}(F) \cap \text{read}(H) = \emptyset$,

- $H \text{ cont } F \overset{\text{def}}{=} F \text{ env } H \land H \text{ env } F$.

As a direct consequence of the definitions, an environment $H$ of $F$ obeys the interference restriction of $F$ and respects the observability restriction of $F$.

**Lemma 5.41** For ImpUNITY programs $F$ and $H$,

$$H \text{ env } F \Rightarrow H \text{ obey } \text{external}(F) \land H \text{ resp } \text{hide}(F).$$

The definition of program union is extended and takes the new sections into account. For simplicity, we do not look at the possibility of renaming local variables, but the work of section 5.4 can be easily lifted. A union of programs may read variables that may be read by one of the components. The actions are restricted by the demonic composition of the *internal-sections.*
Definition 5.42 For ImpUNITY programs \( F \) and \( G \), the union \( F \mid G \) is defined by

\[
\begin{align*}
\text{hide}(F \mid G) & \overset{\text{def}}{=} (\text{hide}(F) \cup \text{hide}(G)), \\
\text{read}(F \mid G) & \overset{\text{def}}{=} (\text{read}(F) \cup \text{read}(G)), \\
\text{external}(F \mid G) & \overset{\text{def}}{=} (\text{external}(F) + \text{external}(G)), \\
\text{internal}(F \mid G) & \overset{\text{def}}{=} (\text{internal}(F) \land \text{internal}(G))^*, \\
\text{init}(F \mid G) & \overset{\text{def}}{=} (\text{init}(F) \land \text{init}(G)), \\
\text{assign}(F \mid G) & \overset{\text{def}}{=} (\text{assign}(F) \cup \text{assign}(G)).
\end{align*}
\]

We have to check that the definition of program union is a proper definition, i.e. that composition yields an ImpUNITY program. The \textit{read}-section of an ImpUNITY program gives a view that must be respected by the union programs and the \textit{internal}-section gives a restriction on the action that must be obeyed by the union. The following lemma, which is straightforward to prove, states that this requirement is satisfied.

Lemma 5.43 For ImpUNITY programs \( F \) and \( G \),

\[
\begin{align*}
F \mid G \text{ obey internal}(F \mid G), \\
F \mid G \text{ resp } (\text{read}(F \mid G))^c.
\end{align*}
\]

It is possible to examine all components separately when checking the environment conditions. This lifts lemma 4.15.

Lemma 5.44 For ImpUNITY programs \( F \), \( G \) and \( H \),

\[
\begin{align*}
H \text{ env } F \mid G & = H \text{ env } F \land H \text{ env } G, \\
F \mid G \text{ env } H & = F \text{ env } H \land G \text{ env } H.
\end{align*}
\]

The notion of \textit{observable refinement} corresponds to the preservation of observable properties as given in definition 5.16. Then, definition 5.17 of refinement is lifted using the new definition of environment as follows.

Definition 5.45 Let \( F \) and \( G \) be ImpUNITY programs. By \( F \sqsubseteq G \) we denote that \( G \) is a refinement of \( F \) which is defined by

\[
F \sqsubseteq G \overset{\text{def}}{=} (\text{read}(F) \supseteq \text{read}(G)) \land (\text{internal}(F) \leq \text{internal}(G)) \land (\forall H : H \text{ env } F : H \text{ env } G \land F[H \sqsubseteq_o G[H]).
\]

The compositionality result of lemma 4.18 still holds.

Lemma 5.46 For ImpUNITY programs \( F \), \( G \) and \( H \), such that \( H \text{ env } F \),

\[
F \sqsubseteq G \Rightarrow F \mid H \sqsubseteq G[H].
\]
Moreover, a refinement $G$ of $F$ is an environment of a program $H$ if $F$ is an environment of $H$. This is expressed by the following lemma.

**Lemma 5.47** For ImpUNITY programs $F$, $G$ and $H$,

$$ F \text{ env } H \land F \subseteq G \Rightarrow G \text{ env } H. $$

As a consequence of lemma 5.46 and lemma 5.47, components or modules may be refined independently. This is expressed by the following theorem.

**Theorem 5.48** For ImpUNITY programs $F$, $F'$, $G$ and $G'$,

$$ G \text{ cont } F \land F \subseteq F' \land G \subseteq G' \Rightarrow F|G \subseteq F'|G'. $$

**Proof:**

$$ G \text{ cont } F \land F \subseteq F' \land G \subseteq G' $$

$$ \Rightarrow \{\text{definition } 5.40, \text{ lemma } 5.47\} $$

$$ G \text{ env } F \land G \text{ env } F' \land F \subseteq F' \land G \subseteq G' $$

$$ \Rightarrow \{\text{lemma } 5.46\} $$

$$ F|G \subseteq F'|G \land F'|G \subseteq F'|G' $$

$$ \Rightarrow \{\text{transitivity}\} $$

$$ F|G \subseteq F'|G' $$

Now, consider the programs $F$ and $F'$ of example 5.5. These programs are extensions of the programs given in example 5.4 with read and internal-sections. The internal-section of program $F$ states that $F$ is allowed to write variables $x$, $y$ and $z_1$ in any environment. The read-section of program $F$ states that $F$ is allowed to read variables $x$, $y$ and $z_2$ in any environment. Program $F'$ is a refinement of $F$ and since the external, internal, hide and read of $F$ and $F'$ are the same, every environment of $F$ is also an environment of $F'$, despite the fact that this refinement introduces a write action on $z_1$ and a read action on $z_2$. Moreover, $F'$ is a refinement of $F$ in any context $H$ of $F$ even when that context is refined at the same time, i.e. if $H \subseteq H'$ then $F|H \subseteq F'|H'$.

### 5.6 Conclusions

In this chapter we introduced observability in the ImpUNITY framework. We gave definitions of observability on the level of predicates, predicate transformers and ImpUNITY programs. The introduction of observability lead to a new notion of program refinement.
A number of program transformation rules were given for the rescheduling of stutterings. In combination with restrictions on interference, data refinement can be handled in the ImpUNITY framework. Moreover, we showed how programs can be seen as modules that can be refined independently.
Chapter 6

Case Study: Register Refinement

In this chapter we give the outline of a case study that shows how a communication register can be refined to another, more detailed register. The refinement consists of a number of steps based on transformation rules presented in the previous chapters. Hence, each step preserves $\text{unless}_O$ and $\rightarrow_O$ properties of the program in any environment.

In a paper [Lam86] on the nature of asynchronous communication, Lamport divides communication acts into two types, transient and persistent. Transient communication corresponds to message-passing, where messages are seen as consumable events; persistent communication is concerned with objects (variables, shared memory) that can be read and written. The difficulties one encounters with persistent communication are related to the low-level semantics of reading and writing in the context of concurrency and limited availability of atomic operations. The objects manipulated by read and write operations are called registers, which are studied in [Mis86]. Registers can be classified by a number of parameters, including the types of operations they admit, the number of concurrent readers and writers that are possible, the size of the register, and how they behave under concurrent access. For instance, a register is called safe if it satisfies the following weak property of concurrent access: provided no two register operations are concurrent, a read operation returns the latest value written to the register; however if a read operation is concurrent with a write operation, then that read operation may return an arbitrary value in the domain of the register. A fundamental question in the study of registers is: what kind of register can be implemented in terms of (a collection of) other kinds of registers? A particular implementation that answers this question is therefore a kind of refinement, and a natural question is to investigate the nature of such a refinement. Our case study examines one such register implementation: the construction of a single-writer, $2n$-bit, safe register using two single-writer, $n$-bit, safe registers.

Previous case studies showed how UNITY applies to transient communication by modelling distributed programs with channels [CM88, San90]. We believe that it is also important to study registers, because they have different atomicity properties than chan-
nels or even simple UNITY assignment statements. For instance, problems of stuttering and fairness are intrinsic in a setting with no low-level atomic operations. Another way to see this is that a writer cannot “know” when a potential reader has effectively read a value being written, and the write operation cannot be implemented by a single action. To our knowledge, register communication is a concept that has not been studied before in the UNITY framework.

In Section 6.1 we show how safe registers can be modelled in the ImpUNITY framework and we describe the goal of refinement. The refinement process is given in section 6.2.

### 6.1 Registers in ImpUNITY

We model the behaviour of a safe register of domain $T$ by the program $Safe$, given in example 6.1. Program $Safe$ has two variables: $in$ and $out$. Variable $out$ represents the output of the register to the environment (i.e. users wishing to read), and reading the register consists of reading the value of variable $out$. The $external$-section of program $Safe$ states that variable $out$ cannot be written by the environment (although we do not use this fact in this example). Variable $in$ represents the input provided by the environment of $Safe$ (i.e. the user wishing to write). The protocol for the environment to write the register is as follows. The special value $\bot (\bot \notin T)$ is used to indicate that $in$ is empty and the register is ready to be written. By writing a (non-$\bot$) value in the variable $in$, the environment starts a write operation. This write operation terminates when the register program resets $in$ to $\bot$. The $external$-section states that the environment may only write variable $in$ with a value of type $T$ and it may only write it when $in = \bot$. So, predicate $in = \bot$ states that the register is waiting for input and predicate $in \neq \bot$ states that a write operation is pending.

**Program** $Safe$

```
external ([\{in.out\}] \land (in = \bot \rightarrow in : \in T))^*
init out : T
assign in \neq \bot \rightarrow out : \in T
  | in \neq \bot \rightarrow out, in := in, \bot
end{Safe}
```

**Example 6.1.**

Program $Safe$ consists of two statements handling the write operation. The first statement models the following characteristic of a safe register: during a write operation,
the output can be an arbitrary value. The second statement makes the input available for the environment and terminates the write operation. By the fairness requirement, this statement will eventually be executed, so, every write operation terminates. This is expressed by the property \( \text{in} \neq \bot \Rightarrow \text{in} = \bot \) \( \text{in} \in \text{Safe} \). Moreover, for any value \( k \in T \), property \( \text{in} = k \Rightarrow \text{in} = \bot \wedge \text{out} = k \) \( \text{in} \in \text{Safe} \) states that each write operation terminates with variable \( \text{out} \) containing the value written. When no write operation is pending, the output does not change spontaneously, which is expressed by the following property \( \text{stable}_{\text{a}}(\text{in} = \bot \Rightarrow \text{out} = k) \) \( \text{in} \in \text{Safe} \).

Note that the interface between the environment and \( \text{Safe} \) is atomic since variables \( \text{in} \) and \( \text{out} \) are read and written in an atomic way; presumably, for this case study to make sense, the variables \( \text{in} \) and \( \text{out} \) are interfaces with different components of the environment that cannot themselves atomically communicate.

The goal of refinement is to obtain a program that implements a \( 2n \)-bit safe register using two \( n \)-bit safe registers. We assume that variable \( \text{out} \) is an array consisting of two parts \( \text{out}[1], \text{out}[2] \) of type \( T_1 \) and \( T_2 \) (for example, one can see it as a bit-string which can be divided into two parts), and \( \text{in} \) is similarly split into two parts. Each part is then implemented by (smaller) safe registers, \( \text{Safe}_1 \) or \( \text{Safe}_2 \), that are instances of \( \text{Safe} \) with smaller buffers, i.e. the program \( \text{Safe} \) with variables renamed: \( \text{Safe}_1 \overset{\text{def}}{=} \text{Safe}[	ext{in}_1, \text{out}[1], T_1 / \text{in}, \text{out}, T] \) and \( \text{Safe}_2 \overset{\text{def}}{=} \text{Safe}[	ext{in}_2, \text{out}[2], T_2 / \text{in}, \text{out}, T] \).

Program \( \text{Com} \), as given in example 6.2, is a communication interface that reads communication variable \( \text{in} \) and distributes it to the inputs of \( \text{Safe}_1 \) and \( \text{Safe}_2 \).

\[ \text{Program Com} \]

<table>
<thead>
<tr>
<th>hide ( w_1, w_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>external ( (\exists {\text{in}_1, \text{in}_2}) \wedge (\text{in}_1 := \bot) \wedge (\text{in}_2 := \bot) \wedge (\text{in} = \bot \Rightarrow \text{in} \in T)^* )</td>
</tr>
<tr>
<td>init ( w_1 = w_2 = \text{false} )</td>
</tr>
<tr>
<td>assign</td>
</tr>
<tr>
<td>( \text{in} \neq \bot \wedge \neg w_1 \Rightarrow \text{in}_1, w_1 := \text{in}[1], \text{true} )</td>
</tr>
<tr>
<td>( \text{in} \neq \bot \wedge \neg w_2 \Rightarrow \text{in}_2, w_2 := \text{in}[2], \text{true} )</td>
</tr>
<tr>
<td>( \text{in}_1 = \text{in}_2 = \bot \wedge w_1 \wedge w_2 \Rightarrow \text{in}, w_1, w_2 := \bot, \text{false}, \text{false} )</td>
</tr>
<tr>
<td>end{Com}</td>
</tr>
</tbody>
</table>

\[ \text{Example 6.2.} \]

The register then consists of the union of three components, namely two safe registers and a component that distributes inputs over the two safe registers. These components communicate by variables \( \text{in}_1 \) and \( \text{in}_2 \). To prevent that an environment reads or writes these variables, they must be local. The external-sections of the three components state
that an environment may not write \( in_1 \) and \( in_2 \). By hiding the variables, they cannot be read by an environment either. Hence, the goal is to refine \( Safe \) to the program \( hide(\{in_1, in_2\} : Com | Safe_1 | Safe_2) \).

### 6.2 Refinement of a Register

In this section we give an overview of the refinement process of the implementation of program \( Safe \) by two safe registers and a communication component. The process consists of applications of the program transformation rules as given in the previous chapters.

The first step is to refine program \( Safe \) in example 6.1. In this step we introduce one smaller safe register \( Safe_1 \). This is done in five refinement steps:

\[
Safe \sqsupseteq S_1 \sqsubseteq S_2 \sqsubseteq S_3 \sqsubseteq S_4 \sqsubseteq hide(\{in_1\} : Com_1 | Safe_1).
\]

Program \( Com_1 \) is a combination of the communication interface and the safe register; it forwards a part of the input to \( Safe_1 \) and deals with the second half of the input itself. The second step is to refine \( Com_1 \). In this step we introduce the second safe register \( Safe_2 \) and obtain a program of the form \( hide(\{in_2\} : Com | Safe_2) \). Composition of these programs results in the program \( hide(\{in_1, in_2\} : Com | Safe_1 | Safe_2) \). We start by examining the refinement steps to obtain \( Safe_1 \) in more detail.

\[Safe \sqsupseteq S_1\] (Superposition, transformation 5.38)

---

**Program \( S_1 \)**

\[
\begin{align*}
\text{hide} & \hspace{1em} in_1, w_1 \\ 
\text{external} & \hspace{1em} ([=\{in, out, in_1, w_1\}] \land (in = \bot \rightarrow in : \in T))^* \\ 
\text{init} & \hspace{1em} w_1 = false \land out \in T \\ 
\text{assign} & \hspace{1em} in \neq \bot \land \neg w_1 \rightarrow in_1, w_1 := in[1], true \\
& \hspace{1.5em} | \hspace{1em} in \neq \bot \rightarrow out : \in T \\
& \hspace{1.5em} | \hspace{1em} (in \neq \bot \rightarrow out : \in T); (in_1 \neq \bot \rightarrow in_1 := \bot) \\
& \hspace{1.5em} | \hspace{1em} in \neq \bot \rightarrow out : \in T \\
& \hspace{1.5em} | \hspace{1em} in \neq \bot \rightarrow out, in, w_1 := in, \bot, false \\
\end{align*}
\]

**Example 6.3.**

In the first step the new local variables of register \( Safe_1 \) are introduced together with the construction of writing this register. We superpose input variable \( in_1 \) of
6.2. Refinement of a Register

Safe₁ and the control variable $w₁$ of $Com$. A part of variable $out$ serves as the output variable of program $Safe$. The behaviour of $Safe$ on variable $out$ is obtained by refining (copies of) statement $in \neq \bot \to out \in T$. This gives program $S₁$ as given in example 6.3.

$S₁ \sqsubseteq S₂$ (Rewriting using a local invariant, transformation 4.37)

---

Program $S₂$

```
hide $in₁, w₁$
external $([=\{in, out, in₁, w₁\}] \land (in = \bot \to in \in T))^*$
init $w₁ = false \land out \in T$
assign $in \neq \bot \land \neg w₁ \to in₁, w₁ := in[1], true$
| $in₁ \neq \bot \to out[1] \in T₁$
| $in₁ \neq \bot \to out[1], in₁ := in₁, \bot$
| $in \neq \bot \to out \in T$
| $in \neq \bot \to out, in, w₁ := in, \bot, false$
```

Example 6.4.

In the second step we reduce the non-determinism. The second and third statement of program $S₁$ were introduced to model the statements of register $Safe₁$. These statements can change variable $out$ arbitrarily, while $Safe₁$ only writes a part of this variable. We reduce the non-determinism of these statements by only assigning an arbitrary value to $out[1]$ using the local invariant $linvariant_{out} \in T$ which implies that the value of $out[2]$ is of type $T₂$ so does not need to be changed. This yields the program $S₂$ as given in example 6.4. The first statement of program $S₂$ corresponds to writing register $Safe₁$ and the second and third statement model register $Safe₁$.

$S₂ \sqsubseteq S₃$ (Rewriting using a local invariant, transformations 4.37, and strengthening a guard, transformation 4.38)

We introduced the statements of register $Safe₁$ but the rest of the program does not satisfy the external-section of this register: the last two statements of $S₂$ write the output variable $out[1]$. In the fourth statement, this reference can be easily removed by reducing the non-determinism of this statement. For the last statement the situation is more complicated. This statement terminates a write operation. This can be done at any moment, even when the write operation on $Safe₁$ has not been finished. In this step we postpone the termination of a write operation on the whole register until the termination of the write operation on $Safe₁$. This is
Program $S_3$

hide $in_1, w_1$

external $(\{in, out, in_1, w_1\} \land (in = \bot \rightarrow in : \in T))^*$

init $w_1 = false \land out \in T$

assign $in \neq \bot \land \neg w_1 \rightarrow in_1, w_1 := in[1], true$

| $in_1 \neq \bot \rightarrow out[1] \in T_1$
| $in_1 \neq \bot \rightarrow out[1], in_1 := in_1, \bot$
| $in \neq \bot \rightarrow out[2] \in T_2$
| $in \neq \bot \land w_1 \land in_1 = \bot \rightarrow out, in, w_1 := in, \bot, false$

end${S_3}$

Example 6.5.

done by strengthening the guard of the last statement of $S_2$. Control variable $w_1$ indicates that a write operation on $Safe_1$ has been started and $in_1 = \bot$ states that no write operation on $Safe_1$ is pending. So, predicate $q \overset{\text{def}}{=} (w_1 \land in_1 = \bot)$ states that $Safe_1$ has been written and, therefore, we strengthen the guard $in \neq \bot$ by $q$. This yields program $S_3$ as given in example 6.5. The conditions for strengthening guards are easy to verify:

$[(w_1 \land in_1 = \bot \land in \neq \bot) \Rightarrow external(S_3)((w_1 \land in_1 = \bot) \lor in = \bot)]$,

$(w_1 \land in_1 = \bot) \text{ unless}_{CM} (in = \bot)$,

$(in \neq \bot) \Rightarrow_{CM} (in = \bot \lor (w \land in_1 = \bot))$.

$S_3 \subseteq S_4$ (Rewriting using a local invariant, transformation 4.37)

In this step we modify the assignment to variable $out$ in the last statement to remove the reference to $out[1]$. As a consequence of the previous refinement, the last statement can only terminate a write operation when the write operation on $Safe_1$ has terminated and $out[1]$ has been written properly. This is expressed by the local invariant

$linvariant_s(w_1 \land in_1 = \bot \Rightarrow out[1] = \bot)$.

Using this invariant, the assignment $out := \bot$ in the last statement can be rewritten to $out[2] := \bot$. Moreover we simplify the guard of this statement using the local invariant

$linvariant_s(w_1 \Rightarrow in \neq \bot)$

and remove the conjunct $in \neq \bot$. This yields program $S_4$ as given in example 6.6.
Program $S_4$

hide $in_1, w_1$

external $((\equiv \{in, out, in_1, w_1\} \land (in = \bot \rightarrow in : \in T))^* \land \forall x) \land (w_1 = false \land out[1] \in T_1 \land out[2] \in T_2)$

init $w_1 = false \land out[1] \in T_1 \land out[2] \in T_2$

assign $\forall x$

| $in \neq \bot \land \neg w_1 \rightarrow in_1, w_1 := in[1], true$
| $in \neq \bot \rightarrow out[1] \in T_1$
| $in \neq \bot \rightarrow out[1], in_1 := in_1, \bot$
| $in \neq \bot \rightarrow out[2] \in T_2$
| $w_1 \land in_1 = \bot \rightarrow out[2], in, w_1 := in[2], \bot, false$

end{$S_4$}

Example 6.6.

$$S_4 \subseteq \text{hide}(\{in_1\} : \text{Com}_1|\text{Safe}_1)$$

We have implicitly introduced the program $\text{Safe}_1$ into the code of $S_4$: it consists of the second and third statement of $S_4$. We can decompose program $S_4$ into two components; the $n$-bit safe register $\text{Safe}_1$ and a part $\text{Com}_1$ as given in example 6.7: we have that $S_4 = \text{hide}(\{in_1\} : \text{Com}_1|\text{Safe}_1)$.

Program $\text{Com}_1$

hide $w_1$

external $((\equiv \{in, out[2], w_1\} \land (\forall x) \land (in = \bot \rightarrow in : \in T))^* \land \forall x) \land (w_1 = false \land out[2] \in T_2)$

init $w_1 = false \land out[2] \in T_2$

assign $\forall x$

| $in \neq \bot \land \neg w_1 \rightarrow in_1, w_1 := in[1], true$
| $in \neq \bot \rightarrow out[2] \in T_2$
| $w_1 \land in_1 = \bot \rightarrow out[2], in, w_1 := in[2], \bot, false$

end{$\text{Com}_1$}

Example 6.7.

In the same way as we refined $\text{Safe}$ to a program containing the register $\text{Safe}_1$, we can refine $\text{Com}_1$ to introduce $\text{Safe}_2$ or actually $\text{hide}(\{in\} : \text{Com} | \text{Safe}_2)$. Then we have:

$$\text{Safe}$$
\[
\text{hide}\{i_{1}\} : \text{Com}_{1}[\text{Safe}_{1}]
\]
\[
\subseteq
\]
\[
\text{hide}\{i_{1}\} : (\text{hide}\{i_{2}\} : \text{Com}[\text{Safe}_{2}])[\text{Safe}_{1}]
\]
\[
\subseteq
\]
\[
\text{hide}\{i_{1}, i_{2}\} : \text{Com}[\text{Safe}_{1}]\text{Safe}_{2}
\]

Hence, the 2n-bit register Safe can be implemented by two n-bit registers Safe₁ and Safe₂ and a component forwarding inputs to both registers.

### 6.3 Conclusions

In this chapter we presented the outline of a case study of the refinement of register communication in the ImpUNITY framework. This case study serves two goals. First, it illustrates the use of the ImpUNITY framework as presented in the previous chapters. It shows how a component can be modelled and refined in this framework and how the program transformation rules are applied. Second, it shows that the concept of register communication can be handled, and hence, also persistent communication can be handled in this framework.
Chapter 7

Procedures

In this chapter we extend the ImpUNITY framework with a remote procedure call mechanism. For simplicity, we omit the read and internal-sections for modular refinement here. But the strategy of section 5.5 can be easily lifted.

The procedure mechanism is a powerful concept in programming. It is one of the main ways to structure programs and it improves the readability and size of the program text. For refinement of programs, the procedure mechanism offers a way to refine repeated specifications at one place. Our main reasons for the introduction of procedures are, however, different. First, by coding external interference by procedure calls, more variables can be seen as local variables of a component. So, the procedure mechanism can be used to enlarge the possibilities for compositional reasoning. Second, Back and Sere showed that procedures can be used to model synchronous communication in the action system formalism [BS94a]. This also holds for the ImpUNITY framework, and the introduction simplifies the mappings from ImpUNITY programs to target architectures with synchronous communication.

The organisation of this chapter is as follows. Section 7.1 deals with procedures in the refinement calculus, and in section 7.2 we introduce the (remote) procedure call mechanism in the ImpUNITY framework. Section 7.3 is about refinement of programs containing procedures. It is shown that most program transformation rules remain valid and that a modification of the data refinement rule can be given.

7.1 Procedures

The procedure mechanism is an important programming concept and therefore it has been extensively studied. In the refinement calculus, the procedure mechanism has been formalised both by Back [Bac87] and by Morgan [Mor90]. In this section we give a short overview of their work and we examine the consequences for the notions of obeying a modifier and respecting a view.
The procedure mechanism consists of three aspects: procedure declarations, procedure definitions and procedure calls. We identify all items by the name of the procedure. A procedure declaration, also called a prototype of the procedure, provides the interface. A declaration defines the name of the procedure and the type of its argument. For example, a procedure \texttt{name} with an argument of type \texttt{Type} is declared by

\begin{verbatim}
proc name(Type).
\end{verbatim}

We consider only a restricted form of procedures. First, we examine only procedures with a single argument. This is not a severe restriction since procedures with multiple arguments can be modelled by using argument tuples. Second, we examine only one way of parameter passing, namely, pass by value. This restriction is made to keep the programming logic simple. Furthermore, it corresponds to a form of synchronous communication in which one party sends a message that is received by the other.

A procedure definition gives the body a procedure. It consists of a procedure declaration in which a name is given to the parameter, and a (terminating, non-miraculous, and conjunctive) statement called the body of the procedure. In the body, the name of the parameter may be used as a variable. For example,

\begin{verbatim}
proc name(a : int) = x := a^2; a := a + a; y := a^2.
\end{verbatim}

We do not allow procedures to call procedures, so we do not allow recursion. The procedure declaration induced by a procedure definition is called a procedure header.

A procedure call consists of the name of the procedure and an expression (a function on the state space) of the proper type, i.e. the type of the argument given in the procedure declaration. Execution of a procedure call corresponds to the execution of the body of the procedure in which the parameter variable is a local variable initialised to the argument of the call. For example, for the procedure \texttt{name} above, the call \texttt{name}(3x + 1) corresponds to the execution of the statement

\begin{verbatim}
|Var| a = 3x + 1. x := a^2; a := a + a; y := a^2|.
\end{verbatim}

In case the procedure definition is known, a call can be resolved by substituting the call by its corresponding statement.

We extend the language of commands to a language of commands with procedure calls as follows:

\begin{verbatim}
B ::= A | name(e) | \bigwedge_{i \in I} B_i | \bigvee_{i \in I} B_i | B_1; B_2,
\end{verbatim}

where \texttt{A} is a command (without procedure calls) and \texttt{name} is a procedure. For a set of procedure definitions \texttt{N}, and a command \texttt{B} containing calls to procedures in \texttt{N}, by \texttt{B[N]} we denote the command in which calls to procedures in \texttt{N} have been resolved.
A command with procedure calls is a statement if we can prove that it is terminating, non-miraculous, and conjunctive under the assumption that all procedure calls are statements.

For a procedure \textit{name} defined by
\[
\text{proc } \textit{name}(a : \textit{Type}) = S,
\]
we can approximate all possible calls to \textit{name} by statement \(\overline{\textit{name}}\) which is defined by
\[
\overline{\textit{name}} \overset{\text{def}}{=} \land_{e \in \text{Type}} [\text{var } a = e. S].
\]

Then, for a set of procedure definitions \(N\), a call to an arbitrary procedure in \(N\) with an arbitrary argument is modelled by \(\overline{N}\) and is defined by
\[
\overline{N} \overset{\text{def}}{=} \land_{\textit{name} \in N} \overline{\textit{name}}.
\]

The notion of obeying a modifier is extended to commands with procedures calls and (sets of) procedure definitions as follows. Let \(M\) be a modifier. We defined that a statement or command \(A\) without procedure calls obeys \(M\) if \(A \text{ obey } M = M \leq A\). A command \(B\) with procedure calls obeys \(M\) if \(B\) is constructed from commands obeying \(M\) and procedure calls, i.e.
\[
\begin{align*}
\textit{name}(e) \text{ obey } M &= \text{true}, \\
(\land_{i \in I} B_i) \text{ obey } M &= \langle \forall i : i \in I : B_i \text{ obey } M \rangle, \\
(\lor_{i \in I} B_i) \text{ obey } M &= \langle \forall i : i \in I : B_i \text{ obey } M \rangle, \\
(B_1; B_2) \text{ obey } M &= (B_1 \text{ obey } M) \land (B_2 \text{ obey } M).
\end{align*}
\]

For a procedure definition we define \((\text{proc } n(a : \textit{Type}) = S) \text{ obey } M \overset{\text{def}}{=} \overline{\pi} \text{ obey } M\). For a a set of procedure definitions \(N\) we define \(N \text{ obey } M \overset{\text{def}}{=} \langle \forall n : n \in N : n \text{ obey } M \rangle\).

The notion of respecting a view is extended to commands with procedures calls and sets of procedure definitions as follows. For a view \(O\) and a statement or command \(A\) without procedure calls we defined \(A \text{ resp } O = \langle \forall O' : O' \subseteq O : A \leq_{\{O',\}} A \rangle\). Like for interference, we define that a command \(B\) with procedure calls respects \(O\) if \(B\) is constructed from commands and procedure calls respecting \(O\), and a procedure call \(\textit{name}(e) \text{ respects } O\) if the initialisation of the block of the call respects \(O\):
\[
\begin{align*}
\textit{name}(e) \text{ resp } O &= (a := e) \text{ resp } O, \\
(\land_{i \in I} B_i) \text{ resp } O &= \langle \forall i : i \in I : B_i \text{ resp } O \rangle, \\
(\lor_{i \in I} B_i) \text{ resp } O &= \langle \forall i : i \in I : B_i \text{ resp } O \rangle, \\
(B_1; B_2) \text{ resp } O &= (B_1 \text{ resp } O) \land (B_2 \text{ resp } O).
\end{align*}
\]

For a procedure definition we define \((\text{proc } n(a : \textit{Type}) = S) \text{ resp } O \overset{\text{def}}{=} \overline{\pi} \text{ resp } O\). For a a set of procedure definitions \(N\) we define \(N \text{ resp } O \overset{\text{def}}{=} \langle \forall n : n \in N : n \text{ resp } O \rangle\).
Let $S$ be a statement with calls to procedures in a set $N$. If both $S$ and $N$ obey a modifier $M$, then also the statement $S[N]$ obeys $M$. A similar result holds for the notion of respecting a view. This is expressed by the next lemma which follows from lemma 4.3 and the fact that refinement through a command commutes with sequential and demonic composition [Wri94].

**Lemma 7.1** Let $N$ be a set of procedure definitions and $S$ a statement containing calls to procedures in $N$. For modifier $M$ and view $O$,

$$S \text{ obey } M \land N \text{ obey } M \Rightarrow S[N] \text{ obey } M,$$

$$S \text{ resp } O \land N \text{ resp } O \Rightarrow S[N] \text{ resp } O.$$  

### 7.2 Procedures in ImpUNITY

Morgan shows that procedures can be used in modules as a way to encapsulate data [Mor90]. For action systems Back and Sere showed that procedures are useful for modelling synchronous communication [BS94a, BS94b]. We follow their work and add a procedure call mechanism to the ImpUNITY framework in a similar way. First, we introduce the remote procedure call mechanism that is used for communication between components. A program can export procedures, and these procedures may be called by an environment. The program itself may call procedures that are imported from the context. Second, we introduce a local procedure call mechanism that allows programs to use their own procedures. Then, we look at program union and examine the consequences of the procedure mechanism for the notion of program refinement.

Remote procedures are incorporated into the ImpUNITY programming language by the following two sections containing procedure declarations and definitions.

- The *import-section*, containing declarations of procedures that are imported by the program. Definitions of these procedures must be provided by an environment. This set of procedure declarations is denoted by $\text{import}(F)$.

- The *export-section*, defining procedures that are exported by the program. The set of procedure definitions in the *external-section* is denoted by $\text{export}(F)$ and the set of procedure headers induced by this section is denoted by $\text{headers}(F)$. We assume that the *import-section* and the *export-section* of a program are disjoint, i.e. that they do not declare procedures with the same name.

We allow statements in the *assign-section* of a program to contain remote procedure calls, i.e. calls to procedures in the *import-section*. Calls to these procedures can only be resolved by composing the program with another program exporting the procedures.

The second step is to introduce a local procedure call mechanism. We extend the language with a section containing definitions of local procedures.
7.2. Procedures in ImpUNITY

- The proc-section, defining local procedures that can only be called by the program itself. The set of procedure definitions in the proc-section is denoted by $\text{proc}(F)$. The names of the procedures declared in this section must be different from the names of the procedures declared in the import-section and the export-section.

We allow statements in the assign-section of a program to contain local procedure calls, i.e. calls to procedures in the export and proc-sections. Since the definitions of these procedures are known, these calls can be resolved.

Declarations and definitions in the import, export or proc-section are separated by $|$, and we use the convention that if the section is empty it is not mentioned explicitly.

We lift the notions of obeying a modifier and respecting a view to ImpUNITY programs with procedures by requiring that all components, including the procedure bodies, satisfy the requirements. Here, we assume that the statements in the program do not contain calls to local procedures. This is not a real restriction, because the definitions of these procedures are known and the calls can be resolved.

**Definition 7.2** Let $F$ be an ImpUNITY program, $M$ a modifier and $O$ a view. Define

\[
\begin{align*}
F \text{ obey } M & \overset{\text{def}}{=} (\text{export}(F) \text{ obey } M) \land (\forall S : S \in \text{assign}(F) : S \text{ obey } M), \\
F \text{ resp } O & \overset{\text{def}}{=} (\text{init}(F) \text{ resp } O) \land (\text{export}(F) \text{ resp } O) \\
& \land (\forall S : S \in \text{assign}(F) : S \text{ resp } O).
\end{align*}
\]

Now, the definitions of environment and context are the same as before, using the new notions of obeying and respecting.

**Definition 7.3** For ImpUNITY programs $F$ and $H$, by $F \text{ env } H$ we denote that $H$ is an environment of $F$ which is defined by

\[
H \text{ env } F \overset{\text{def}}{=} H \text{ obey } \text{external}(F) \land H \text{ resp } \text{hide}(F).
\]

By $H \text{ cont } F$ we denote that $H$ is a context of $F$ which is defined

\[
H \text{ cont } F \overset{\text{def}}{=} F \text{ env } H \land H \text{ env } F.
\]

Consider program Buf given in example 7.1. This program models a buffer that communicates with its environment by a procedure interface. Messages are put in the buffer by calling the procedure $\text{flush}_1$ and the buffer outputs messages by calling a procedure $\text{flush}_0$. Messages are stored in the local variables $b_1$ and $b_2$ that are modelled as lists. We use the standard list notation: $\text{[]}$ denotes the empty list, $++$ denotes list concatenation, and $\text{hd}$ and $\text{tl}$ are the functions on lists that yield the head respectively the tail of a list. Output is taken care of by the second statement of program Buf. The first statement removes a message from local variable $b_1$, in which input messages are stored and, by
calling the local procedure $\text{flush}_m$, puts the message in local variable $b_2$, in which output messages are stored.

We give two operations on ImpUNITY programs to transform remote procedures to local procedures. First, the hiding operator restricts the set of exported procedures of a program. This prevents an environment from calling these procedures and restricts the interaction of an environment. Hiding consists of moving a set of procedures from the export-section to the proc-section.

**Definition 7.4** For ImpUNITY program $F$ and a set of procedure declarations $N$, program $\text{proc}(N : F)$ only differs from $F$ in the export and proc-sections:

- $\text{headers}(\text{proc}(N : F)) = \text{headers}(F) \setminus N$,
- $\text{export}(\text{proc}(N : F)) \cup \text{proc}(\text{proc}(N : F)) = \text{export}(F) \cup \text{proc}(F)$.

Second, we give a way to resolve imported procedures of a program. This is done by providing definitions of these procedures.

**Definition 7.5** Let $F$ be an ImpUNITY program and let $N$ be a set of procedure definitions. Program $F[N]$ is the program that only differs from $F$ in the import and proc-sections:

- $\text{import}(F[N]) \overset{\text{def}}{=} \text{import}(F) \setminus N$,
- $\text{proc}(F[N]) \overset{\text{def}}{=} \text{proc}(F) \cup N$.

The remote procedure mechanism provides a way of interaction between programs, i.e. components can call procedures of other components. In the next definition program
union is extended to deal with procedures. For simplicity, we only look at programs without local procedures, which can always be resolved.

**Definition 7.6** For ImpUNITY programs $F$ and $G$, the union $F \mid G$ is defined by

- $\text{hide}(F \mid G) \overset{\text{def}}{=} (\text{hide}(F) \cup \text{hide}(G))$
- $\text{external}(F \mid G) \overset{\text{def}}{=} (\text{external}(F) + \text{external}(G))$
- $\text{export}(F \mid G) \overset{\text{def}}{=} (\text{export}(F) \cup \text{export}(G))$
- $\text{import}(F \mid G) \overset{\text{def}}{=} (\text{import}(F) \cup \text{import}(G)) \setminus (\text{headers}(F) \cup \text{headers}(G))$
- $\text{init}(F \mid G) \overset{\text{def}}{=} (\text{init}(F) \wedge \text{init}(G))$
- $\text{assign}(F \mid G) \overset{\text{def}}{=} (\text{assign}(F) \cup \text{assign}(G))$

Not all ImpUNITY programs with procedures can be composed. If one component exports a procedure that is imported by the other, the declarations of the procedure must be the same in both components. Furthermore, if a procedure is imported by both components, then the declarations of the procedure must be the same in both components; and if a procedure is exported by both components, then the definitions of the procedure must be the same in both components.

Program $\text{Buf}$ in example 7.1 is an ImpUNITY program modelling a buffer. It can be seen as a program that consists of two internal buffers, $b_1$ and $b_2$. In example 7.2 two ImpUNITY programs, $\text{Buf}_1$ and $\text{Buf}_2$, are given, and it is easy to verify that program $\text{Buf}$ is a kind of union of $\text{Buf}_1$ and $\text{Buf}_2$: $\text{Buf} = \text{proc}(\{\text{flush}_m\} : \text{Buf}_1 \mid \text{Buf}_2)$.

---

**Example 7.2.**

Again, we can define a notion of program union in which local variables can be renamed.
Definition 7.7 Let $F$ and $G$ be ImpUNITY programs, let $Y_F$ be the set of all local variables of $F$ and let $Y_G$ be the set of all local variables of $G$. The union $F \cup G$ is defined by $F \cup G \overset{\text{def}}{=} F' \cup G'$, where $F' \overset{\text{def}}{=} F[Z_F/Y_F]$ and $G' \overset{\text{def}}{=} G[Z_G/Y_G]$, and $Z_F$ and $Z_G$ are disjoint sets of variables that are fresh to both $F$ and $G$ and have the same number of elements as $Y_F$ and $Y_G$, respectively.

If we allow local variables to be renamed before composition, we have to use a notion of environment that allows renaming too. Therefore, we lift definition 5.35 of environment after renaming $env$, and context after renaming $cont$, by to programs with procedures, by using the modified definition of environment and context.

An ImpUNITY program without procedures can be seen as a closed system. This is not the case for an ImpUNITY program that imports procedure calls; it needs some environment to provide these procedures. A program that can run in isolation, i.e. a program which $\text{import}$-section is empty, is called a full program.

Definition 7.8 An ImpUNITY program $F$ is called a full program if $\text{import}(F) = \emptyset$.

If we use the convention that all calls to procedures in the $\text{export}$ and $\text{proc}$-sections are resolved, then a full program does not contain procedure calls. For this kind of programs we defined the notion of observable refinement in definition 5.36: preservation of observable properties.

Again, we want to have a compositional notion of program refinement. Since observable program refinement is about full programs, it is only interesting to examine environments for which the union result in a full program. Such an environment is called a full environment.

Definition 7.9 For ImpUNITY programs $F$ and $H$, by $F \overset{\text{env}_F}{\text{env}} H$ we denote that $H$ is an full environment of $F$ which is defined by

$$H \overset{\text{env}_F}{\text{env}} F \overset{\text{def}}{=} (H \overset{\text{env}_F}{\text{env}} F) \land \text{import}(F \cup H) = \emptyset.$$ 

By $H \overset{\text{cont}_F}{\text{cont}} F$ we denote that $H$ is a full context of $F$ which is defined by

$$H \overset{\text{cont}_F}{\text{cont}} F \overset{\text{def}}{=} F \overset{\text{env}}{\text{env}} H \land H \overset{\text{env}}{\text{env}} F.$$ 

Then, we define refinement of ImpUNITY programs as observable refinement in any full environment.

Definition 7.10 For ImpUNITY programs $F$ and $G$, by $F \subseteq G$ we denote that $G$ is a refinement of $F$ which is defined by

$$F \subseteq G \overset{\text{def}}{=} (\forall H : H \overset{\text{env}_F}{\text{env}} F : H \overset{\text{env}_F}{\text{env}} G) \land F[H \subseteq G] \overset{\text{def}}{=} (H \overset{\text{env}_F}{\text{env}} F \land H \overset{\text{env}_F}{\text{env}} G).$$
Now, we lift the ImpUNITY logic to programs with procedures. External interference of an ImpUNITY program is modelled by the modifier $\text{inter}(F)$. For an ImpUNITY program $F$ without procedures all interference is modelled by the $\text{external}$-section, i.e. $\text{inter}(F) = \text{external}(F)$. For ImpUNITY programs with procedures this notion must be modified. Statements of an environment may contain calls to procedures exported by $F$. Therefore, for a program $F$ we define the external interference by

$$\text{inter}(F) \overset{\text{def}}{=} (\text{external}(F) \land \text{export}(F))^*.$$  

Next, we consider statements that contain calls to imported procedures. The bodies of imported procedures are not known and may differ for different environments. However, we can approximate the body of the call by giving a lower bound. By this restriction, calls to procedures imported from an environment must obey the external interference restriction, i.e. for every call $\text{name}$ we know that $\text{name}$ obey $\text{external}(F)$ holds. This gives a way to calculate properties of a program; we can substitute $\text{external}(F)$ for each call to an imported procedure. For example, for program $\text{Buf}$ of example 7.1, property $i \in b_1 \Rightarrow i \in b_2$ holds.

Because only a lower bound is used and the actual bodies of the procedures are not known, the resulting logic is rather weak and will in general not be strong enough for the full specification of components. Properties in the logic only refer to state transitions and there is no way to reason about procedure calls. Nevertheless, the logic is strong enough to express interesting properties of specific components. This is important in the design process for getting a proper intuition of the program and the program development. Moreover, we can use this logic for the formulation of program transformation rules.

### 7.3 Program Transformation Rules

In this section we give program transformation rules for ImpUNITY programs with procedures. We lift most of the program transformation rules given in the previous chapters to programs with procedures. We also give a modification of the data refinement rule that allows bodies of exported procedures to be refined.

First, we look at programs without restrictions on observability and we lift the program transformation rules that are given in section 4.4. Let $F$ and $G$ be ImpUNITY programs with procedures, and suppose that we want to use a transformation rule of section 4.4 to show the refinement $F \sqsubseteq G$. Since the rule only changes the $\text{assign}$-section we know that

\[
\begin{align*}
\text{import}(F) &= \text{import}(G), \\
\text{export}(F) &= \text{export}(G), \\
\text{external}(F) &= \text{external}(G).
\end{align*}
\]
Let $F'$ be the ImpUNITY program defined by
\[
\begin{align*}
\text{external}(F') & \overset{\text{def}}{=} \text{inter}(F), \\
\text{init}(F') & \overset{\text{def}}{=} \text{init}(F), \\
\text{assign}(F') & \overset{\text{def}}{=} \text{assign}(F).
\end{align*}
\]
Since $F$ and $F'$ have the same \textit{initially} and \textit{assign}-sections, both programs have the same UNITY properties. Furthermore, since $\text{inter}(F') = \text{inter}(F)$, both programs have the same ImpUNITY properties. Let $G'$ be defined from $G$ in the same way as $F'$ is defined from $F$.

Let ImpUNITY program $H$ be a full environment of $F$ and let $H'$ be defined from $H$ in the same way as $F'$ was defined from $F$. Then, exported procedures of $H$ are refinements of $\text{external}(F)$, and exported procedures of $F$ are refinements of $\text{external}(H)$, i.e. both $\text{export}(H)$ obey $\text{external}(F)$ and $\text{export}(F)$ obey $\text{external}(H)$. Consequently, $H'[\text{export}(F)]$ is an environment of $F'[\text{export}(H)]$. Then,
\[
\begin{align*}
F[H] \sqsubseteq O G[H]
= & \{\text{union}\} \\
F[\text{export}(H)][H[\text{export}(F)] \sqsubseteq O G[\text{export}(H)][H[\text{export}(G)]
= & \{\text{external-section}\} \\
F'[\text{export}(H)][H[\text{export}(F)] \sqsubseteq O G'[\text{export}(H)][H[\text{export}(G)]
= & \{\text{export}(F) = \text{export}(G)\} \\
F'[\text{export}(H)][H[\text{export}(F)] \sqsubseteq O G'[\text{export}(H)][H[\text{export}(F)]
= & \{\text{definition}\} \\
F'[\text{export}(H)] \sqsubseteq G'[\text{export}(H)]
\end{align*}
\]
So, $F[H]$ is observably refined by $G[H]$ if $F'[\text{export}(H)]$ is refined by $G'[\text{export}(H)]$. Suppose that all procedure calls in $F$ and $G$ obey the modifier $\text{external}(F)$ and that we can prove that programs $F$ and $G$ satisfy the condition of the transformation. Then we can also prove that $F'$ and $G'$ satisfy the condition and, by the transformation rule, we have proven that $F'[N] \sqsubseteq G'[N]$ for any set of procedure definitions $N$ such that $N$ obey $\text{external}(F)$. Since $\text{export}(H)$ satisfies this condition, we have shown the refinement $F[H] \sqsubseteq O G[H]$ for every full environment $H$ of $F$, so, the refinement $F \sqsubseteq G$ holds. Hence, the program transformation rule also applies to ImpUNITY programs with procedures.

Transformation rule 4.37 allows statements of a program to be refined using a local invariant. A local invariant can also be used to rewrite the bodies of exported procedures.
(that are not called by the program itself). Since, the statements of an environment are built from statements the external-section of a program $F$, a local invariant of $F$ holds continuously during the execution of that statement; also at the moment that a procedure is called. Therefore, transformation 4.37 can be generalised to as follows.

**Transformation 7.11 (Local invariant)** Let $F$ and $G$ be ImpUNITY programs that do not call exported procedures, that only differ in their export and assign-sections, and

\[
\begin{align*}
\text{export}(F) &= \{ \text{proc name}_j(a : \text{Type}_j) = T_j \mid j \in J \}, \\
\text{export}(G) &= \{ \text{proc name}_j(a : \text{Type}_j) = T'_j \mid j \in J \}, \\
\text{assign}(F) &= \{ S_i \mid i \in I \}, \\
\text{assign}(G) &= \{ S'_i \mid i \in I \}.
\end{align*}
\]

If for some predicate $r$

- $\text{linvariant}_r \text{ in } F$,
- $\{ r \}; T_j \leq T'_j$, for all $j \in J$, and
- $\{ r \}; S_i \leq S'_i$, for all $i \in I$,

then $F \sqsubseteq G$.

For programs that do have calls to exported procedures, these calls can be resolved first. In subsection 5.3.1, we gave two transformation rules about the rescheduling of stuttering statements of a program $F$. These rules are based on the fact that variables hidden by $F$ are also hidden in any composition. In a similar way as above, we can lift these program transformation rules to ImpUNITY programs with procedures.

Also transformation 5.3.1 for abstraction through a command can be lifted. However, this rule needs some modification. Consider the case that we want to refine $F$ by $G$ through an abstraction based on $P$. The condition for non-interference, i.e. $[P \Rightarrow \text{external}(F)(P)]$ is used to, prove that each statement of an environment is refined through $A$ by themselves. In case the environment calls exported procedures of $F$ this is not always true. Since executions of these calls can read and write local variables of $F$ they are not necessarily refined through $A$ by itself. Therefore, also the bodies of exported procedures need to be refined through this abstraction.

**Transformation 7.12 (Abstraction)** Let $F$ and $G$ be ImpUNITY programs that only differ in their export, initially and assign-sections. Let

\[
\begin{align*}
\text{assign}(F) &= \{ S_i \mid i \in I \}, \\
\text{assign}(G) &= \{ S'_i \mid i \in I \}, \\
\text{export}(F) &= \{ \text{proc name}_j(a : \text{Type}_j) = T_j \mid j \in J \}, \\
\text{export}(G) &= \{ \text{proc name}_j(a : \text{Type}_j) = T'_j \mid j \in J \}.
\end{align*}
\]
If for sets of variables $Y$ and $Z$ such that $Y \cup Z \subseteq \text{hide}(F)$ and for some $(Y \sim Z)$-abstraction $A$ based on $P$,

- $[P \Rightarrow \text{external}(F)(P)]$,
- $[A(init(F)) \Leftarrow init(G)]$,
- $T_j \leq_A T'_j$, for all $j \in J$, and
- $S_i \leq_A S'_i$, for all $i \in I$,

then $F \subseteq G$.

In section 5.4 data refinement is introduced. For ImpUNITY programs with procedures this can be done in the same way, and this results in a rule for data refinement and a rule for simulation. Like the transformation for refinement through an abstraction, also these rules require that procedure bodies are refined. This results in the following transformation rules.

**Transformation 7.13 (Data refinement)** Let $F$ and $G$ be the following ImpUNITY programs

**Program $F$**
- hide $X \cup Y$
- external $M + [=Y]$
- import $Imp$
- export
  - $\{ \text{proc } n_j(a : \text{Type}_j) = T_j \mid j \in J \}$
  - init $initF$
  - assign $\langle [i : i \in I : S_i] \rangle$
- end$\{F\}$

**Program $G$**
- hide $X \cup Z$
- external $M + [=Z]$
- import $Imp$
- export
  - $\{ \text{proc } n_j(a : \text{Type}_j) = T'_j \mid j \in J \}$
  - init $initG$
  - assign $\langle [i : i \in I : S'_i] \rangle$
- end$\{G\}$

such that $Z \setminus X$ is fresh to $F$ and $Y \setminus X$ is fresh to $G$. If for $(Y \sim Z)$-abstraction $A$ based on $P$

- $[P \Rightarrow (M + [= Y \cup Z])(P)]$,
- $[A(initF) \Leftarrow initG]$,
- $T_j \leq_A T'_j$, for all $j \in J$, and
- $S_i \leq_A S'_i$, for all $i \in I$,

then $F \subseteq G$. 
The rule for weak simulation is as follows.

**Transformation 7.14 (Weak Simulation)** Let F and G be the following ImpUNITY programs

Program F

```
hide X ∪ Y
external M + [= Y]
import Imp
export
{proc n_i(a : Type_j) = T_j | j ∈ J},
init initF
assign {i : i ∈ I : S_i}
end{F}
```

Program G

```
hide X ∪ Z
external M + [= Z]
import Imp
export
{proc n_j(a : Type_j) = T'_j | j ∈ J},
init initG
assign {i : i ∈ I : S'_i}
end{G}
```

such that Z \ X is fresh to F and Y \ X is fresh to G. If for (Y ∼ Z)-abstraction A based on P and for a statement T

- \[ P \Rightarrow (M + [= Y \cup Z])(P), \]
- \[ ((\Lambda_{i \in I} S_i \text{ ANDskip}) \lor [\geq_{\text{hide}(F)}]) \leq T, \]
- \[ [A; (T^*)_L(initF) \Leftarrow initG], \]
- \[ T_j \leq_A T'_j, \text{ for all } j \in J, \]
- \[ S_i; T^* \leq_A S'_i, \text{ for all } i \in I, \text{ and} \]
- \[ ((\Lambda_{j \in I} S_j) \land \text{skip}); T^* \leq S''_i, \text{ for all } i \in I', \]

then \( F \subseteq G \).

In section 5.5 we showed how we could adapt the framework for refinement of modules. This modification can be directly lifted to the ImpUNITY framework with procedures.

### 7.4 Conclusions

In this chapter we introduced a procedure call mechanism into the ImpUNITY framework. It enhances the possibilities for compositional reasoning about ImpUNITY programs. Procedures are introduced as a generalization of the standard ImpUNITY framework. Program transformations can be lifted and also the ImpUNITY logic can be used. Although, the ImpUNITY logic obtained is not very powerful, it is suitable for modelling program refinement. In the next chapter we work out a case study based on the use of procedures.
Chapter 8

Case Study: A Memory Component

In chapter 6 we gave the outline of a case study on the refinement of a safe register. In this chapter we present the outline of another case study on persistent communication; we deal with a memory component. This memory component is implemented by a memory that communicates via a remote procedure call component. At the Dagstuhl Seminar on Specification and Refinement of Reactive Systems [Dag94], organised by Lamport and Broy, this case study has been treated using different formalisms. In this chapter we examine the solution in the ImpUNITY framework. To describe the case study and the goals of the refinement we quote the original Specification Problem.

This chapter is organised as follows. Section 8.1 deals with the specification of a procedure interface between components. It defines some of the domains used in this chapter. In section 8.2 a memory component is specified and we show that a reliable memory is a refinement of an unreliable memory. Section 8.3 gives a remote procedure call component and section 8.4 shows that an unreliable memory can be implemented by a combination of a reliable memory and a remote procedure component.

8.1 The Procedure Interface

About the procedure interface, the Specification Problem says:

The problem calls for the specification and verification of a series of components. Components interact with one another using a procedure-calling interface. One component issues a call to another, and the second component responds by issuing a return. A call is an indivisible (atomic) action that communicates a procedure name and a list of arguments to the called component. A return is an atomic action issued in response to a call. There are two kinds of returns; normal and exceptional. A normal call returns a value (which could be a list). An exceptional return also returns a value, usually
indicating some error condition. An exceptional return of a value \( e \) is called raising exception \( e \). A return is issued only in response to a call. There may be “syntactic” restrictions on the types of arguments and return values.

A component may contain multiple processes that can concurrently issue procedure calls. More precisely, after one process issues a call, other processes can issue calls to the same component before the component issues a return from the first call. A return action communicates to the calling component the identity of the process that issued the corresponding call.

In the ImpUNITY framework, components are modelled by programs. A component providing services exports a call procedure and imports a return procedure of the calling component. A call communicates the identity of the calling process, a procedure name and a list of arguments. A return communicates the identity of the process and a value. We do not distinguish normal and exceptional returns, normal and exceptional values are just different items of the same domain. So, we use the following types for the arguments of call and return procedures:

\[
\text{Call Type} \triangleq \text{ProcId} \times \text{Names} \times \text{Args}, \\
\text{Ret Type} \triangleq \text{ProcId} \times \text{Values},
\]

where ProcId is the domain of process identities, Names is the domain of procedure names, Values is the domain of argument values, and Args the domain of lists of Values.

Selection of a field of a tuple is denoted by the variable name followed by a dot and the name of the field in the tuple. For a variable \( a : \text{Call Type} \), the fields are named by \( a = (a.proc, a.name, a.args) \). For a variable \( a : \text{Ret Type} \), the fields are named by \( a = (a.proc, a.ret) \). For lists we have the standard list operations cons (\( : \)), head (\( \text{hd} \)) and tail (\( \text{tl} \)), the length \# giving the length of a list, and the selector \( .n \) that selects the \( n^{th} \) item in the list.

To handle calls, components have an array of slots as a local variable. A slot is a variable of the type

\[
\text{Slot Type} \triangleq \text{Tstat} \times \text{ProcId} \times \text{Names} \times \text{Args} \times \text{Values},
\]

and is used for storing status information, the argument and the value to return. For a slot \( S : \text{Slot Type} \), fields are named by \( S = (S.stat, S.proc, S.name, S.args, S.ret) \). On a call, a free slot is chosen to store the information of the call. With every slot a set of statements is associated dealing with the call. The built-in UNITY fairness takes care of the progress for each call. A consequence of this approach is that only a fixed number of processes can be served.
8.2 A Memory Component

About the memory component, the Specification Problem says:

The component to be specified is a memory that maintains the contents of a set MemLocs of locations. The contents of a location is an element of a set MemVals. This component has two procedures, described informally below. Note that being an element of MemLocs or MemVals is a “semantic” restriction, and cannot be imposed solely by syntactic restrictions on the types of arguments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>loc : an element of MemLocs</td>
</tr>
<tr>
<td>Return Value</td>
<td>an element of MemVals</td>
</tr>
<tr>
<td>Exceptions</td>
<td>BadArg: argument loc is not an element of MemLocs.</td>
</tr>
<tr>
<td></td>
<td>MemFailure: the memory cannot be read.</td>
</tr>
<tr>
<td>Description</td>
<td>Returns the value stored in address loc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>loc : an element of MemLocs</td>
</tr>
<tr>
<td></td>
<td>val : an element of MemVals</td>
</tr>
<tr>
<td>Return Value</td>
<td>some fixed value</td>
</tr>
<tr>
<td>Exceptions</td>
<td>BadArg: argument loc is not an element of MemLocs, or</td>
</tr>
<tr>
<td></td>
<td>argument val is not an element of MemVals.</td>
</tr>
<tr>
<td></td>
<td>MemFailure: the write might not have succeeded.</td>
</tr>
<tr>
<td>Description</td>
<td>Stores the value val in address loc.</td>
</tr>
</tbody>
</table>

The memory must eventually issue a return for every Read and Write call.

Define an operation to consist of a procedure call and the corresponding return. The operation is said to be successful iff it has a normal (non-exceptional) return. The memory behaves as if it maintains an array of atomically read and written locations that initially all contain the value InitVal, such that:

- An operation that raises a BadArg exception has no effect on the memory.
- Each successful Read(l) operation performs a single atomic read to location l at some time between the call and return.
- Each successful Write(l, v) operation performs a sequence of one or more atomic writes of value v to location l at some time between the call and return.
Each unsuccessful Write(l, v) operation performs a sequence of zero or more atomic writes of value v to location l at some time between the call and return.

A variant of the Memory Component is the Reliable Memory Component, in which no MemFailure exceptions can be raised.

**Problem 1** (a) Write a formal specification of the Memory component and of the Reliable Memory component.

(b) Either prove that a Reliable Memory component is a correct implementation of a Memory component, or explain why it should not be.

**Solution of problem 1(a)** The memory component uses the procedure interface as described in section 8.1. So, it exports a procedure call, it imports a return procedure and has a local variable Slot.

The call procedure stores its argument in a free slot, i.e. a slot which status is FREE and changes the status of this slot to CALL. With every slot two actions are associated. The first chooses the value to return and updates a local memory variable in the proper way. The second takes care of issuing the return and frees the slot. The built-in ImpUNITY fairness takes care of the progress: every call has a return.

In the specification of the memory component, we use the following abbreviations:

\[
\begin{align*}
Read(name, args) & \quad \text{def} \quad name = \text{Read} \land \#(args) = 1, \\
Write(name, args) & \quad \text{def} \quad name = \text{Write} \land \#(args) = 2, \\
IsMemCall(name, args) & \quad \text{def} \quad \text{Read}(name, args) \lor \text{Write}(name, args), \\
IsRead(name, args) & \quad \text{def} \quad \text{Read}(name, args) \\
& \quad \phantom{=} \land (\exists l : l \in \text{MemLocs} : args = [l]), \\
IsWrite(name, args) & \quad \text{def} \quad \text{Write}(name, args) \\
& \quad \phantom{=} \land (\exists l, v : l \in \text{MemLocs}, v \in \text{MemVals} : args = [l, v]), \\
BadArgs(name, args) & \quad \text{def} \quad \text{IsMemCall}(name, args) \\
& \quad \phantom{=} \land \neg (\text{IsRead}(name, args) \lor \text{IsWrite}(name, args)).
\end{align*}
\]

Now, the Memory component Mem is specified in example 8.1.

The Reliable Memory component Rel is similar to the memory component but it cannot fail, i.e. it does not return a MemFailure. This is modelled by the program Rel in example 8.2.
Program \textit{Mem} \\
hide \{\text{Slot, mem}\} \\
external \{\text{Slot, mem}\} \\
export \\
\text{proc} \text{call}(a : \text{CallType}) = \\
\quad \text{IsMemcall}(a.\text{name}, a.\text{args}) \\
\quad \rightarrow \text{if} [\text{Slot}[i].\text{stat} = \text{FREE} \\
\quad \rightarrow \text{Slot}[i] := (\text{CALL}, a.\text{proc}, a.\text{name}, a.\text{args}, \text{MemFailure}) \\
\fi \\
\text{import} \text{proc} \text{return}(\text{RetType}) \\
\text{init} (\forall i : i \in \text{MemLocs} : \text{mem}[i] = \text{InitVal}) \land (\forall i : i \in \text{SlotLocs} : \text{Slot}[i].\text{stat} = \text{FREE}) \\
\text{assign} \\
\quad (\forall s : s \in \text{SlotLocs} : \\
\quad \quad \text{Slot}[s].\text{stat} = \text{CALL} \\
\quad \quad \rightarrow \text{if} \text{IsRead} (\text{Slot}[s].\text{name}, \text{Slot}[s].\text{args}) \rightarrow \text{Slot}[s].\text{ret} := \text{mem}[\text{Slot}[s].\text{args}].1] \\\n\quad \quad \mid \text{IsWrite} (\text{Slot}[s].\text{name}, \text{Slot}[s].\text{args}) \rightarrow \text{mem}[\text{Slot}[s].\text{args}].1 := \text{Slot}[s].\text{args}.2 \\\n\quad \quad \mid \text{BadArgs} (\text{Slot}[s].\text{name}, \text{Slot}[s].\text{args}) \rightarrow \text{Slot}[s].\text{ret} := \text{BadArg} \\\n\quad \quad \mid \text{true} \rightarrow \text{Slot}[s].\text{ret} := \text{MemFailure} \\
\quad \fi \\
\quad ) \\
\quad (\forall s : s \in \text{SlotLocs} : \\
\quad \quad \text{Slot}[s].\text{stat} = \text{CALL} \rightarrow \text{return}((\text{Slot}[s].\text{proc}, \text{Slot}[s].\text{ret})) \\\n\quad \quad \quad \quad ; \text{Slot}[s].\text{stat} := \text{FREE} \\
\quad ) \\
\text{end}\{\text{Mem}\}

\begin{example}

\textbf{Solution of problem 1(b)} \quad \text{The Reliable Memory component is a refinement of the Memory component and this refinement is proven in two steps. First, the nondeterminism of the statements in the first set is reduced by transformation 4.37. Second, the guards of the statements in the second set are strengthened by transformation 4.38. We have to prove that the following properties hold in the reliable memory components:}

\begin{align*}
(Slot[s].\text{stat} = \text{CALL}) & \Rightarrow_{\text{CM}} (Slot[s].\text{stat} \neq \text{CALL} \lor \text{ret} \neq \text{MemFailure}), \\
(\text{ret} \neq \text{MemFailure}) & \text{unless}_{\text{CM}} (Slot[s].\text{stat} = \text{CALL}).
\end{align*}

This holds if MemFailure \not\in \text{MemVals}.

\end{example}
Program \( \text{Rel} \)

\[
\begin{align*}
\text{hide} \{ \text{Slot}, \text{mem} \} \\
\text{external} \ = \{ \text{Slot}, \text{mem} \} \\
\text{export} \\
\quad \text{proc} \ \text{call}(a : \text{CallType}) = \\
\qquad \text{IsMemcall}(a, \text{name}, a, \text{args}) \\
\qquad \rightarrow \text{if} [\text{Slot}[i].\text{stat} = \text{FREE} \\
\qquad \quad \rightarrow \text{Slot}[i] := (\text{CALL}, a, \text{proc}, a, \text{name}, a, \text{args}, \text{MemFailure}) \\
\quad \text{fi} \\
\text{import} \ \text{proc} \ \text{return}(\text{RetType}) \\
\text{init} \ (\forall i : i \in \text{MemLocs} : \text{mem}[i] = \text{InitVal}) \land (\forall i : i \in \text{SlotLocs} : \text{Slot}[i].\text{stat} = \text{FREE}) \\
\text{assign} \\
\quad (\{ s : s \in \text{SlotLocs} : \\
\quad \quad \text{Slot}[s].\text{stat} = \text{CALL} \\
\quad \quad \rightarrow \text{if} \ \text{IsRead}(\text{Slot}[s].\text{name}, \text{Slot}[s].\text{args}) \rightarrow \text{Slot}[s].\text{ret} := \text{mem}[\text{Slot}[s].\text{args}.1] \\
\quad \quad \quad \text| \ \text{IsWrite}(\text{Slot}[s].\text{name}, \text{Slot}[s].\text{args}) \rightarrow \text{mem}[\text{Slot}[s].\text{args}.1] := \text{Slot}[s].\text{args}.2 \\
\quad \quad \quad \quad ; \text{Slot}[s].\text{ret} := \text{FixedVal} \\
\quad \quad \quad \text| \ \text{BadArgs}(\text{Slot}[s].\text{name}, \text{Slot}[s].\text{args}) \rightarrow \text{Slot}[s].\text{ret} := \text{BadArg} \\
\quad \quad \text{fi} \\
\quad \} \\
\quad \} \\
\text{end}\{\text{Rel}\}
\]

Example 8.2.

8.3 The RPC Component

About the RPC component, the Specification Problem says:

The RPC component interfaces with two environment components, a sender and a receiver. It relays procedure calls from the sender to the receiver, and relays the return values back to the sender. Parameters of the component are a set \( \text{Proc} \) of procedure names and a mapping \( \text{ArgNum} \), where \( \text{ArgNum}(p) \) is the number of arguments of each procedure \( p \). The RPC component contains a single procedure:
Name: RemoteCall
Arguments: Procs: name of a procedure
            args: list of arguments
Return Value: any value that can be returned by a call to Procs
Exceptions: RPCFailure: the call failed
            BadCall: Procs is not a valid name or args is not a
                      syntactically correct list of arguments for Procs.
Description: Calls procedure Procs with arguments args

A call of RemoteCall(Procs, args) causes the RPC component to do one of the following:

- **Raise a BadCall exception if args is not a list of ArgNum(Procs) arguments.**
- **Issue one call to procedure Procs with arguments args, wait for the corresponding return (which the RPC component assumes will occur) and either (a) return the value (normal or exceptional) returned by that call, or (b) raise the RPCFailure exception.**
- **Issue no procedure call, and raise the RPCFailure exception.**

The component accepts concurrent calls of RemoteCall from the sender, and can have multiple outstanding calls to the receiver.

**Problem 2** Write a formal specification of the RPC component.

**Solution of problem 2** The RPC component is an interface component between two components, a sender and a receiver. It provides a procedure-calling interface to the sender. So, the component exports a call procedure (rpc.call), imports a return procedure (rpc.return) and uses the slot mechanism as described in section 8.1. The RPC component also uses a procedure-calling interface to the receiver. Therefore it imports a call procedure (call) and exports a return procedure (return).

In the description of the RPC component we use the following abbreviations.

\[
\begin{align*}
\text{IsRpcCall}(name, \text{args}) & \triangleq name = \text{RemoteCall} \land \#(\text{args}) > 1 \\
\text{GoodCall}(\text{args}) & \triangleq \text{arg}.1 \in \text{Procs} \land \text{ArgNum}(\text{args}.1) = \#(\text{args}) - 1 \\
\text{BadCall}(\text{args}) & \triangleq \neg \text{GoodCall}(\text{args})
\end{align*}
\]

Now, the RPC component is specified in example 8.3. A call to rpc.call communicates a remote procedure call and this information is stored in a free slot. In case of a bad call
Program \textit{RPC}
\begin{verbatim}
hide \{Slot\}
external \(=\{\text{Slot}\}\)
export
  proc rpc\_call(a : CallType) =
  \text{IsRpcCall}(a.name)
  \rightarrow \text{if} \ i,\text{Slot}[i].\text{stat} = \text{FREE}
  \rightarrow \text{Slot}[i] := (\text{CALL}, a.\text{comp}, a.\text{proc}, a.\text{name}, a.\text{args}, \text{RPCFailure})
  \text{fi}
  \text{proc return}(a : \text{RetType}) = \text{if} \text{true} \rightarrow \text{Slot}[a.\text{proc}].\text{ret} := a.\text{val}
  \quad; \text{Slot}[a.\text{proc}].\text{stat} := \text{RETURN}
  \text{fi}
\text{import} \text{proc} \text{rpc\_return}(\text{RetType})
  \text{proc} \text{call}(\text{CallType})
\text{init} (\forall i : i \in \text{SlotLocs} : \text{Slot}[i].\text{stat} = \text{FREE})
\text{assign}
  (\forall s : s \in \text{SlotLocs} :$
  \quad\text{if} \text{Slot}[s].\text{stat} = \text{CALL}
  \quad\rightarrow \text{if} \text{GoodCall}(\text{Slot}[s].\text{args}) \rightarrow \text{Slot}[s].\text{ret} := \text{CALLING}
  \quad\quad; \text{Slot}[s].\text{stat} := \text{RETURN}
  \quad\quad; \text{Slot}[s].\text{stat} := \text{RETURN}
  \quad\quad\text{fi}
  \quad\text{BadCall}(\text{Slot}[s].\text{args}) \rightarrow \text{Slot}[s].\text{ret} := \text{BadCall}
  \quad\quad; \text{Slot}[s].\text{stat} := \text{RETURN}
  \quad\quad\text{true} \rightarrow \text{Slot}[s].\text{ret} := \text{RPCFailure}
  \quad\quad; \text{Slot}[s].\text{stat} := \text{RETURN}
  \quad\text{fi}
  \quad\text{Slot}[s].\text{stat} = \text{RETURN}
  \quad\rightarrow \text{rpc\_return}((\text{Slot}[s].\text{proc}, \text{Slot}[s].\text{ret}))
  \quad; \text{Slot}[s].\text{stat} := \text{FREE}
  \quad\text{fi}
\end{verbatim}
\text{Example 8.3.}
the second statement of the program decides to return with a BadCall or a RPCFailure by setting the status field to RETURN. On a proper call, the statement forwards the call to the receiver or decides to return with a RPCFailure. The receiver calls the return procedure with the result of the remote call or an exception. The last statement takes care of the return.

8.4 Implementation of the Memory

About implementing the memory, the Specification Problem says:

A Memory component is implemented by combining an RPC component with a Reliable Memory component as follows. A Read or Write call is forwarded to the Reliable Memory by issuing the appropriate call to the RPC component. If this call returns without raising an RPCFailure exception, the value returned is returned to the caller. (An exceptional return causes an exception to be raised.) If the call raises an RPCFailure exception, then the implementation may either reissue the call to the RPC component or raise a MemFailure exception. The RPC call can be retried arbitrarily many times because of RPCFailure exceptions, but a return from the Read or Write call must eventually be issued.

Problem 3 Write a formal specification of the implementation, and prove that it correctly implements the specification of the Memory component of Problem 1.

Solution of problem 3 A memory component is implemented by combining the RPC component, as given in example 8.3, with a (Reliable) Memory component, as given in example 8.2. Besides these two components, the implementation Imp contains an interface component Int, i.e. Imp = proc({rpc_call, rpc_return} : Int[RpcMem]) where RpcMem = proc({call, return} : RPC|Rel). Component Int is an interface component that accepts memory calls and relays them to component RpcMem. Therefore, component Int provides a procedure-calling interface to the environment, it exports a call procedure (call), imports a return procedure (return), and uses the slot mechanism as described before. Furthermore, it uses the procedure-calling interface of the RPC component, i.e. it imports the call procedure (rpc_call) and exports a return procedure (rpc_return).
The \emph{Int} component is specified by program \emph{Int} in example 8.4, where \emph{Map} is the function that maps \emph{RPCFailure} to \emph{MemFailure} and is the identity on all other arguments.
To show that this is a proper implementation of a Memory component we have to prove the refinement $\text{Mem} \sqsubseteq \text{Imp}$. This is done in a number of steps. We start with the program $\text{Mem}$, given in example 8.1, and show it is refined by a program $\text{proc}\{\text{rpc\_call, rpc\_return}\} : \text{Int}[\text{RelMem}']$. Then, program $\text{RelMem}'$ is refined to the program $\text{proc}\{\text{rpc\_call, rpc\_return}\} : \text{RPC}[\text{Mem}]$. In section 8.2, we have proven the refinement $\text{Mem} \sqsubseteq \text{Rel}$, so, we may conclude that $\text{RefMem}' \sqsubseteq \text{RefMem}$ and, consequently, $\text{Mem} \sqsubseteq \text{Imp}$. The complete proof is rather long and can be found in [UK94]. All steps are applications of the refinement rules we have given before.

### 8.5 Conclusions

In this chapter we presented the outline of a case study of the refinement of a memory component in the ImpUNITY framework. First, it illustrates the use of procedures in the ImpUNITY framework as presented in the previous chapter. It shows how a memory can be modelled and refined in this framework and how the program transformation rules are applied. It also shows that refinement in environment is useful. Furthermore, it is another example of modelling persistent communication in this framework.
Appendix A

UNITY Theorems

In this appendix we give an overview of theorems about properties in the UNITY logic. We only give the basic theorems given by Chandy and Misra [CM88] and some corollaries that are used in this thesis. Since the theorems hold in the logic of Chandy and Misra (definition 2.7), in the logic of Sanders (definition 2.10), and in the ImpUNITY logic (definition 4.19), we omit the subscripts of the properties here. For theorems that are not given in [CM88], we give the proofs.

Theorems about Unless

A.1 (Anti) Reflexivity

\[ p \text{ unless } p \]
\[ p \text{ unless } \neg p \]

A.2 Consequence weakening

\[ (p \text{ unless } q \land [q \Rightarrow r]) \Rightarrow p \text{ unless } r \]

A.3 Conjunction

\[ (p \text{ unless } q \land p' \text{ unless } q') \Rightarrow (p \land p') \text{ unless } (p \land q') \lor (a \land q) \lor (q \land q') \]

A.4 Disjunction

\[ (p \text{ unless } q \land p' \text{ unless } q') \Rightarrow (p \lor p') \text{ unless } (\neg p \land q') \lor (\neg p' \land q) \lor (q \land q') \]

A.5 Cancelation

\[ (p \text{ unless } q \land q \text{ unless } r) \Rightarrow (p \lor q) \text{ unless } r \]
A.6 Implication
\[
[p \Rightarrow q] \Rightarrow p \text{ unless } q \\
[\neg p \Rightarrow q] \Rightarrow p \text{ unless } q
\]

A.7 \((p \lor r) \text{ unless } q \Rightarrow p \text{ unless } (q \lor r)\)

A.8 \(p \text{ unless } (q \lor r) = (p \land \neg q) \text{ unless } (q \lor r)\)

A.9 \([\neg q \Rightarrow (p = p')] \Rightarrow (p \text{ unless } q = p' \text{ unless } q)\)

A.10 \((p \text{ unless } (r \lor q) \land r \text{ unless } q) \Rightarrow (p \lor r) \text{ unless } q\)

Proof:
\[
p \text{ unless } (r \lor q) \land r \text{ unless } q \\
\Rightarrow \{A.4, \text{ disjunction with } r \text{ respectively } q\} \\
(p \lor r) \text{ unless } (r \lor q) \land (r \lor q) \text{ unless } q \\
\Rightarrow \{A.5, \text{ cancelation}\} \\
(p \lor r \lor q) \text{ unless } q \\
\Rightarrow \{A.7\} \\
(p \lor r) \text{ unless } q
\]

\[\square\]

A.11 For any set \(W\)
\[
\langle \forall w : w \in W : p_w \text{ unless } q \rangle \Rightarrow \langle \exists w : w \in W : p_w \rangle \text{ unless } q
\]

Theorems about Ensures

A.12 Reflexivity
\(p \text{ ensures } p\)

A.13 Consequence weakening
\((p \text{ ensures } q \land [q \Rightarrow r]) \Rightarrow p \text{ ensures } r\)

A.14 Impossibility
\(p \text{ ensures false} \Rightarrow \neg p\)
A.15 Conjunction

\[(p \text{ unless } q \land p' \text{ ensures } q') \Rightarrow (p \land p') \text{ ensures } (p \land q') \lor (a \land q) \lor (q \land q')\]

A.16 Disjunction

\[p \text{ ensures } q \Rightarrow (p \lor r) \text{ ensures } (q \lor r)\]

A.17 Implication

\[\llbracket p \Rightarrow q \rrbracket \Rightarrow p \text{ ensures } q\]

A.18 \((p \lor r) \text{ ensures } q \Rightarrow p \text{ ensures } (q \lor r)\)

A.19 \(p \text{ ensures } (q \lor r) \Rightarrow (p \land \neg q) \text{ ensures } (q \lor r)\)

A.20 \(p \text{ ensures } (q \lor r) = (p \land \neg q) \text{ ensures } (q \lor r)\)

**Proof:** The implication from left to right follows from A.19 and the reverse implication follows from A.16, disjunction with the predicate \(p \land q\).

A.21 \[\neg q \Rightarrow (p = p') \Rightarrow (p \text{ ensures } q = p' \text{ ensures } q)\]

**Theorems about Leadsto**

A.22 Implication

\[\llbracket p \Rightarrow q \rrbracket \Rightarrow p \leadsto q\]

A.23 Possibility

\[p \leadsto \text{false} \Rightarrow [\neg p]\]

A.24 Disjunction

\[\langle \forall w : w \in W : p_w \leadsto q_w \rangle \Rightarrow \langle \exists w : w \in W : p_w \rangle \leadsto \langle \exists w : w \in W : q_w \rangle\]

A.25 Cancelation

\[(p \leadsto (q \lor b) \land b \leadsto r) \Rightarrow p \leadsto (q \lor r)\]

A.26 PSP (Progress-Safety-Progress)

\[(p \leadsto q \land r \text{ unless } b) \Rightarrow (p \land r) \leadsto (q \land r) \lor b\]
A.27 \[ p \rightarrow q = (p \land \neg q) \rightarrow q \]

A.28 \[ \neg q \Rightarrow (p = p') \Rightarrow (p \rightarrow q = p' \rightarrow q) \]

A.29 \[ (p \rightarrow (r \lor q) \land r \rightarrow q) \Rightarrow (p \lor r) \rightarrow q \]

Proof:

\[
\begin{align*}
    p \rightarrow (r \lor q) \land r \rightarrow q \\
    \Rightarrow & \quad \{ \text{A.24, disjunction with } r \rightarrow r \text{ respectively } q \rightarrow q \} \\
    & \quad (p \lor r) \rightarrow (r \lor q) \land (r \lor q) \rightarrow q \\
    \Rightarrow & \quad \{ \text{definition } \rightarrow \} \\
    & \quad (p \lor r) \rightarrow q
\end{align*}
\]

\[ \square \]
Appendix B

ImpUNITY

In this appendix we give an overview of the ImpUNITY framework.

Statements

The ImpUNITY programming languages is based the notion of statements. For denoting statements we use the language of commands as given by Back in the refinement calculus. A statement is a special kind of command satisfying several restrictions. The syntax of the command language is as follows

\[ A ::= \langle f \rangle | [p] | \{ p \} | \land_{i \in I} A_i | \lor_{i \in I} A_i | A_1; A_2, \]

where \( f \) is a function from states to states, \( p \) is a predicate, and \( I \) is any set. For a predicate \( q \), the semantics of commands in terms of monotonic predicate transformers is defined by:

\[
\begin{align*}
\langle f \rangle(q) & \quad \text{def} \quad q \circ f, \\
[p](q) & \quad \text{def} \quad (p \Rightarrow q), \\
\{ p \}(q) & \quad \text{def} \quad (p \land q), \\
(\land_{i \in I} A_i)(q) & \quad \text{def} \quad (\forall i : i \in I : A_i(q)), \\
(\lor_{i \in I} A_i)(q) & \quad \text{def} \quad (\exists i : i \in I : A_i(q)), \\
(A_1; A_2)(q) & \quad \text{def} \quad A_1(A_2(q)).
\end{align*}
\]

Commands are ordered by the refinement ordering which is defined by

\[ A \leq A' \quad \text{def} \quad (\forall q :: A(q) \leq A'(q)). \]

Then, refinement through a command \( B \) is defined by

\[ A \leq_B A' \quad \text{def} \quad B ; A \leq A'; B. \]
A statement $S$ is a command that is terminating ($S(\text{true}) = \text{true}$), non-miraculous ($S(\text{false}) = \text{false}$), and conjunctive ($S(\forall i : i \in I : p_i) = (\forall i : i \in I : S(p_i))$ for an arbitrary non-empty set of predicates $\{p_i \mid i \in I\}$). In the ImpUNITY framework we use some abbreviations. The multiple assignment $x_1, \ldots, x_n := e_1, \ldots, e_n$ is an abbreviation of the update command $\langle f \rangle$, where $f$ assigns the value of expression $e_i$ to variable $x_i$. The if-statement
\[
\begin{align*}
\text{if } p_1 \rightarrow S_1 \\
\text{[} p_2 \rightarrow S_2 \\
\text{[} \cdots \\
\text{fi}
\end{align*}
\]
is an abbreviation for
\[
(\wedge_{i \in \{1, 2, \ldots\}} ([p_i]; S_i)) \land (\forall i : i \in \{1, 2, \ldots\} : \neg p_i)).
\]
The if-statement chooses nondeterministically one of its branches of which the guard evaluates to true. When all the guards are false, then the statement skips. This is different from a more standard if-statement that would abort when every guard is false. The statement $p \rightarrow S$ is an abbreviation of if $p \rightarrow S$ fi. For a command $A$, and a variable $x$, hiding the variable $x$ in a block construct is denoted by $\langle \text{var } x := e. A \rangle$ and is defined by $\langle f \rangle; A; \langle g \rangle$, where $f$ extends a state with variable $x$ and assigns to $x$ its initial value: $f(\sigma)(x) = e(\sigma)$ and $f(\sigma)(y) = \sigma(y)$ for all other variables $y \in X$. Function $g$ removes variable $x$ from the state.

A procedure mechanism is introduced in the following way. A procedure definition $\text{proc name}(a : \text{Type}) = S$ consists of the name of the procedure, the type of the argument and a statement $S$ which is called the body. A procedure declaration only consists of the name of the procedure and the type of the argument. Furthermore, we extend the language of commands to a language of commands with procedure calls as follows:
\[
B \ ::= \ A \mid \text{name}(e) \mid \wedge_{i \in I} B_i \mid \lor_{i \in I} B_i \mid B_1; B_2,
\]
where $A$ is a command (without procedure calls) and $\text{name}$ is a procedure. For a procedure $\text{name}$ defined by $\text{proc name}(a : \text{Type}) = S$, the semantics of the call $\text{name}(e)$ is the statement $[\text{var } a = e. S]$. In case the procedure definition is known, a call can be resolved by substituting the call by its corresponding statement. A command with procedure calls is a statement if we can prove that it is terminating, non-miraculous, and conjunctive under the assumption that all procedure calls are statements.

The ImpUNITY Programming Language

In this section we introduce the ImpUNITY programming language. An ImpUNITY program consists of the following sections.
• The external-section, containing a modifier specifying in which way an environment is allowed to change the state. A modifier $M$ is statement that satisfies both $M \leq \text{skip}$ and $M; M = M$. This modifier is denoted by $\text{external}(F)$. Typically, we will use modifiers of the form $[=Y]$, where $Y$ is a set of variables. This modifier specifies that it is not allowed to change the values of variables in $Y$.

• The hide-section, containing a set of program variables. This set is called a view and specifies the variables that cannot be read by an environment. The view is denoted by $\text{hide}(F)$.

• The import-section, containing declarations of the procedure that are imported by the program. This set of imported procedures is denoted by $\text{import}(F)$, and the definitions of these procedures must be provided by an environment.

• The export-section, defining procedures that are exported by the program. The set of procedure definitions is denoted by $\text{export}(F)$ and the set of procedure declarations is denoted by $\text{headers}(F)$.

• The initially-section, containing a predicate that specifies the possible initial states of the program. This predicate is denoted by $\text{init}(F)$. There should exist at least one state that satisfies $\text{init}(F)$.

• The assign-section, containing a set of statements that may contain calls to procedures in the import-section and the export-section. The set of statements is denoted by $\text{assign}(F)$ and statements are separated by the symbol $\lbrack \rbrack$.

As an example of a ImpUNITY program, consider the program Buf given in example B.1. This program models a buffer that communicates with its environment by a

---

**Program Buf**

\[
\begin{align*}
\text{external} & \ [=\{b\}] \\
\text{hide} & \ \{b\} \\
\text{import} & \ \text{proc } \text{flushout}(\text{int}) \\
\text{export} & \ \text{proc } \text{flushin}(a : \text{int}) \ = \ (b := b + + [a]) \\
\text{init} & \ b = [] \\
\text{assign} & \ b \neq [] \rightarrow \text{flushout}(\text{hd}(b)); b := \text{tl}(b) \\
\text{end} & \{\text{Buf}\}
\end{align*}
\]

---

**Example B.1.**
procedure interface. Messages are put in the buffer by calling the procedure `flushin` and the buffer outputs messages by calling a procedure `flushout`. The `hide`-section of `Buf` specifies that variable `b` cannot be read by an environment and the modifier `|=\{b\}` in the `external`-section specifies that variable `b` cannot be written by an environment. The statement in the `assign`-section of `Buf` takes care of the output of the messages in `b`. Parallel composition of programs is modelled by the union operator `|`.

**Definition B.1** Let `F` and `G` be ImpUNITY programs. The union `F|G` is defined by

\[
\begin{align*}
\text{hide}(F|G) & \overset{\text{def}}{=} (\text{hide}(F) \cup \text{hide}(G)), \\
\text{external}(F|G) & \overset{\text{def}}{=} (\text{external}(F) + \text{external}(G)), \\
\text{export}(F|G) & \overset{\text{def}}{=} (\text{export}(F) \cup \text{export}(G)), \\
\text{import}(F|G) & \overset{\text{def}}{=} (\text{import}(F) \cup \text{import}(G))\setminus (\text{headers}(F) \cup \text{headers}(G)), \\
\text{init}(F|G) & \overset{\text{def}}{=} (\text{init}(F) \land \text{init}(G)), \\
\text{assign}(F|G) & \overset{\text{def}}{=} (\text{assign}(F) \cup \text{assign}(G)),
\end{align*}
\]

where `M_1 + M_2 \overset{\text{def}}{=} \bigwedge \{M \mid (M_1 \lor M_2) \leq M\}`.

**The ImpUNITY Logic**

The ImpUNITY logic is a UNITY-like logic for ImpUNITY programs. The logic is based on the properties in the Chandy and Misra logic and takes the interference of environments into account. External interference of a program `F` is defined by

\[
\text{inter}(F) \overset{\text{def}}{=} (\text{external}(F) \land \overline{\text{export}(F)})^*,
\]

where `\overline{\text{export}(F)}` is an approximation of possible calls to procedures in `\text{export}(F)`. For a set of procedures `N`, this approximation is defined by `\overline{N} \overset{\text{def}}{=} \bigwedge_{n \in N} \overline{n}`, where for a procedure `proc n(a : T type) = S`,

\[
\overline{n} \overset{\text{def}}{=} \bigwedge_{e \in T type} \llbracket \text{var} \ a = e. \ S \rrbracket.
\]

Properties in the ImpUNITY logic are based on the properties in the Chandy and Misra logic. In contrast to UNITY programs, ImpUNITY programs may contain procedure calls. Calls to exported procedures can be resolved, but the bodies of imported procedures are not known and may differ for different environments. However, we can approximate a call to an imported procedure by `\text{external}(F)`. This gives a way to calculate standard UNITY properties of a program; we substitute `\text{external}(F)` for each call to an imported procedure and treat the resulting program as a standard UNITY program. In this way it makes sense to write `(b = []) \ unless_{CM} false` for the ImpUNITY program `Buf` of example B.1. Now we define the properties in the ImpUNITY logic. To distinguish properties in the ImpUNITY logic from the other properties, these properties are subscripted by `*`. 
Definition B.2 For ImpUNITY program $F$, properties of $F$ are defined by

\[
\begin{align*}
\text{linvariant}_{r} r \in F \quad & \text{def} \quad [r \Rightarrow \text{inter}(F)(r)] \land \text{invariant}_{\text{CM}} r \in F, \\
p \text{unless}_{r} q \in F \quad & \text{def} \quad \langle \exists r : \text{linvariant}_{r} r \in F : (p \land r) \text{unless}_{\text{CM}} q \in F \rangle, \\
p \text{ensures}_{r} q \in F \quad & \text{def} \quad \langle \exists r : \text{linvariant}_{r} r \in F : (p \land r) \text{ensures}_{\text{CM}} q \in F \rangle, \\
p \mapsto_{r} q \in F \quad & \text{def} \quad \langle \exists r : \text{linvariant}_{r} r \in F : (p \land r) \mapsto_{\text{CM}} q \in F \rangle, \\
p \rightarrow_{r} q \in F \quad & \text{def} \quad \langle \exists r : \text{linvariant}_{r} r \in F : (p \land r) \rightarrow_{\text{CM}} q \in F \rangle,
\end{align*}
\]

where $\rightarrow_{\text{CM}}$ is the smallest relation Prop that satisfies the following conditions:

1. If $p$ ensures$_{\text{CM}}$ $q$ in $F$, then $p$ Prop $q$ in $F$.

2. If $p$ Prop $(r \lor q)$ in $F$, $r$ Prop $q$ in $F$, and $[(r \land \neg q) \Rightarrow \text{inter}(F)(\neg p \lor r \lor q)]$, then $(p \lor r)$ Prop $q$ in $F$.

3. If for any set $W$ $\langle \forall w : w \in W : p w \text{ Prop}_{\text{CM}} q \in F \rangle$, and

\[
\langle \forall w : w \in W : [(p w \land \neg q) \Rightarrow \text{inter}(F)(\forall i : i \in W : \neg p i) \lor p w \lor q] \rangle,
\]

then $\langle \exists w : w \in W : p w \rangle$ Prop $q$ in $F$.

The properties defined above are UNITY-like properties in the sense that they can be used in a similar way as the standard UNITY properties. All theorems derived in [CM88] for properties of a single program (see appendix A) also hold for the ImpUNITY properties. Furthermore, the theorems derived for the ensures$_{\text{CM}}$ property also hold for the $\rightarrow_{\text{CM}}$ and $\rightarrow_{r}$ properties. Furthermore, the following substitution theorem holds.

**Theorem B.3** Let $F$ be an ImpUNITY program and let $r$ be a predicate such that $\text{linvariant}_{r} r \in F$. If for predicates $p, p', q$ and $q'$ both $[r \Rightarrow (p = p')]$ and $[r \Rightarrow (q = q')]$ hold, then

\[
\begin{align*}
p \text{unless}_{r} q \in F &= p' \text{unless}_{r} q' \in F, \\
p \text{ensures}_{r} q \in F &= p' \text{ensures}_{r} q' \in F, \\
p \mapsto_{r} q \in F &= p' \mapsto_{r} q' \in F, \\
p \rightarrow_{r} q \in F &= p' \rightarrow_{r} q' \in F.
\end{align*}
\]

Before we give the lemma that expresses the compositionality of the ImpUNITY properties, we formally define what we mean by obeying a modifier, respecting a view and the formal notion of environment.
Let \( M \) be a modifier. We define that a statement or command \( A \) without procedure calls obeys \( M \) by \( A \text{ obey } M \quad \text{def} \quad M \leq A \). A command \( B \) with procedure calls obeys \( M \) if \( B \) is constructed from commands obeying \( M \) and procedure calls, i.e.

\[
\text{name}(e) \text{ obey } M = \text{true}, \\
(\forall i \in I) B_i \text{ obey } M = \langle \forall i : i \in I : B_i \text{ obey } M \rangle, \\
(\exists i \in I) B_i \text{ obey } M = \langle \exists i : i \in I : B_i \text{ obey } M \rangle, \\
(B_1; B_2) \text{ obey } M = (B_1 \text{ obey } M) \land (B_2 \text{ obey } M).
\]

For a procedure definition we define \((\text{proc } n(a : \text{Type}) = S) \text{ obey } M \quad \text{def} \quad \pi \text{ obey } M\). Then, we define that a program \( F \) obeys modifier \( M \) by

\[
F \text{ obey } M \quad \text{def} \quad \langle \forall n : n \in \text{export}(F) : n \text{ obey } M \rangle \\
\land \langle \forall S : S \in \text{assign}(F) : S \text{ obey } M \rangle.
\]

Let \( O \) be a view and let \( R(\sigma, \rho) \quad \text{def} \quad \langle \forall y : y \not\in Y : \sigma(y) = \rho(y) \rangle \) be a relation on states. Then, command \( \{\text{def } \sigma \in O \} \quad \text{def} \quad \{R\} \) is the command that changes the values of variables in \( O \) in an angelic way. For predicate \( p \) and view \( O \), we define \( p \text{ resp } O \quad \text{def} \quad [\{\text{def } \sigma \in O \}(p) \Rightarrow p] \) and for a statement or command \( A \), we define \( A \text{ resp } O \quad \text{def} \quad \langle \forall O' : O' \subseteq O : A \subseteq \{\text{def } \sigma \} A \rangle \).

Like for interference, we define that a command \( B \) with procedure calls respects \( O \) if \( B \) is constructed from commands and procedure calls respecting \( O \), and a procedure call \( \text{name}(e) \text{ respects } O \) if the initialisation of the block of the call respects \( O \):

\[
\text{name}(e) \text{ resp } O = (a := e) \text{ resp } O, \\
(\forall i \in I) B_i \text{ resp } O = \langle \forall i : i \in I : B_i \text{ resp } O \rangle, \\
(\exists i \in I) B_i \text{ resp } O = \langle \exists i : i \in I : B_i \text{ resp } O \rangle, \\
(B_1; B_2) \text{ resp } O = (B_1 \text{ resp } O) \land (B_2 \text{ resp } O).
\]

For a procedure definition we define \((\text{proc } n(a : \text{Type}) = S) \text{ resp } O \quad \text{def} \quad \pi \text{ resp } O\). Then, we define that a program \( F \) respects the view \( O \) by

\[
F \text{ resp } O \quad \text{def} \quad \langle \text{init}(F) \text{ resp } O \rangle \\
\land \langle \forall n : n \in \text{external}(F) : n \text{ resp } O \rangle \\
\land \langle \forall S : S \in \text{assign}(F) : S \text{ resp } O \rangle.
\]

Now, an environment of \( F \) is defined as a program that satisfies both the interference restriction and the observability restriction of \( F \) and a context of \( F \) is an environment which restrictions are satisfied by \( F \).

**Definition B.4** For ImpUNITY programs \( F \) and \( H \), by \( H \text{ env } F \) we denote that \( H \) is an environment of \( F \) which is defined by

\[ H \text{ env } F \quad \text{def} \quad H \text{ obey } \text{external}(F) \land H \text{ resp } \text{hide}(F). \]
By $H \text{ cont } F$ we denote that $H$ is a context of $F$ which is defined as:

$$H \text{ cont } F \equiv F \text{ env } H \land H \text{ env } F.$$ 

The compositionality of the properties is expressed by the following theorem.

**Lemma B.5** Let $F$ and $H$ be ImpUNITY programs. If $H \text{ env } F$, then

- $p \text{ unless}_s q \text{ in } F|H \iff p \text{ unless}_s q \text{ in } F \land p \text{ unless}_{CM} q \text{ in } H,$
- $p \text{ ensures}_s q \text{ in } F|H \iff p \text{ ensures}_s q \text{ in } F \land p \text{ unless}_{CM} q \text{ in } H,$
- $p \rightarrow_\cdot q \text{ in } F|H \iff p \rightarrow_\cdot q \text{ in } F \land p \text{ unless}_{CM} q \text{ in } H.$

If $H \text{ cont } F$, then

- $p \text{ unless}_s q \text{ in } F|H \iff p \text{ unless}_s q \text{ in } F \land p \text{ unless}_s q \text{ in } H,$
- $p \text{ ensures}_s q \text{ in } F|H \iff p \text{ ensures}_s q \text{ in } F \land p \text{ unless}_s q \text{ in } H,$
- $p \rightarrow_\cdot q \text{ in } F|H \iff p \rightarrow_\cdot q \text{ in } F \land p \text{ unless}_s q \text{ in } H.$

**Program Refinement**

ImpUNITY supports the compositional refinement of both specifications and programs as is depicted in figure B.2. First, a system is specified using the properties in the logic of Sanders. There are two ways to refine specifications. A specification can refined using the standard UNITY theorems. Moreover, a specification can be split into specifications of components using the compositional ImpUNITY properties. A specification of a component can again be refined using the standard UNITY theorems. As soon as a specification is specific enough, an ImpUNITY program is developed that satisfies the specification. This can be done independently for each component in the specification. Then, a program is transformed using program refinement. The ImpUNITY framework has two notions of program refinement: observable refinement and a compositional notion of refinement. The observable notion of refinement is the natural notion of program refinement for programs that run in isolation. It expresses that all observable properties of a program are preserved, i.e., an ImpUNITY program $F$ is refined by an ImpUNITY program $G$ if $G$ satisfies every specification that is also satisfied by $F$. This notion of program refinement is not compositional. The second notion of program refinement is compositional and says that an ImpUNITY program $F$ is refined by an ImpUNITY program $G$ if $G|H$ satisfies every specification that is also satisfied by $F|H$ for every environment $H$. For this notion of program refinement we give a number of program transformation rules.

Next, we introduce the two notions of program refinement. Observational program refinement is based on observable properties, properties that do not refer to hidden variables.
Definition B.6 For ImpUNITY program \( F \) and predicates \( p \) and \( q \), observable properties of \( F \) are defined by

\[
\begin{align*}
 p \text{ unless}_O q & \text{ in } F & \overset{\text{def}}{=} & p \text{ unless}_S q \text{ in } F', \\
p \text{ ensures}_O q & \text{ in } F & \overset{\text{def}}{=} & p \text{ ensures}_S q \text{ in } F', \\
p \vDash_O q & \text{ in } F & \overset{\text{def}}{=} & p \vDash_S q \text{ in } F',
\end{align*}
\]

where \( F' \) is a program that is similar to \( F \) except that variables in \( \text{hide}(F) \) are renamed such that both \( p \text{ resp. hide}(F') \) and \( q \text{ resp. hide}(F') \)

Observable refinement is about refinement of full programs. A full program is a program that can run in isolation, i.e. a program with an empty import-section. Observable refinement is defined as preservation of observable properties of a full program.

Definition B.7 For full ImpUNITY programs \( F \) and \( G \), \( G \) is an observable refinement of \( F \), denoted by \( F \sqsubseteq_O G \), if for all predicates \( p \) and \( q \),

\[
\begin{align*}
p \text{ unless}_O q & \text{ in } F \Rightarrow p \text{ unless}_O q \text{ in } G, \\
p \vDash_O q & \text{ in } F \Rightarrow p \vDash_O q \text{ in } G.
\end{align*}
\]

The compositional notion of program refinement is about observable refinement in any environment. Here we use a more general form of program union that allows local variables of the components to be renamed before programs are composed. We say that a set of variables is local to an ImpUNITY program if the variables can neither be read nor written by the environment, and a set of variables is fresh if the variables are not read nor written by the program, and the program does not pose any restrictions on the use of the variables by the environment.

Definition B.8 A set of variables \( Y \) is called local to ImpUNITY program \( F \) if

\[
Y \subseteq \text{hide}(F),
\]

\[
\text{external}(F) \text{ obey } [= Y].
\]
A set of variables $Z$ is called fresh to ImpUNITY program $F$ if

$$Z \cap \text{hide}(F) = \emptyset,$$

$$\text{external}(F) \text{ obey } [=z],$$

$$\text{init}(F) \text{ resp } Z,$$

$$F \text{ obey } [=z].$$

Then, a local renaming is a renaming of a program where local variables are replaced by fresh variables.

**Definition B.9** Let $Y$ and $Z$ be sets of variables and let $F$ be an ImpUNITY program. ImpUNITY program $G \overset{\text{def}}{=} F[Z/Y]$ is called a renaming of $F$ if $Y$ is local to $F$ and $Z$ is fresh to $F$.

**Definition B.10** Let $F$ and $G$ be ImpUNITY programs, let $Y_F$ be the set of all local variables of $F$ and let $Y_G$ be the set of all local variables of $G$. The union $F \mid G$ is defined by $F \mid G \overset{\text{def}}{=} F' \mid G'$, where $F' \overset{\text{def}}{=} F[Z_F/Y_F]$ and $G' \overset{\text{def}}{=} G[Z_G/Y_G]$, and $Z_F$ and $Z_G$ are disjoint sets of variables that are fresh to both $F$ and $G$ and have the same number of elements as $Y_F$ and $Y_G$, respectively.

Note that this definition of program union is only unique up to local renaming.

A program $H$ is an environment of a program $F$ if $H$ respects the interference and observability restrictions of $F$. If we allow local variables to be renamed before programs are composed a weaker notion of environment and context can be used.

**Definition B.11** Let $F$ and $H$ be ImpUNITY programs. By $H \text{ env}_r F$ we denote that $H$ is an environment after a suitable renaming of $F$:

$$H \text{ env}_r F \overset{\text{def}}{=} \langle \exists F' : F' \text{ local renaming of } F : H \text{ env } F' \rangle.$$

By $H \text{ cont}_r F$ we denote that $H$ is a context after renaming of $F$ which is defined

$$H \text{ cont}_r F \overset{\text{def}}{=} F \text{ env}_r H \land H \text{ env}_r F.$$

Since observable program refinement is about full programs, it is only interesting to examine environments for which the union result in a full program. Such an environment is called a full environment.

**Definition B.12** For ImpUNITY programs $F$ and $H$, by $F \text{ env}_f H$ we denote that $H$ is an full environment of $F$ which is defined by

$$H \text{ env}_f F \overset{\text{def}}{=} (H \text{ env}_r F) \land \text{import}(F \mid G, H) = \emptyset.$$

By $H \text{ cont}_f F$ we denote that $H$ is a full context of $F$ which is defined by

$$H \text{ cont } F \overset{\text{def}}{=} F \text{ env } H \land H \text{ env } F.$$
Then, we define refinement of ImpUNITY programs as observable refinement in any full environment.

**Definition B.13** For ImpUNITY programs $F$ and $G$, by $F \sqsubseteq G$ we denote that $G$ is a refinement of $F$ which is defined by

$$F \sqsubseteq G \overset{\text{def}}{=} \langle \forall H : H \text{ env}_F F : H \text{ env}_G G \land F \vDash H \sqsubseteq_0 G \vDash H \rangle.$$  

### Program Transformation Rules

Now, we present some of the program transformation rules for ImpUNITY programs. The first, and probably most powerful, transformation rule is the rule for weak simulation. It allows the modification of local data structures of a program by data refinement, it allows stutterings to be rescheduled and it allows new statements to be added as long as they do not introduce new transitions.

**Transformation B.14 (Weak Simulation)** Let $F$ and $G$ be the following ImpUNITY programs

**Program** $F$

- `external M + [= Y]`
- `hide X \cup Y`
- `import Imp`
- `export`
  - `\{ proc \ n_i(a : \text{Type}_j) = T_j \mid j \in J \}`
- `init initF`
- `assign \langle \{i : i \in I : S_i\} \rangle`
- `end\{F\}`

**Program** $G$

- `external M + [= Z]`
- `hide X \cup Z`
- `import Imp`
- `export`
  - `\{ proc \ n_j(a : \text{Type}_j) = T'_j \mid j \in J \}`
- `init initG`
- `assign \langle \{i : i \in I : S''_i\} \rangle`
- `\langle \{i : i \in I' : S''_i\} \rangle`
- `end\{G\}`

such that $Z \setminus X$ is fresh to $F$ and $Y \setminus X$ is fresh to $G$. Let $P$ be a predicate such that $\langle \{\{x \mid Y\}(P)\} \rangle$ and let $A \overset{\text{def}}{=} \{[x \mid Y]; \{P\}; \{[x \mid Z]\}\}$. Let $T$ be a statement, $T^* \overset{\text{def}}{=} \bigwedge_{i \in \mathbb{N}} T^i$ where $T^0 \overset{\text{def}}{=} \text{skip}$ and $T^{i+1} \overset{\text{def}}{=} T; T^i$; and $T^*_L$ is that statement such that $T^*_L; T^* \leq \text{skip} \land \text{skip} \leq T^*; T^*_L$. If,

- $[P \Rightarrow (M + [= Y] + [= Z])(P)]$,
- $((\Lambda_{i \in I} S_i \land \text{skip}) \lor [\exists_{\text{hide}(F)}]) \leq T$,
- $[A; T^*_L(\text{initF}) \Leftarrow \text{initG}]$,
- $T_j \leq_A T'_j$, for all $j \in J$,
• $S_i \cdot T^* \leq_A S'_i$, for all $i \in I$, and

• $(\bigwedge_{j \in J} S_j \land \text{skip}) \leq S'_i'$, for all $i \in I'$,

then $F \sqsubseteq G$.

The second rule is about the combination of statements. Statements that behave similarly but differ only in their guards can be combined into one statement that is enabled if one of its components is enabled.

**Transformation B.15 (Combine statements)** Let $F$ and $G$ be ImpUNITY programs that only differ in their assign-sections. Let $U = \{p_i \rightarrow S \mid i \in I\}$ be a subset of $\text{assign}(F)$, and let statement $T \overset{\text{def}}{=} (\exists i : i \in I : p_i) \rightarrow S$. If

$$\text{assign}(G) = \text{assign}(F) \setminus U \cup \{T\},$$

then $F \sqsubseteq G$.

The third transformation rule is about the use of local invariants. Each statement of the program may be replaced by a different statement as long as the other statement behaves the same if the local invariant holds. This rule is a kind of substitution rule on the level of programs.

**Transformation B.16 (Local invariant)** Let $F$ and $G$ be ImpUNITY programs that do not call exported procedures, that only differ in their export and assign-sections, and

$$\begin{align*}
\text{export}(F) &= \{\text{proc name}_j(a : \text{Type}_j) = T_j \mid j \in J\}, \\
\text{export}(G) &= \{\text{proc name}_j(a : \text{Type}_j) = T'_j \mid j \in J\}, \\
\text{assign}(F) &= \{S_i \mid i \in I\}, \\
\text{assign}(G) &= \{S'_i \mid i \in I\}.
\end{align*}$$

If for some predicate $r$

• linvariant$_r r$ in $F$,

• $\{r\}; T_j \leq T'_j$, for all $j \in J$, and

• $\{r\}; S_i \leq S'_i$, for all $i \in I$,

then $F \sqsubseteq G$.

The fourth rule is about the strengthening of a guard. Here, we denote by $F \setminus S$ the program $F$ from which statement $S$ is removed from the assign-section.
Transformation B.17 (Strengthening guard) Let $F$ be an ImpUNITY program and let $S = (p \rightarrow T)$ be a statement such that $S \in \text{assign}(F)$. Let $G$ be the ImpUNITY program that only differs from $F$ in the assign-section and

\[ \text{assign}(G) = \text{assign}(F) \setminus \{S\} \cup \{S'\}, \]

where $S' = (p \land q \rightarrow T)$. If

- $[(q \land p) \Rightarrow \text{inter}(F)(q \lor \neg p)]$,
- $q \text{ unless}_{CM} \neg p \text{ in } (F \setminus S)$, and
- $p \rightarrow_{CM} \neg p \lor q \text{ in } (G)$,

then $F \subseteq G$.

The fifth transformation rule concerns the splitting of a statement.

Transformation B.18 (Split statement) Let $F$ be an ImpUNITY program and let $S = (p \rightarrow T)$ be a statement such that $S \in \text{assign}(F)$. Let $\{q_i \mid i \in I\}$ be a set of predicates and let $G$ be the ImpUNITY program that only differs from $F$ in the assign-section and

\[ \text{assign}(G) = \text{assign}(F) \setminus \{S\} \cup \{S_i \mid i \in I\}, \]

where $S_i = (p \land q_i \rightarrow T)$ for $i \in I$. If

- $[p \Rightarrow \langle i : i \in I : q_i \rangle]$,
- $[p \land q_i \Rightarrow \text{inter}(F)(\neg p \lor q_i)]$, for all $i \in I$, and
- $q_i \text{ unless}_{CM} \neg p \text{ in } (G \setminus S_i)$, for all $i \in I$,

then $F \subseteq G$.

The last rule concerns the removal of stutterings from a program.

Transformation B.19 Let $F$ be an ImpUNITY program, and let $S$ and $T$ be statements in $\text{assign}(F)$ such that $\llbracket \text{hide}_{F} \rrbracket \leq T$. Let $G$ be the ImpUNITY program that only differs from $F$ in the assign-section:

\[ \text{assign}(G) = \text{assign}(F) \setminus T \cup \{S; T\}. \]

Then, $F \subseteq G$. 


[ZGK93] Sengzong Zhou, Rob Gerth, and Ruurd Kuiper. Transformations preserving properties and properties preserved by transformations in fair transition
This glossary presents the meaning of symbols that are frequently used in this thesis. The symbols occur as such, or in a primed or subscripted form.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma, T$</td>
<td>state spaces</td>
</tr>
<tr>
<td>$\sigma, \rho$</td>
<td>states</td>
</tr>
<tr>
<td>$A, B$</td>
<td>commands</td>
</tr>
<tr>
<td>$F, G$</td>
<td>programs</td>
</tr>
<tr>
<td>$H$</td>
<td>environment program</td>
</tr>
<tr>
<td>$I, J$</td>
<td>index sets</td>
</tr>
<tr>
<td>$M$</td>
<td>modifier</td>
</tr>
<tr>
<td>$N$</td>
<td>set of procedures</td>
</tr>
<tr>
<td>$O$</td>
<td>view</td>
</tr>
<tr>
<td>$P$</td>
<td>abstraction predicate</td>
</tr>
<tr>
<td>$R$</td>
<td>relation on states</td>
</tr>
<tr>
<td>$S, T$</td>
<td>statements</td>
</tr>
<tr>
<td>$W$</td>
<td>index sets</td>
</tr>
<tr>
<td>$X, Y, Z$</td>
<td>sets of variables</td>
</tr>
<tr>
<td>$e$</td>
<td>state expression</td>
</tr>
<tr>
<td>$f, g$</td>
<td>state transformers</td>
</tr>
<tr>
<td>$n, name$</td>
<td>procedures</td>
</tr>
<tr>
<td>$p, q, r$</td>
<td>state predicates</td>
</tr>
<tr>
<td>$v$</td>
<td>extended sequence</td>
</tr>
<tr>
<td>$x, y, z$</td>
<td>program variables</td>
</tr>
<tr>
<td>$z$</td>
<td>sequence of states</td>
</tr>
</tbody>
</table>
Index

abstraction, 79
  \( Y \sim Z \), 89
predicate, 89
relation, 93
action system formalism, 3
adjoint, 10
angelic composition, 8
assert command, 8
assignment, 13
best compositional
  model, 22, 33, 37
  refinement, 23, 30, 35, 37
cancelation, 135, 137
command, 8, 139
  conjunctive, 9
deterministic, 10
disjunctive, 9
non-miraculous, 9
terminating, 9
  with procedure calls, 110, 140
compositional
  model, 21, 29, 34
  properties, 19, 54, 56
  refinement, 22, 51, 78, 92, 97, 116, 148
conjugated command, 9
conjunction, 135, 137
connected-stutter-free, 25
consequence weakening, 135, 136
context, 49, 76, 96, 113, 145
  after renaming, 92, 116, 147
full, 116, 147
coupling invariant, 94
demonic composition, 8
disjunction, 135, 137
distributed system, 1
dual command, 9
ensures, see properties
environment, 17, 49, 76, 96, 113, 144
  after renaming, 92, 116, 147
  full, 116, 147
equivalence command, 10
fairness, 16, 25, 39
full
  context, 116, 147
  environment, 116, 147
  program, 116, 146
fully abstract, see best compositional
gap, 25
guard command, 8
hole, 25
ImpUNITY, 43, 139
  full program, 116, 146
  logic, 52, 56, 135, 142
  program refinement, 51, 78, 92, 97, 116, 148
    observable, 51, 78, 92, 97, 116, 146
    program section
      assign, 49, 141
export, 112, 141
external, 49, 141
hide, 76, 141
import, 112, 141
initially, 49, 141
internal, 96
proc, 113
read, 96
programming language, 49, 76, 112, 140
properties, 52, 56
transformation rule, see transformation rule
union, 50, 77, 97, 115, 142
after renaming, 91, 116, 147
implication, 136, 137
impossibility, 137
internal until →, see ImpUNITY, properties
invariant, see properties
leadsto →, see properties
linvariant, see properties, ImpUNITY
local invariant, see ImpUNITY, properties
logic
    ImpUNITY, see ImpUNITY, logic
model
    best compositional, 22, 33, 37
    compositional, 21, 29, 34
    operational, 24, 26
    property based, 34
    refinement in a, 30, 35
modifier, 45

non-interference condition, 89

obey a modifier
    command, 46, 144

command with procedure call, 111, 144
procedure call, 111, 144
procedure definition, 111, 144
program, 49, 113, 144
observability, 71
observable properties, 77, 146
observable refinement, see ImpUNITY,
    program refinement, observable
parallel system, 1
predicate, 7
predicate transformer, 8
monotonic, see command
weakest precondition, 8
procedure mechanism, 110, 140
progress-safety-progress, 137
properties
    Chandy and Misra, 18
    ImpUNITY, 52, 56
    Sanders, 20

refinement
    best compositional, 23, 30, 35, 37
    command, 9, 139
    compositional, 22
    ImpUNITY programs, see ImpUNITY,
        program refinement
    in a model, 30, 35
    observable, see ImpUNITY, program
        refinement, observable
    through a command, 9, 139
    through an abstraction, 79, 90
refinement calculus, 7, 139
reflexivity, 135, 136
renaming, 14
    local, 91, 147
resolve a procedure, 110, 113, 140
respect a view
    command, 74, 144
command with procedure calls, 111, 144
predicate, 72, 144
procedure call, 111, 144
procedure definition, 111, 144
program, 76, 113, 144

semantic model, see model
sequence, 24
connected, 26
execution, 24
extended, 25

sequential composition, 8
stable, see properties
statement, 9, 140
guarded, 13
with procedure calls, 111, 140

stutter-free, 24
stuttering, 24, 25
connected, 25
with respect to a view, 71

substitution
axiom, 19

theorem
ImpUNITY, 53, 143
UNITY, 20

transformation rule
abstraction, 90, 119
add statement, 63
combine statements, 63, 149
data refinement, 93, 120
fresh variables, 92
local invariant, 64, 119, 149
remove stuttering, 87, 150
reschedule stutterings, 84
split statement, 67, 150
strengthening guard, 66, 150
weak simulation, 94, 121, 148

truth values, 7

union
ImpUNITY, see ImpUNITY, union
UNITY, see UNITY, union

UNITY, 2, 14
fairness, 16, 25
logic, 135
Chandy and Misra, 18, 135
Sanders, 20, 135

program, 16
program design, 2
properties
Chandy and Misra, 18
Sanders, 20
semantics, see model
union, 17

unless, see properties
until, see properties
update command, 8

variable
abstract, 93
concrete, 93
fresh, 91, 147
local, 91, 146
program, 13

view, 71
Een parallel of gedistribueerd systeem is een systeem waarin meer met elkaar verbonden computers samenwerken. Iedereen komt tegenwoordig, bewust of onbewust, steeds vaker met gedistribueerde systemen in aanraking. Eén van de bekendste is wel de geldautomaat waarmee men (bijna) altijd en (bijna) overal ter wereld geld van een bankrekening kan halen. Computers in gedistribueerde systemen moeten met elkaar kunnen communiceren. Dit kan via het telefoonnet, via speciale fiberglass kabels of zelfs via de satelliet. De infrastructuur voor telecommunicatie en ook de technieken om computers met elkaar te laten communiceren worden beter. Hierdoor wordt het mogelijk grotere en complexere gedistribueerde systemen te bouwen. We zullen steeds afhankelijker worden van deze gedistribueerde systemen en het wordt daarom belangrijker dat deze systemen correct werken.

In het menselijk lichaam, en zeker in het menselijk brein, spelen zich een groot aantal parallelle processen af en we schijnen weinig moeite te hebben om deze te controle- ren en beheersen. Het nadenken over gedistribueerde systemen kost ons daarentegen veel moeite, juist omdat in deze systemen meerdere dingen tegelijk gebeuren die elkaar op subtiele wijze beïnvloeden. Opdrachtgevers kunnen hun wensen vaak niet duidelijk verwoorden en door de complexiteit van de meeste gedistribueerde systemen is het vaak moeilijk te bepalen of het uiteindelijke systeem aan de wensen voldoet. Formele, wiskundige methoden kunnen hier behulpzaam zijn. Ze kunnen gebruikt worden om systemen duidelijk te specificeren en ze geven middelen om te bewijzen dat een systeem aan zijn specifi catie voldoet. Verder kunnen ze een beter inzicht geven in wat een specificatie precies inhoudt en wat er zich mogelijk in een specifiek gedistribueerd systeem afspeelt.

Dit proefschrift presenteert een wiskundige methode die het ontwerpen en bouwen van gedistribueerde systemen ondersteunt. Bijzonder aan deze methode is dat zij een compositioneel ontwerpproces mogelijk maakt; dit wil zeggen dat onderdelen in het systeem onafhankelijk beschreven en gebouwd kunnen worden, waarbij de correctheid van het gehele systeem behouden blijft. Dit kan de ontwikkeling van complexe systemen aanzienlijk vereenvoudigen. We gaan hierbij uit van een bestaand formalisme: UNITY.

Het UNITY formalisme is geïntroduceerd door Chandy en Misra [CM88] en het is een relatief eenvoudig formalisme voor het ontwerpen van parallelle en gedistribueerde
systemen. Het bestaat uit twee delen: een programmeertaal en een programma logica. De
programmeertaal is beperkt en wordt gebruikt om gedistribueerde systeem te modelleren
zonder daarbij allerlei details van een specifieke netwerkarchitectuur in ogenenschouw te
nemen. In de logica kunnen (temporele) eigenschappen worden uitgedrukt en is daarom
geschikt om programma’s te specificeren en om over programma’s te redeneren.

Het ontwerpen van een gedistribueerd systeem in UNITY begint met het geven van
een specificatie: dit is een verzameling van temporele eigenschappen waaraan het te
bouwen systeem moet voldoen. Deze specificatie dient als contract tussen de opdracht-
gever en de ontwerper; zij is zo algemeen mogelijk en abstrahiert van allerlei details.
De ontwerper van het systeem zal vervolgens de specificatie in een aantal stappen ver-
fijnen waarbij problemen van algoritmische aard worden opgelost. Als een specificatie
voldoende uitgekristalliseerd is wordt er een UNITY programma ontwikkeld en bewijst
men dat het programma aan de specificatie voldoet. Dit programma moet worden afge-
beeld op de specifieke netwerkarchitectuur waarvoor men het systeem ontwikkelt. Deze
stap is niet formeel: in het UNITY formalisme kan men de correctheid van deze stap niet
wiskundig bewijzen. Daarom moet de afbeelding zo eenvoudig mogelijk zijn. Vaak heeft
het UNITY programma echter niet de juiste vorm voor een eenvoudige afbeelding. Dan
zou het handig zijn als het programma getransformeerd kon worden naar een geschiktere
vorm, waarbij de correctheid van het programma natuurlijk weer behouden moet blijven.
Dit wordt programmaverfijning genoemd en het wordt door het UNITY formalisme niet
ondersteund. In dit proefschrift bekijken we hoe programmaverfijning aan het UNITY
formalisme kan worden toegevoegd. We geven hiervoor verschillende mogelijkheden aan
de hand van semantische modellen van UNITY programma’s. Deze modellen resulteren
in verschillende noties van programmaverfijning. Twee van deze noties zijn compo-
stationeel; dat wil zeggen dat delen van een programma afzonderlijk verfijnd kunnen worden
zonder dat de correctheid van het hele systeem daarbij in het geding komt. We zullen
laten zien dat de compositionele noties in een bepaald opzicht de beste compositionele
noties van verfijning zijn die we in het UNITY formalisme kunnen bereiken.

In dit proefschrift introduceren we tevens het ImpUNITY formalisme, een uitbrei-
ding van UNITY. Programma’s in UNITY hebben weinig structuur en ook de manier
waarop ze kunnen worden samengesteld is beperkt. In het ImpUNITY formalisme is het
mogelijk meer structuur aan programma’s te geven en ook de manier om programma’s
samen te stellen is uitgebreider. Zo is het mogelijk om de interferentie van een omge-
ving te beperken en een deel van de toestandsruimte voor de omgeving af te schermen.
Tevens biedt ImpUNITY een nieuwe manier van communicatie: componenten kunnen
met elkaar communiceren via een zogenaamd procedure mechanisme. Deze uitbreidingen
vergroten de mogelijkheden voor het compositioneel ontwerpen van programma’s en
kunnen de afbeeldingen van programma’s naar verschillende netwerkarchitecturen ver-
eenvoudigen. De uitbreidingen worden ook doorgevoerd in de ImpUNITY logica en de
notie van programmaverfijning. Net als ImpUNITY programma's ondersteunt de logica een compositionele manier van redeneren en de logica gebruikt worden om componenten afzonderlijk te specificeren. ImpUNITY ondersteunt een compositionele notie van programmaverfijning en een uitgebreide set van programma transformatieregels.

In een tweetal case studies laten we zien hoe het ImpUNITY formalisme gebruikt kan worden bij het oplossen van praktische problemen.
Curriculum Vitae

Rob Theodorus Udink

2 december 1967 Geboren te Terborg

1980 - 1986 Ongedeeld VWO,
Isala College te Silvolde

1986 - 1991 Doctoraal Informatica (met lof),
Universiteit Twente te Enschede

1991 - 1995 Onderzoeker in Opleiding,
Stichting Informatica-Onderzoek Nederland (SION) van de
Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO),
gedetacheerd bij de vakgroep Informatica van de Universiteit Utrecht