A Progress Preserving Refinement

Master Thesis

by

Simon Hudon

July, 2011

Supervisor : Dr. Thai Son Hoang
Professor : Prof. Dr. David Basin

Chair of Information Security
Department of Computer Science
ETH-Zurich
Abstract

Designing correct parallel programs is a difficult challenge, because of the interleaving nature of the execution of processes. Event-B has improved on the situation by applying the techniques of refinement to the design of parallel algorithms in such a way that safety properties can be integrated into the design one gradually. Unfortunately, it does not cover the notion of progress, that is to say that nothing prevents a refinement in Event-B from introducing individual starvation.

In UNITY, temporal properties (both safety and liveness) are part of a comprehensive program logic. However, the distinction that UNITY makes regarding specifications – sets of temporal properties –, and programs – action systems –, rules out the possibility of developing programs in a stepwise fashion.

In this thesis, we provide a model for the UNITY logic and use it to define a refinement order and a set of refinement laws which will enable the symbols of our specifications to do the work in the process of refinement. By taking progress properties as a guideline for the development, the addition of new actions to an existing specification is motivated by a direct mapping to the requirement of progress rather than a vague understanding of what it might achieve. The result of our work is a modelling method called Unit-B, combining refinement and the reasoning about progress properties.
Acknowledgements

My thanks go to all the people who made it possible for me to write this thesis.

First of all, I want to thank Prof. Dr. David Basin who gave me the opportunity to work on this project. Special thanks to my supervisor Dr. Thai Son Hoang for initially believing in me and in my project, for tirelessly trying to shed some light on my ideas and for all the help he offered me.

I also want to thank my friends for their inspiring conversations and their encouragement: Malte Schwerhoff, Roland Häusler, Michael Haspra, Markus Pilman, Julian Tschannen, Bernhard Brodowski and the others.

Special thanks to my brother Francois for our regular conversations, for his support and for his patience. Finally, I would like to thank my parents, Louis and Danielle, for their unwavering support and their constant encouragement.
Contents

Abstract

Acknowledgements

0 Introduction
  0.1 Contribution ................................................. 3
  0.2 Structure of the Thesis ................................. 6

1 Background
  1.0 Computation Calculus - A Short Introduction ........... 8
    1.0.0 Some Useful Theorems ................................ 16
  1.1 Program Notation ........................................... 25
    1.1.0 Hoare Triples ........................................ 25
    1.1.1 Program Relations .................................... 26
  1.2 Transition Systems ......................................... 27
    1.2.0 Event-B ................................................. 30
    1.2.1 UNITY ................................................ 31
# Temporal Properties

## 2.0 Safety Properties

### 2.0.0 The co operator

### 2.0.1 The unless operator

## 2.1 Progress Properties

### 2.1.0 The leads-to operator

### 2.1.1 The ensures operator

### 2.1.2 The transient operator

# The Unit-B Method

## 3.0 Existing Scheduling Policies in UNITY

### 3.0.0 Weak Fairness

### 3.0.1 Strong Fairness

## 3.1 General Scheduling

## 3.2 The Unit-B Notation

## 3.3 Verification Conditions

### 3.3.0 Invariance

### 3.3.1 co Properties

### 3.3.2 Transient Properties
Chapter 0

Introduction

The problem of constructing reliable programs is a challenging one. As programs become large, so does the difficulty of the challenge. In concurrent programming, the problem is made much harder by the combinatorial explosion of possible executions. Because of the flexibility in interleaving of programs, it is extremely difficult to construct all the possible executions of the given programs and verify their properties. One solution to this problem is to avoid the need of exploring all the possible executions. Indeed, with formal methods, we have an alternative way of reasoning about programs other than the exhaustive exploration of the possible executions.

We adhere to the school of constructive formal methods which is constituted of two elements. The first one can be traced back EW Dijkstra’s paper [5] suggests that we integrate the concern of correctness to the activity of construction. An alternative is to have a construction phase and a —possibly formal— verification phase. Decoupling the two phases makes both harder to do. The integration of the two has the benefit that, not only do we reduce significantly the probability of getting away with a bad solution — thus making verification easier— but we also increase the probability of finding a correct solution —thus making construction easier—. In short,
Dijkstra suggested that the concern for correctness dictate the directions to explore when constructing a program.

The second aspect of constructive formal methods, the formal part, which can be traced back to Hoare, Naur and Floyd \[24, 12, 16\], can be split in the two branches of constructive formal methods and formal verification. They share the observation that treating a program as a formula and letting its constituting symbols dictate what we can do with the formula —other than interpreting it— relieves us from considering a huge space of discrete possibilities of which we would be most assured to forget some members.

0.1 Contribution

This work is strongly inspired by Event-B from Jean-Raymond Abrial \[1\] and UNITY from Mani K. Chandy and Jayadev Misra \[4\]. Both are concerned with discrete transition systems such as concurrent programs and control systems.

They choose the action system as the basic structure of their programs. For concurrent programs and control systems, it has the advantage that it allows us to describe what has to be done before we take care of distributing the burden among concurrent components of the targeted discrete system. In the case of both methods, the idea that the work is to be carried out by cooperating sequential processes is all but forgotten in the design stage.

Although our initial aim is an extension of Event-B to deal with progress properties, the results are a theory and a method that are half-way between Event-B and UNITY. Whereas the method is more strongly inspired by Event-B, the theory is much closer to UNITY. Instead of thinking of them as a different Event-B, we decided to start thinking of them as different formal method inspired by both. We called it Unit-B, with the following intuitions.
**Event-B**  On the one hand, Event-B is designed for developing safety properties incrementally using refinement. A given specification can be built by adding a safety property, in the form of an invariant. This provides a simple way of refining a specification without having to deal explicitly with every single invariant that it preserves. Unfortunately, Event-B’s treatment is rather poor when it comes to progress: it is limited to the usual treatment for termination in structured programming.

**UNITY**  On the other hand, UNITY is very strong when it comes to progress. It is equipped with an elegant temporal logic which deals both with safety and progress properties. However, UNITY has a strong distinction between a specification and a program. It defines a specification as a set of temporal properties and it is refined by adding properties or by replacing some by stronger ones. Also, in UNITY, a program is an action system and the possibility for refinement is rather limited at this point. A usual UNITY program development starts with the statement of a specification to proceed with its refinement and, when it has reached a suitable state, it is transferred, almost all at once, into a program. This is in sharp contrast with Event-B in which an action system is the first stage of a specification and refinement rules are applied to it.

**Goal**  We would like to bridge the gap between those two methods. Like in Event-B, we would like to take advantage of considering programs to be a special cases of specifications. This allows us to build a chain of very small changes from the first specification to the last. Doing so, we avoid ever stepping over a frontier between specifications and programs, like in UNITY. On the other hand, we would like the refinement rules to preserve progress properties which is not currently the case in Event-B. From UNITY, we would like to use the beautiful relation between temporal logic and programs and the effective separation of concerns —the UNITY notion of fairness— which allows it.
We would like to make clear what we are not looking for a way to verify that a given program satisfies its specification and neither are we looking for a refinement framework to support the verification of the refinement relation between two specifications. The latter is more constructive, in a technical sense, than the former but we deem it insufficient. It is our goal to find a way to calculate a suitable refinement for a given specification. The difference between the last two is small but, we think, important. When verifying a refinement, we have to guess a concrete specification and then try to verify it indeed refines its abstract specification. If we calculate a refinement, we try to build an elegant proof of refinement and fit a concrete specification to it. It is a our preferred approach because of how little guesswork is involved.

**Event-B and progress** The current way of dealing with progress properties in Event-B is to leave them for the end so that no refinement risk breaking their validity after they have been proven. The problem with the approach is that the introduction of safety properties is usually done in many refinement steps and, for simplicity, it is not always desirable to take the weakest specification meeting the properties. Therefore, it is not uncommon that, at the point where progress is supposed to enter a design, it has been completely precluded by the chain of refinements.

**The methodological promise of the concern for progress** It is the author’s experience and the support from various notes by Dijkstra that, when designing a loop, finding the variant early turns out to be of strong heuristic value when the time comes to design the body of the loop. Since progress in general transition systems is strongly reminiscent of termination in sequential programming, the analogy between sequential programming and concurrent programming seems like a fruitful one and it set us to use the notion of progress as heuristic for designing concurrent programs.
About fairness  Although initially hesitant to adopt fairness assumptions, we have eventually realized the improvement they make for the separation of concerns in the implementation of a progress property. Without fairness, there are two distinct ingredients that, by necessity, get entangled together in the proofs of progress: the individual contribution of the actions to progress and their proper coordination. Indeed, an action can be proved to help make progress but, if it is not executed often enough, it all goes to waste. The separation between the two is very convenient because the notion of progress is central to the functionality of a program whereas the coordination is very general and therefore very remote from the central functionality. The consequences of this are that addressing the requirement of progress will be very revealing to the structure of a suitable solution whereas coordination will not and addressing it will produce more intellectual pollution than insight. Coordination is therefore best left at the latest stages of a development. Before it is addressed, we use fairness as an abstract statement of what it can achieve.

It can also be seen that the strict structuring of variants of Event-B can be the final target after removal of the fairness assumption, that is to say the implementation of the coordination, but the exact structuring need not impede us until everything else have been dealt with.

0.2 Structure of the Thesis

In Chapter 1, we give some background information for our work. This includes a brief review of the computation calculus from R. Dijkstra[10], and how the computation calculus can be used to give the semantic basis for general transition systems. Chapter 2 shows how temporal properties (safety and progress) are specified and reasoned about using the computation calculus. Our main contribution, the Unit-B method is described in Chapter 3. Next, we show how refinement can be performed within our framework
(Chapter 4). The method is illustrated by an example: the development of a mutual exclusion specification and its implementation in Chapter 5. We conclude and give future research directions in Chapter 6.
Chapter 1

Background

1.0 Computation Calculus - A Short Introduction

This section gives a brief introduction to computation calculus, based on [10].

We have:

\[ S : \text{the state space which is a non-empty set of “state”}. \]

\[ C : \text{the computation space, a set of non-empty (finite or infinite) sequences of states, which are called “computations”}. \]

Furthermore, the set of statements \( \text{CPred} \) is defined as follows.

**Definition 1** (Computation Predicate). \( \text{CPred} = C \rightarrow B \), i.e. functions from computations to Booleans.
The standard Boolean operators of the predicate calculus are lifted accordingly, i.e. extended to apply to CPred. For example, assuming \( s, t \in \text{CPred} \) and \( \tau \in \mathcal{C} \), we have:

\[
(s \Rightarrow t).\tau \equiv (s.\tau \Rightarrow t.\tau) \quad (1.1)
\]

\[
\langle \forall i:: s.i \rangle .\tau \equiv \langle \forall i:: s.i.\tau \rangle \quad (1.2)
\]

The everywhere-operator quantifies universally over all computations, i.e.

\[
[ s ] \equiv \langle \forall \tau:: s.\tau \rangle \quad (1.3)
\]

**Postulate 2.** CPred is a predicate algebra.

A consequence of Postulate 2 is that CPred satisfies all postulates for the predicate calculus as defined in [8]. In particular, \texttt{false} and \texttt{true} are the “top” and the “bottom” elements of the complete boolean lattice with the order \([ \_ \Rightarrow \_ ]\) specifying by these postulates. The lattice operations are denoted by various boolean operators including \(\land, \lor, \neg, \Rightarrow, \text{etc.}\)

**Sequential Composition**  Sequential composition operator, i.e. 
\(;\), is defined as follows.

**Definition 3** (Sequential Composition).

\[
(s; t).\tau \equiv (\#\tau = \infty \land s.\tau) \lor \\
(\exists n:: n < \#\tau: s.(\tau \uparrow n + 1) \land t.(\tau \downarrow n)) \quad (1.4)
\]

where \(\#\), \(\uparrow\) and \(\downarrow\) denote sequence operations ‘length’, ‘take’ and ‘drop’, respectively.

Sequential composition operator satisfies the following postulate.

**Postulate 4.** 
\(;\) is universally disjunctive in its first and positively disjunctive in its second argument.
In the course of reasoning using computation calculus, we make use of the distinction between infinite (“eternal”) and finite computations. Two constants $E, F \in \text{CPr}ed$ has been defined for this purpose.

**Definition 5** (Eternal and Finite Computations).

\begin{align*}
E &= \text{true}; \text{false} \quad (1.5) \\
F &= \neg E \quad (1.6) \\
\text{s is eternal } &\equiv [s \Rightarrow E] \quad (1.7) \\
\text{s is finite } &\equiv [s \Rightarrow F] \quad (1.8)
\end{align*}

for all predicate s.

We borrow some important properties related to $E$ and $F$ from [10].

\begin{align*}
s;\text{false} &= s \land E \quad (1.9) \\
[s \Rightarrow E] \Rightarrow s; t &= s \quad (1.10) \\
[s; t &= (s \land E) \lor (s \lor F); t] \quad (1.11)
\end{align*}

A constant $\mathbb{1}$ is defined as (left- and right-) neutral element for sequential composition.

**Definition 6** (Neutral Element for Sequential Composition).

\begin{align*}
\mathbb{1}.\tau &\equiv \#\tau = \mathbb{1} \quad (1.12)
\end{align*}

**State Predicates** In fact, $\mathbb{1}$ is the characteristic predicate of the state space. Moreover, without distinguishing between a single state and the singleton computation consists of that state, we can identify predicates on the state by the predicates that hold only for singleton computation. Denote the set of state predicates by $\text{SPr}ed$, we have the following definition.
**Definition 7** (State Predicate).

\[ p \in \text{SPred} \equiv [ p \Rightarrow \top ] \quad (1.13) \]

for all predicates \( p \).

**Theorem** (State Restriction). For all \( s \) and all state predicates \( p \), we have

\[
\begin{align*}
p; s &= p; \text{true} \land s \\
q; s &= s \land \text{true}; p \end{align*}
\]  
(State restriction)

For a state predicate \( p \), the set of computations with the initial state satisfying \( p \) is captured by \( p; \text{true} \). A special notation \( \bullet : \text{SPred} \to \text{CPred} \) is introduced to denote this restriction.

**Definition 8** (Initially Operator). \( \bullet p = p; \text{true} \)

Later, for simplicity, we often make transformations by applying Definition 8 implicitly.

Another rule related to state predicate which we also anonymously apply is the following, where \( p, q \) are state predicates.

\[
p; q = p \land q \quad (1.14)
\]

**Iterations** Iterator are given by two unary operators \( ^* \) and \( ^\infty \), representing the “finite iterator” and “infinite iterator”, respectively. Their definitions are given in the standard way using fix-point as follows.

**Definition 9.** For all \( s \), we have

\[
\begin{align*}
^* s &= \langle \mu x :: s; x \lor 1 \rangle \\
^\infty s &= \langle \nu x :: s; x \rangle
\end{align*}
\]

(1.15)  
(1.16)
An important and useful theorem with iteration is the following *Tail-recursion theorem*.

**Theorem** (Tail-recursion Theorem).

\[
\langle \mu x:: s; x \lor t \rangle = *s; t \\
\langle \nu x:: s; x \lor t \rangle = *s; t \lor \infty s \quad \text{(Tail-recursion theorem)}
\]

Some other important properties are borrowed from [10].

\[
[ s \Rightarrow \neg 1 ] \Rightarrow [ \infty s \Rightarrow E ]
\]  \hspace{1cm} (1.17)

**Temporal Logic** The most important temporal operator is the unary “next” operator, denoted as \( X \). Informally, a computation satisfies \( Xs \) if the first atomic step proceeds a computation satisfying \( s \). Specifically, an atomic computation is a computation of length 2. In computation calculus, the notion is captured by a constant \( X \in C\text{Pred} \).

**Definition 10** (Atomic Actions).

\[
X.\tau \equiv \#\tau = 2 \quad (1.18)
\]

\[
\alpha \text{ is an atomic computation } \equiv [ \alpha \Rightarrow X ] \quad (1.19)
\]

for all trace \( \tau \) and computation \( \alpha \).

Given the definition, the “next” operation in LTL can just be expressed as \( Xs = X; s \). Important properties of \( X \) related to non-singleton computations, finite and eternal computations are as follows.

\[
[ 1 \not\equiv X; \text{true} ] \quad (1.20)
\]

\[
\infty X = E \quad (1.21)
\]

\[
* X = F \quad (1.22)
\]
The “some-time” operator is defined using the prefix $\mathbf{F}$, in particular, this satisfying the axiom of LTL, i.e.

$$\mathbf{F}s = \langle \mu x :: X; x \lor s \rangle,$$  \hspace{1cm} (1.23)

for all $s$.

The “always” $\mathbf{G} : \mathbb{C} \mathbb{P} \mathbb{r} \mathbb{e} \mathbb{d} \to \mathbb{C} \mathbb{P} \mathbb{r} \mathbb{e} \mathbb{d}$ operator is defined as the dual of “some-time”.

**Definition 11** (Always Operator). $\mathbf{G}s = \neg(\mathbf{F};\neg s)$, for all $s$

This “always” operator $\mathbf{G}$ satisfies similar fix-point characterisation from LTL, i.e.

$$\mathbf{G}s = \langle \nu x :: (X; x \lor 1) \land s \rangle,$$  \hspace{1cm} (1.24)

for all $s$.

An important property of $\mathbf{G}$ is that it is strengthening, i.e. for all $s$

$$[ \mathbf{G}s \Rightarrow s ].$$  \hspace{1cm} (1.25)

Another useful technique that is frequently applied is the combination of monotonicity of $\mathbf{G}$ and the possibility of $\mathbf{G}$ to distribute over conjunction. This allows us to strip all the outer $\mathbf{G}$ in some proofs.

$$[ (\mathbf{G}s \Rightarrow (\mathbf{G}t \Rightarrow \mathbf{G}u)) \iff (s \land t \Rightarrow u) ]$$

$$\mathbf{G}s \Rightarrow (\mathbf{G}t \Rightarrow \mathbf{G}u)
= \{ \text{shunting} \}$$
\[ Gs \land Gt \Rightarrow Gu \]
\[ = \{ G \text{ over } \land \} \]
\[ G(s \land t) \Rightarrow Gu \]
\[ \Leftarrow \{ \text{ monotonicity } \} \]
\[ s \land t \Rightarrow u \]

**Definition 12** (Persistence). For all \( s \), \( s \) is persistent \( \equiv s = Gs \).

We borrow the *persistence rule* from [10]: for all \( s, t \) and persistent \( u \), we have
\[ [s; t \land u \Rightarrow s;(t \land u)] . \quad (1.27) \]

Still about about the notion of persistence, we also use the following theorem from [10]:
\[ s \text{ is persistent } \Leftarrow [s \Rightarrow X;S] . \quad (1.28) \]

**Substitution Theorem**

It is an axiom from Misra and Chandy [4] and we can prove it using our model. It is the most straightforward way to use the invariant of a program. It allows us to introduce it or remove it from any temporal property. Instead of proving it for all UNITY operators, we will prove it for two computation calculus operators, \( G \) and \( ; \).

For any computation \( s \) and state predicate \( p \), and assuming
\[ G \bullet p \]
we observe:
\[ \text{Gs} \]
\[ = \{ \text{G} \cdot p \} \]
\[ \text{Gs} \land \text{G} \cdot p \]
\[ = \{ s \land \_ \text{ over G for persistent s} \} \]
\[ \text{G} (s \land \text{G} \cdot p) \]

At this point, we can continue propagating \( \text{G} \cdot p \) in \( s \) or transform it into \( \cdot p \). The following allows us to get rid of \( \text{G} \).

\[ \text{G} (s \land \text{G} \cdot p) \]
\[ = \{ \text{G is strengthening} \} \]
\[ \text{G} (s \land \cdot p \land \text{G} \cdot p) \]
\[ = \{ \text{state restriction; s \land \_ over G} \} \]
\[ \text{Gp}; s \land \text{G} \cdot p \]
\[ = \{ \text{G} \cdot p \} \]
\[ \text{Gp}; s \]

In the same conditions, we can propagate an invariant in the middle of a sequential composition, with \( s \) and \( t \) two computations.

\[ s; t \]
\[
= \{ \text{persistence rule (C9) with } G \bullet p \} \\
= s; (t \land G \bullet p)
\]

And the same goes for getting rid of \( G \) or propagating it further.

\[
= s; (t \land G \bullet p) \\
= \{ G \text{ is strengthening} \} \\
= s; (t \land p \land G \bullet p) \\
= \{ \text{persistence rule (C9)} \} \\
= s; (t \land p) \\
= \{ \text{state restriction} \} \\
= s;p;t
\]

### 1.0.0 Some Useful Theorems

We first define a constant \( S \) that will be used in our formulation later.

**Definition 13 (Constant \( S \)).**

\[
[S \equiv 1 \lor X] \\
(1.29)
\]
**Theorem.** Given $p$ is a state predicate, we have

\[
\begin{align*}
[p \Rightarrow G \bullet p] & \quad \text{(C0)} \\
[G \bullet p \equiv p;(G \bullet p)] & \quad \text{(C1)} \\
[*;s;\infty s \equiv \infty s] & \quad \text{(C2)} \\
[(G \bullet p);s \equiv \infty(p;X) \lor *(p;X);p;s] & \quad \text{(C3)} \\
[(G \bullet p);s \equiv \langle \forall x:: p;X;x \lor p;\rangle] & \quad \text{(C4)} \\
[G \bullet p \equiv (G \bullet p);S;(G \bullet p)] & \quad \text{(C5)} \\
[u \Rightarrow G s] = [u \Rightarrow s], \text{ where } u \text{ is persistent} & \quad \text{(C8)} \\
[u \Rightarrow (s;t \equiv s;(t \land u))], \text{ where } u \text{ is persistent} & \quad \text{(C9)} \\
[A \Rightarrow X] \Rightarrow \infty(F;A) \text{ is persistent} & \quad \text{(C10)} \\
[S;false \equiv false] & \quad \text{(C11)} \\
[\forall i:: F;(G s.i)] \equiv F;\langle \forall i:: G s.i \rangle] & \quad \text{(C12)} \\
[F;(G F;A;true) \equiv G F;A;true] & \quad \text{(C13)}
\end{align*}
\]

provided $[A \Rightarrow X]$

**Proof.** We proof the theorems separately.

- Proof of (C0)

\[
\begin{align*}
[p \Rightarrow G \bullet p] \\
\subseteq \{ \text{fix-point induction} \} \\
[p \Rightarrow (X;p \lor 1) \land \bullet p] \\
\subseteq \{ \text{strengthening} \} \\
[p \Rightarrow 1 \land \bullet p] \\
= \{ \text{State restriction} \} \\
\text{true}
\end{align*}
\]
• Proof of (C1)

\[
G \ p; \ true \\
= \quad \{ \ \text{G is strengthening (1.25)} \} \\
G \ p; \ true \land p; \ true \\
= \quad \{ \ \text{State restriction} \} \\
p; (G \ p; \ true)
\]

• We prove (C2) using a ping-pong argument.

Ping:

\[
[ \ast; \ast; \infty; s \Rightarrow \infty; s ] \\
= \quad \{ \ \text{Iteration (1.16)} \} \\
[ \ast; \ast; \infty; s \Rightarrow \langle \nu \ x :: s; x \rangle ] \\
\Leftarrow \quad \{ \ \text{fix-point induction} \} \\
[ \ast; \ast; \infty; s \Rightarrow s; \ast; \ast; \infty; s ]
\]

From here, we simply transform the right-hand side of the above formula into its left-hand side.

\[
\ast; \ast; \ast; s; \ast; \infty; s \\
= \quad \{ \ \text{iteration rolling} \} \\
\ast; s; \ast; \ast; \ast; \infty; s \\
= \quad \{ \ \text{fix-point (un-)folding} \} \\
\ast; \ast; \ast; s; \ast; \infty; s
\]

Pong:

\[
\ast; \ast; s; \ast; \infty; s
\]

18
\[ \{ [ \top \Rightarrow \ast ] \} \]
\[ \top; \infty \ast \]
\[ = \{ \text{Identity} \} \]
\[ \infty \ast \]

* Proof of (C3)

\[
(G \bullet p); s
= \{ \text{Fix-point formulation (1.24)} \}
\langle \forall x :: (X; x \lor \top) \land \bullet p) ; s \rangle
= \{ \text{State restriction} \}
\langle \forall x :: p; X; x \lor p) ; s \rangle
= \{ \text{Tail-recursion theorem} \}
\ast (p; X); p; s \lor \infty (p; X); s
= \{ (1.10) using \{ p; X \Rightarrow \neg \top \} and (1.17) \}
\ast (p; X); p; s \lor \infty (p; X)
\]

* Proof of (C4)

\[
(G \bullet p); s
= \{ (C3) \}
\infty (p; X) \lor \ast (p; X); p; s
= \{ \text{Tail-recursion theorem} \}
\langle \forall x :: p; X; x \lor p; s \rangle
\]
• Proof of (C5)

\[(G \cdot p); S; (G \cdot p)\]

\[\{\text{(C3) twice} \}\]

\[\infty(p; X) \lor *\left((p; X); p; S; \ast(p; X); p\right) \lor *\left((p; X); p; S; \infty(p; X)\right)\]

\[\{ \text{(0) and (1) below} \}\]

\[\infty(p; X) \lor *(p; X); p\]

\[\{\text{(C3)}\}\]

\[G \cdot p\]

\[
\begin{align*}
\ast(p; X); p; S; \ast(p; X); p & \equiv \ast(p; X); p \\
& \text{(0)}
\end{align*}
\]

\[\ast(p; X); p; S; \ast(p; X); p\]

\[\{ S = X \lor 1 \}\]

\[\ast(p; X); p; \ast(p; X); p \lor \ast(p; X); p; X; \ast(p; X); p\]

\[\{ \text{iteration rolling} \}\]

\[\ast(p; X); \ast(p; X); p \lor \ast(p; X); p; X; \ast(p; X); p\]

\[\{ p; p = p \text{ and iteration rolling} \}\]

\[\ast(p; X); \ast(p; X); p \lor \ast(p; X); p; X; \ast(p; X); p\]

\[\{ \text{iteration rolling with s := p; X} \}\]

\[\ast(p; X); \ast(p; X); p \lor \ast(p; X); \ast(p; X); p; X; p\]

\[\{ \text{Kleene algebra: } \ast x; \ast x = \ast x, \text{ twice} \}\]

\[\ast(p; X); p \lor \ast(p; X); p; X; p\]

\[\{ \text{Kleene algebra: } \ast x = \ast x \lor \ast x; x \}\]

\[\ast(p; X); p\]
\[(\ast(p;X);p;S;\infty(p;X) \equiv \infty(p;X))\] 

\[
\ast(p;X);p;S;\infty(p;X)
= \{ S = X \lor 1 \}
\]

\[
\ast(p;X);p;\infty(p;X) \lor \ast(p;X);p;X;\infty(p;X)
= \{ \text{iteration rolling} \}
\]

\[
\ast(p;X);p;\infty(X;p) \lor \ast(p;X);p;X;\infty(p;X)
= \{ \text{fix-point (un-)folding} \}
\]

\[
\ast(p;X);\infty(p;X)
= \{ \text{(C2)} \}
\]

\[
\infty(p;X)
\]

- We prove (C8) using a ping-pong argument.

Ping:

\[
[u \Rightarrow G s]
= \{ \text{Fix-point formulation (1.24)} \}
\]

\[
[u \Rightarrow \langle \forall x :: (X;x \lor 1) \land s \rangle]
\leftarrow \{ \text{fix-point induction} \}
\]

\[
[u \Rightarrow (X;u \lor 1) \land s]
= \{ \text{Persistence rule (C9)} \}
\]

\[
[u \Rightarrow (X;\text{true} \lor 1) \land s]
= \{ (1.20) \}
\]

\[
[u \Rightarrow s]
\]
Pong:

\[ [u \Rightarrow G s] \]

\[ \Rightarrow \{ G \text{ is strengthening (1.25)} \} \]

\[ [u \Rightarrow s] \]

• Proof of (C9)

\[ [u \Rightarrow (s; t \equiv s; (t \land u))] \]

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

\[ [u \Rightarrow (s; t \Rightarrow s; (t \land u))] \land [u \Rightarrow (s; (t \land u) \Rightarrow s; t)] \]

\[ = \{ \text{Shunting; monotonicity of ;} \} \]

\[ [u \land s; t \Rightarrow s; (t \land u)] \land [u \Rightarrow \text{true}] \]

\[ = \{ \text{Persistence rule (1.27); predicate calculus} \} \]

true

• Proof of (C10)

\[ \infty(F; A) \]

\[ = \{ \text{fix-point (un-)folding} \} \]

\[ F; A; \infty(F; A) \]

\[ = \{ (1.22) \} \]

\[ *X; A; \infty(F; A) \]

\[ = \{ \text{fix-point (un-)folding} \} \]

\[ X; F; A; \infty(F; A) \lor A; \infty(F; A) \]

\[ = \{ \text{fix-point (un-)folding} \} \]

\[ X; \infty(F; A) \lor A; \infty(F; A) \]
\[ A \Rightarrow X \]
\[ X; \infty (F; A) \]

- Proof of (C11)

\[ S; false \]
\[ = \{ (1.9) \} \]
\[ S \land E \]
\[ \Rightarrow \{ [S \Rightarrow F] \} \]
\[ F \land E \]
\[ = \{ (1.6) \} \]
\[ false \]

- We prove (C12) using a ping-pong argument.

ping \Rightarrow

We calculate with the negation of the implication and transform it into false

\[ \forall i :: F;(G.s.i) \land \neg(F;(\forall i :: (G.s.i))) \]
\[ = \{ \text{duals} \} \]
\[ \forall i :: F;(G.s.i) \land G(\exists i :: \neg(G.s.i)) \]
\[ \Rightarrow \{ \text{predicate calculus} \} \]
\[ \forall i :: F;(G.s.i) \land G(\exists j :: \neg(G.s.j)) \}
\[ \Rightarrow \{ \text{substitution} \} \]
\[ \forall i :: F;(G.s.i \land (\exists j :: \neg(G.s.j))) \}
\[ = \{ \land \text{over } \exists \} \]
\( \forall i :: F; (\exists j :: G s.i \land \neg(G s.j)) \)
\[ = \{ \text{predicate calculus} \} \]
\( \forall i :: F; \text{false} \)
\[ = \{ (1.9) \text{ and } (1.6) \} \]
false

\( \text{pong} \leftarrow \)

\( F; (\forall i :: G s.i) \)
\[ \implies \{ \text{quantified constant} \} \]
\( \forall j :: F; (\forall i :: G s.i) \)
\[ \implies \{ \text{instantiation with } i := j \} \]
\( \forall j :: F; (G s.j) \)

- Proof of (C13)

\( F; (G F; A; \text{true}) \)
\[ = \{ \infty(F; A) \text{ is persistent} \} \]
\( F; \infty(F; A) \)
\[ = \{ \text{rolling} \} \]
\( F; F; \infty(A; F) \)
\[ = \{ F = F; F \} \]
\( F; \infty(A; F) \)
\[ = \{ \text{rolling} \} \]
\( \infty(F; A) \)

24
1.1 Program Notation

In the next section, we introduce notations to make it practical to apply results of computation calculus to atomic actions described built of a finer granularity than the usual computation calculus. The finer granularity is required to deal with program variables and build the state predicates to which our theorems of computation calculus will apply.

1.1.0 Hoare Triples

In computation calculus, when impose a constraint on a step \( a \), we express it as

\[
[p; a \Rightarrow X; q]
\]

instead of the classical hoare triple

\[
\{ p \} a \{ q \}
\]

It is very convenient for semantic work but when dealing with specific programs, this notation will be too cumbersome. In a similar way to [15], we
will express \( a \) as a relation between two states by naming the variables of the states rather than the states themselves and using unprimed symbols for the initial values of program variables and primed symbols for their final values.

By conventions, state predicates like \( p \) and \( q \) only contain symbols that pertain to one state. By default, they pertain to the first state of a computation but we can choose to apply the predicates to a second state by appending a prime to its name. Accordingly, \( p \) contains only unprimed symbols and \( q' \) only primed symbols. This convention encodes the intricacies of \( X;_\_ \) which can be removed from the formulae. The above Hoare triple will therefore be formulated as

\[
[p \land a \Rightarrow q']
\]

whenever we need a more fine grained formulation of a program than what is required in the semantics parts.

1.1.1 Program Relations

For the purpose of being as general as possible, we will build actions by starting to true (if it is not a refinement, see the refinement chapter for more details) and adding as conjuncts constraints relating the initial values and the final values of variables. As much as possible, we try to make each constraint focus on one final value. It often leads to having constraints of the form:

\[
x' = x + y
\]

Two points are worth noticing. First, it has almost the shape of an assignment. The shape makes it easy to substitute \( x' \) in postconditions and, at
the end of a development, it is sensible to expect that an assignment can be calculated from this shape.

Second, \( y' \) is absent from the above formulae. It is important to realize that, as a consequence of this absence, any value of \( y' \) will satisfy the above constraint. When something is learned about the role of variable \( y \), it is reasonable to add a constraint over it. Because we keep the action specifications as weak as possible and that, until then, \( y' \) hasn’t been constrained, the new constraint won’t contradict what was constructed so far.

The discovery of new constraint will be facilitated by their general usage as hypotheses in proofs. The role as hypotheses is made possible by abandoning the predicate transformer semantics and adopting a relational one. Assertions and programs can then be expressed in the same notation, namely predicate calculus; a greater homogeneity is thus attained.

1.2 Transition Systems

A state transition is a tuple \((\mathcal{S}, \mathcal{I}, \rightarrow)\), where:

- \( \mathcal{S} \) is a set of states.
- \( \mathcal{I} \subseteq \mathcal{S} \) is a set of initial states.
- \( \rightarrow \subseteq \mathcal{S} \times \mathcal{S} \) is a binary relation over \( \mathcal{S} \) represent the possible transitions between states.

We are interested in how the execution of transition systems can be captured using computation calculus. Here, we synthesise the notion of states for transitions system and computation calculus, a program \( P \) is a tuple \((i, A)\), where \( i \) is a state predicate representing the set of initial state \( \mathcal{I} \); and \( A \) are the set of atomic actions corresponding to the transitions \( \rightarrow \). The
execution of a transition system $P$ is captured as a conjunction of two different predicate representing the safety computations $saf.P$ and scheduling constraints $sched.P$.

$$[ex.P \equiv saf.P \land sched.P] \quad (1.30)$$

Whilst most transition systems, such as TLA [19], Action Systems [3], UNITY [4], Event-B [1], agree on the definition of set of safety computations captured by $saf.P$, they are different in terms of the scheduling constraints captured by $sched.P$.

The definition for $saf.P$ is as follows.

$$[saf.P \equiv \Diamond i \land G((step.P \lor J);true)] , \quad (1.31)$$

where

$$[step.P \equiv \langle \exists a: a \in A: a \rangle ] ,$$

and $J$ is a special atomic action which does not change the state, i.e. satisfying

$$[J \Rightarrow X]$$

$p;J = J;p$ , for all state predicate $p$ . \quad (1.32)

(1.31) specifies that every computation starts from a state satisfying $i$, and every two consecutive states are supported by an atomic action in $A$ or the special atomic action $J$.

Note that the safety constraint $saf.P$ does not place any restrictions on how often the actions are executed, in particular on skipping. The constraint on the execution frequency of actions are captured by the scheduling constraint $sched.P$, and will be different depending on the transition systems.
Considering only term $\text{saf}.P$ of transition system, we can reason about safety properties of $P$, including the well-known invariance principle.

Informally, in order to prove that $p$ holds for every reachable state of the transition system, we need to show that (0) all initial states satisfy $p$, and (1) every transition maintains $p$. This can be formalised in computation calculus as follows.

**Theorem** (Invariance Principle).

\[
\begin{align*}
[ i \Rightarrow p ] & \land \quad (1.33) \\
[ p; \text{step}.P \Rightarrow X; p ] & \quad (1.34) \\
\Rightarrow \\
[ \text{ex}.P \Rightarrow G \bullet p ]
\end{align*}
\]

**Proof.** Our proof is built mostly around the use of fixed point induction. We $G B; \text{true}$ as the safety constraint of the program we are interested in, with $B$ the disjunction of all actions. For a long stretch of the proof, the exact formulation of the safety constraint does not have a role to play so we substitute it with $u$ for that part, using only the fact that $u$ is persistent.

\[
\begin{align*}
[ \bullet p \land u \Rightarrow G \bullet p ] \\
\Leftarrow \begin{cases} 
\text{induction using the weakest fixed point definition of } G \\
[ \bullet p \land u \Rightarrow (X; (\bullet p \land u) \lor 1) \land \bullet p ]
\end{cases} \\
\Leftarrow \{ \text{predicate calculus} \} \\
[ \bullet p \land u \Rightarrow X; (\bullet p \land u) \}\ \\
= \{ \text{persistence rule} \} \\
[ \bullet p \land u \Rightarrow X; \bullet p ] \\
= \{ u = G B; \text{true}; \text{state restriction} \} \\
[ p; G B; \text{true} \Rightarrow X; \bullet p ]
\end{align*}
\]
\[ \begin{align*}
\{ \text{G is strengthening} \} & \quad [p; B; \text{true} \Rightarrow X; p; \text{true}] \\
\{ \text{monotonicity} \} & \quad [p; B \Rightarrow X; p] 
\end{align*} \]

The above formulation allows us to prove the invariance of a predicate. From the disjunctivity of \( ; \), it follows that the proof obligation can be satisfied individually by each action. Additionally, it is easy to see from (1.32) that \( J \) maintains all invariants. \( \Box \)

1.2.0 Event-B

A model in Event-B [1] typically contains a variable \( v \), invariant \( I(v) \) and several guarded events of the form \( \text{evt} \triangleq \text{when} \ G(v) \text{ then } A(v, v') \text{ end} \), where \( G(v) \) the enabled-ness condition (the guard) and \( A(v, v') \) are the before-after predicate representing the action updating the value of \( v \). A special event \( \text{init} \triangleq \text{begin } A_{\text{init}}(v') \text{ end} \) is used for the initialisation. The representation of an Event-B model in computation calculus takes the set of states to be the values of the variable \( v \). The initialisation \( \text{i} \) corresponds to the initial after value predicate \( A_{\text{init}} \). Finally, each event corresponds to the atomic action \( G; BA \) where \( BA \) is the atomic action corresponding to the before-after predicate \( A(v, v') \).

The conditions (1.33) and (1.34) become \([A_{\text{init}} \Rightarrow I]\) and \([I; G; BA \Rightarrow X; I]\), respectively, which in turns corresponding to standard Event-B proof obligations.
For an Event-B model, say \( P \), containing elements as above, the following scheduling constraint are imposed.

\[
\text{[ sched}_{\text{Event-B}}.P \equiv G(\text{grd}.P; \text{true} \Rightarrow F; \text{step}.P; \text{true}) ] ,
\]

(1.35)

where \( \text{grd}.P \) is the disjunction of all guards of events in \( P \). This states that during the execution of an Event-B model, if at least one event is enabled then eventually an event is executed. In particular, this schedule rules out the situation where an execution stops in a state where an event is enabled.

### 1.2.1 UNITY

An UNITY program \([4]\) contains a set of variables \( v \), an initial-section containing an equality \( v = E_{\text{init}} \) defining initial values of variables, and an assignment-section containing a set of assignment statements. Each assignment statement contain several guarded deterministic assignments as follows.

\[
v := \text{Exp}_1 \text{ if } G_1 \\
v := \text{Exp}_2 \text{ if } G_2 \\
\hspace{1cm} \ldots \\
v := \text{Exp}_n \text{ if } G_n
\]

If none of the guard evaluated to \( \text{true} \) then the statement behaves like \text{skip}. If more than one guard evaluated to \( \text{true} \) then the corresponding assignments must give the same values to the same variables. As a result, every assignment statement is deterministic. The representation of an assignment statement in terms of atomic action is \( G_1; \text{AE}_1 \lor G_2; \text{AE}_2 \ldots \lor G_n; \text{AE}_n \lor \sim \text{Grd}; \text{J} \), where \( \text{Grd} \equiv G_1 \land G_2 \ldots G_n \) and \( \text{AE}_i \) is an atomic action corresponding to assignment \( v := \text{Exp}_i \).

Invariant is one of the fundamental notions in UNITY proof theory, and it follows conditions such as (1.33) and (1.34). In particular, since \( J \) maintains
invariant trivial by doing nothing, the conditions (1.34) will simplify down to prove that each assignment maintain the invariant under its guard condition, i.e. $G_i \land I(v) \Rightarrow I(\text{Exp}_i)$.

The assumption for scheduling constraint of an UNITY program $P$ is that in an infinite execution of $P$, each assignment statement is execute infinitely often. This can be formalised as follows.

$$[\text{sched}_{\text{UNITY}} \equiv \forall a: a \in P: G(F; a; \text{true})]$$  \hspace{1cm} (1.36)
Chapter 2

Temporal Properties

In the following chapter, we introduce an adaptation we made of the UNITY temporal logic. Similar to other temporal logics, the UNITY logic specifies properties of whole computation histories.

For programming, the UNITY logic is very attractive. Instead of building temporal properties out of other temporal properties, like usual temporal logics do, they are being built out of state predicates. Methods like [13] show how successful assertion-based reasoning has been for designing sequential programs and we believe that the UNITY logic allows us to apply the same kind of reasoning for concurrency. In the following, we shall use the word properties to designate temporal properties and the word predicates for state predicates.

In the following sections, we give computation calculus formulations for the operators of UNITY logic. The formulation for programs is given separately in the next chapter.

In UNITY, every property is provided in combination with one program and no meaning independent of the programs is given to properties. Instead, we
have decided to give first a computation calculus formulation for properties, then a formulation for program executions. The statement that a program $F$ satisfies a certain property $P$ is then taken to be an implication. For $\text{ex}.F$ the computation predicate satisfied exactly by the execution histories of $F$, we have

$$[ P \iff \text{ex}.F ]$$

to stand for the UNITY formula

$$P \text{ in } F$$

Additionally, although the contributors to UNITY do not hesitate to make use of meta-proofs —that is, proof by induction over the proof trees of the premises— we shall refrain from doing so and shall provide our own proofs for some of the theorems of UNITY proven in that way. The reason for avoiding meta-proofs is that, in principle, such a proof violates the separation of concerns embodied in the formulation of a theorem; instead of using a theorem on the ground that it is true, it uses it by virtue of its being supported by a proof. It does so at great expenses since it has to make assumptions on the precise the shape of the proof, hence, on the way the logic is axiomatized. In a surprising way, changing the axiomatization for a stronger one can invalidate proofs that are build in this way.

Temporal properties are often divided into two categories: safety properties and progress properties. According to Lamport [18], "safety properties [...] assert that the program does not do something bad" and "liveness properties [...] assert that the program does eventually do something good". A more precise, description from Alpern and Schneider [?] states that safety properties can only be violated by finite computation prefixes, that is to say that
they become violated at a given point in time. In contrast, liveness properties cannot be violated by finite computation prefixes. A consequence of the definition of liveness properties is that, at any given time, there’s always a way to save an execution and make it satisfy a given liveness property.

In the following, we will present a new treatment of the UNITY way of dealing with safety and progress, a special kind of liveness property. Such a class of liveness properties derive their usefulness from their resemblance to the Hoare triples of sequential programming. Techniques pertaining to sequential programming can therefore easily be adapted for concurrency.

2.0 Safety Properties

This section deals with safety properties defined by UNITY’s operators such as co and unless. The unless operator was presented in the publication of [4] which introduced the UNITY logic and the related methods. The unless operator was later replaced by co because the latter has more interesting properties. Although we appreciate the properties of the co operator, we think that unless deserves to be kept for its useful properties. Our use of the computation calculus allows us to be more explicit about the relation between program execution and temporal properties which makes an important distinction between unless and co. Whereas unless is easier to relate to arbitrary sequences of states, co is easier to establish with the shapes of the actions.

2.0.0 The co operator

Definition 14 (The co operator).

\[
[ (p \ co \ q) \equiv G(\bullet p \Rightarrow X; \bullet q) ]
\] (2.1)
Theorems about $co$

We take our theorems from [22]. The purpose is more to show the application of our formalization of UNITY rather than to provide more tools for the application to examples. However, some theorems will prove to be useful.

Statements of Theorems

- $[false\ co\ p]$ (2.2)
  $[(p\ co\ 1) \iff E]$ (2.3)

- Conjunction, disjunction
  
  $[(p \land p' \ co\ q \land q')] \iff (p \ co\ q) \land (p' \ co\ q')$ (2.4)
  $[(p \lor p' \ co\ q \lor q')] \iff (p \ co\ q) \land (p' \ co\ q')$

- Antimonotonicity of left-hand side
  
  $[(p \ co\ q) \Rightarrow (r \ co\ q)] \iff G \bullet (r \Rightarrow p)$ (2.5)

- Monotonicity of right-hand side
  
  $[(p \ co\ q) \Rightarrow (p \ co\ r)] \iff G \bullet (q \Rightarrow r)$ (2.6)

Proofs of Theorems

- Proof of (2.2)
false \co p
= \{ \co \}
G (\bullet \text{false} \Rightarrow X; \bullet p)
= \{ \text{definition of } \bullet (8) \}
G (\text{false}; \text{true} \Rightarrow X; \bullet p)
= \{ \text{left zero of ; } \}
G (\text{false} \Rightarrow X; \bullet p)
= \{ \text{Predicate calculus } \}
G \text{ true}
= \{ G \text{ is universally conjunctional } \}
\text{true}

• Proof of conjunction (2.3)

p \co \one
= \{ (2.1) \text{ then definition of } \bullet (8) \}
G (\bullet p \Rightarrow X; \text{true})
= \{ \text{fixpoint formulation } \}
\langle \forall x:: (X; x \vee \one) \land (\bullet p \Rightarrow X; \text{true}) \rangle
= \{ \text{implication } \}
\langle \forall x:: (X; x \vee \one) \land (\bullet \sim p \lor X; \text{true}) \rangle
= \{ \land \text{ over } \lor \}
\langle \forall x:: (X; x \land \bullet \sim p) \lor (\one \land \bullet \sim p) \lor (X; x \land X; \text{true}) \lor (\one \land X; \text{true}) \rangle
= \{ \text{state restriction and } [ \one \not= X; \text{true } ] \}

37
\[ \forall x:: \sim p; X; x \lor \sim p \lor (X; x \land X; \text{true}) \]
\[ = \{ \text{monotonicity of ; twice} \} \]
\[ \forall x:: X; x \lor \sim p \]
\[ = \{ \text{tail recursion} \} \]
\[ \infty X \lor ^{*}X; \sim p \]
\[ = \{ (1.21) \text{ and (1.22)} \} \]
\[ E \lor F; \sim p \]
\[ \Leftarrow \{ \text{strengthening} \} \]
\[ E \]

• Proof of disjunction (2.4)

We prove the implications by assuming their antecedent and calculating with consequent.

\[ \bullet p \land \bullet p' \]
\[ \Rightarrow \{ \text{Assumptions} \} \]
\[ X; \bullet q \land X; \bullet q' \]
\[ = \{ \text{Abide rule (54)} \} \]
\[ X; (\bullet q \land \bullet q') \]

\[ \bullet p \lor \bullet p' \]
\[ \Rightarrow \{ \text{Assumptions} \} \]
\[ X; \bullet q \lor X; \bullet q' \]
\[ = \{ ; \text{over } \lor \} \]

38
\[ X ; (\bullet q \lor \bullet q') \]

- The proofs of antimonotonicity (2.5) and monotonicity (2.6) are left as exercises.

(end of proofs)

2.0.1 The \textit{unless} operator

The idea of the \textit{unless} properties is one of limited stability: an invariant with an expiry date. \( p \text{ un} q \) holds for the computations which maintain \( p \) starting when it first becomes true and ending just before \( q \) becomes true.

\textbf{Definition 15 (The \textit{un} operator).}

\[
\begin{align*}
\left( p \text{ un} q \right) & \equiv (G(\bullet p \Rightarrow (G \bullet p);S;\bullet q)) \\
\end{align*}
\] (2.7)

The above formulation is almost an exact translation of the idea of \textit{unless}. The only surprise is \( S \), which is a constant of computation calculus we define earlier (see (1.29)). The reason for giving it a name is that we almost never use its exact meaning, only the fact that it is finite. If we replace \( S \) with \( 1 \), that forces \( p \) to hold when \( q \) becomes true and terminates the sequence of \( p \). On the other hand, if we replace \( S \) by \( X \), \( q \) is only considered for terminating the sequence if \( p \) does not hold. It has the consequence that, if \( q \) holds when \( p \) first becomes true, the sequence satisfying \( G \bullet p \). The result is a proper model for the axioms Chandy and Misra adopted to define \textit{unless}.

Relation Between \textit{co} and \textit{unless}

Since \textit{unless} and \textit{co} are safety properties related to different aspects of a computation, namely computation histories and individual steps, it would
be very useful to have a theorem relating them. As a matter of fact, the way Misra and Chandy defined unless in [4] gives us the theorem directly. Since we decoupled the temporal properties from the programs, this is the closest we get the definition of unless given in [4].

Theorem.

\[ (p \text{ un } q) \equiv (p \land \neg q \text{ co } p \lor q) \]  

(2.8)

We prove it by mutual implication and using the computation calculus formulations.

Proof. \textbullet Ping (\(\leftrightarrow\)):

This direction is proven by indirect inequality. We use the fact that both co and un operators yield persistent computations in order to assume persistence with our dummy too (u).

The main calculation is driven by the removal of the G operators. We need to remove them because (0) there are too many of them (1) the inner one is inside a ; expression and no such expression occurs in the computation formulation of co.

\[
\begin{align*}
[u \Rightarrow & \text{G}(\bullet p \Rightarrow (G \bullet p);S;\bullet q)] \\
= & \{ \text{(C8)} \} \\
[u \Rightarrow & (\bullet p \Rightarrow (G \bullet p);S;\bullet q)] \\
= & \{ \text{(C4) and predicate calculus} \} \\
[u \land & \bullet p \Rightarrow \langle \nu x :: p;X;x \land p;S;\bullet q \rangle] \\
\Leftarrow & \{ \text{Induction} \} \\
[u \land & \bullet p \Rightarrow p;X;(u \land \bullet p) \land p;S;\bullet q] \\
= & \{ \text{Persistence rule (C9); (1.29)} \} \\
[u \land & \bullet p \Rightarrow p;X;\bullet p \land p;X;\bullet q \land p;\bullet q] \\
= & \{ \text{Shunting} \}
\end{align*}
\]
\[ u \Rightarrow (\textbullet p \Rightarrow p; X; \textbullet p \land p; X; \textbullet q \land p; \textbullet q) \]
\[ \Leftarrow \{ (2.10) \text{ see below } \} \]
\[ u \Rightarrow (p \land \sim q \co p \land q) \]

In the last step, we appealed to a lemma because its proof does not depend on the general structure of the manipulated formula. Indeed, it allows us to calculate without \( u \Rightarrow \) and without \( \_ \).

\[ \begin{align*}
\textbullet p & \Rightarrow p; X; \textbullet p \lor p; X; \textbullet q \lor p; \textbullet q \\
& = \{ ; \text{ over } \lor \text{ then state restriction } \} \\
\textbullet p & \Rightarrow \textbullet p \land (X; (\textbullet p \lor \textbullet q) \lor \textbullet q) \\
& = \{ \text{ Predicate calculus } \} \\
\textbullet p \land \sim q & \Rightarrow X; (\textbullet p \lor q) \\
& \Leftarrow \{ G \text{ is strengthening } \} \\
G (\textbullet p \land \sim q & \Rightarrow X; (\textbullet p \lor q)) \\
& = \{ (2.1) \} \\
p \land \sim q & \co p \lor q
\end{align*} \]

- Pong \((\Rightarrow)\)

Since the formulae on both sides of the implication are of the form \( Gs \) for some \( s \), we can simplify out proof obligation by stripping away the \( G \). It would be attractive to calculate only with the \( G \) term and
transform it into something else so we massage the demonstrandum to
remove $\bullet p \Rightarrow _\bot$ from the antecedent of the implication we have to
prove.

\[
\left[ G(\bullet p \Rightarrow (G \bullet p); S; \bullet q) \Rightarrow G(\bullet p \land \neg \bullet q \Rightarrow X; (\bullet p \lor \bullet q)) \right]
\leftarrow \{ \text{ monotonicity } \}
\left[ (\bullet p \Rightarrow (G \bullet p); S; \bullet q) \Rightarrow (\bullet p \land \neg \bullet q \Rightarrow X; (\bullet p \lor \bullet q)) \right]
\leftarrow \{ \text{ shunting } \}
\left[ (\bullet p \Rightarrow (G \bullet p); S; \bullet q) \Rightarrow (\bullet p \Rightarrow (\bullet q \lor X; (\bullet p \lor \bullet q))) \right]
\leftarrow \{ \text{ monotonicity of } x \Rightarrow _\bot \}
\left[ (G \bullet p); S; \bullet q \Rightarrow (\bullet q \lor X; (\bullet p \lor \bullet q)) \right]
\]

At this point, we can start calculating with the antecedent of what is
left. Thanks to (C3), we can unfold the computation once and obtain
a formulation closer to the consequent.

\[
(G \bullet p); S; \bullet q
= \{ (C3) \}
\infty(p;X) \lor *(p;X);p;S;\bullet q
= \{ \text{ fix-point (un-)folding and fix-point rolling } \}
p;X;p;\infty(X;p) \lor p;X;p;*(X;p);S;\bullet q \lor S;\bullet q
\Rightarrow \{ \text{ Monotonicity } \}
1;X;p;true \lor S;\bullet q
= \{ (1.29) \}
X;p;true \lor X;\bullet q \lor \bullet q
= \{ ; \text{ over } \lor \}
\]

42
\[ X_i (\bullet p \lor \bullet q) \lor \bullet q \]

\[ \square \]

**Theorems about unless**

**Statements of Theorems**

- **Reflexivity**

  \[ [p \ un \ p] \]
  \[ (p \ un \ \neg p) \iff E \]

  \[ (2.11) \]
  \[ (2.12) \]

- **Monotonicity of left-hand side**

  \[ [(p \ un \ q) \Rightarrow (p \ un \ r)] \iff G \bullet (q \Rightarrow r) \]

  \[ (2.13) \]

- **Conjunction and disjunction**

  \[ [(p \land p' \ un \ (p \land q') \lor (p' \land q) \lor (q \land q'))] \]
  \[ \iff (p \ un \ q) \land (p' \ un \ q') \]

  \[ (2.14) \]

  \[ [(p \lor p' \ un \ (\neg p \land q') \lor (\neg p' \land q) \lor (q \land q'))] \]
  \[ \iff (p \ un \ q) \land (p' \ un \ q') \]

  \[ (2.15) \]

- **Simple conjunction and simple disjunction**

  \[ [(p \land p' \ un \ q \lor q')] \iff (p \ un \ q) \land (p' \ un \ q') \]

  \[ (2.16) \]

  \[ [(p \lor p' \ un \ q \lor q')] \iff (p \ un \ q) \land (p' \ un \ q') \]

  \[ (2.17) \]
• Cancellation

\[ [ (p \lor q \ \text{un} \ r) \iff (p \ \text{un} \ q) \land (q \ \text{un} \ r) ] \]  

(2.18)

Proofs of Theorems

• Proof of reflexivity (2.11)

\[ p \ \text{un} \ p \]
\[ = \{ (2.8) \} \]
\[ p \land \neg p \ \text{co} \ p \lor p \]
\[ = \{ \text{contradiction} \} \]
\[ \text{false} \ \text{co} \ p \]
\[ = \{ (2.2) \} \]
\[ \text{true} \]

• Proof of irreflexivity (2.12)

\[ p \ \text{un} \ \neg p \]
\[ = \{ (2.8) \} \]
\[ p \land p \ \text{co} \ p \lor \neg p \]
\[ = \{ \text{excluded middle} \} \]
\[ p \ \text{co} \ 1 \]
\[ \iff \{ (2.3) \} \]
\[ \text{E} \]

44
• Proof of monotonicity of right-hand side (2.13)
We use the antecedent of both implications as hypotheses and calculate with the consequent. We first strip the outer $G$ (as showed in (1.26)), then continue the proof as follows.

\[
\begin{align*}
\bullet p \\
\Rightarrow & \quad \{ p \text{ un } q \} \\
(G \bullet p); S; \bullet q \\
\Rightarrow & \quad \{ G \bullet (q \Rightarrow r) \} \\
(G \bullet p); S; \bullet r
\end{align*}
\]

• Proof of conjunction (2.14) and disjunction (2.15)
The proofs of (2.14) and (2.15) are very similar and we shall present only the proof of (2.14). The proof of (2.15) is left as an exercise.
We proceed by transforming the consequent into the antecedent.

\[
\begin{align*}
(p \land p') \text{ un } (p \land q') & \lor (p' \land q) \lor (q \land q') \\
= & \quad \{ (2.8) \} \\
& \quad p \land (\neg p \lor \neg q') \land p' \land (\neg p' \lor \neg q) \lor (\neg q \lor \neg q') \\
& \quad \text{co } (p \land p') \lor (p \land q') \lor (p' \land q) \lor (q \land q') \\
= & \quad \{ \text{Complement twice; } \land \text{ over } \lor \} \\
p \land p' \land \neg q' \land \neg q \land (\neg q \lor \neg q') & \quad \text{co } (p \lor q) \land (p' \lor q') \\
= & \quad \{ \text{Absorption} \} \\
p \land \neg q \land p' \land \neg q' & \quad \text{co } (p \lor q) \land (p' \lor q') \\
\Leftarrow & \quad \{ \text{conjunction} \} \\
(p \land \neg q \text{ co } p \lor q) & \land (p' \land \neg q' \text{ co } p' \lor q') \\
= & \quad \{ (2.8) \}
\end{align*}
\]
Proof of simple conjunction (2.16) and simple disjunction (2.17)

The two proofs are very similar and we only present that of (2.16).

We aim at an early application of the previously proved conjunction theorem. Since the antecedent of (2.16) is the same as that of the conjunction rule, we start from there. The remainder is imposed by the shape of the proof.

\[(p \land q) \land (p' \land q')\]

Proof of cancellation (2.18)

For this last proof, we return to computation calculus.

Although it is often the case that a calculation should be attempted by starting with the most complicated side of a relation, in this case, like in many other we encountered, the simplest side of the implication in the consequent –the consequent being \(p \lor q\) \(\lor\) \(r\)– allows us to apply one of the hypotheses which gives an opportunity to apply the second after which all the steps are forced upon us by the shape of the formulae.

\[
(p \land q) \land (p' \land q')
\]

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

\[
p \land p' \land (p \land q') \lor (p' \land q) \lor (q \land q')
\]

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

\[
p \land p' \land q' \lor q \lor (q \land q')
\]

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

\[
p \land p' \land q' \lor q
\]
\( \bullet p \lor \bullet q \)
\[ \Rightarrow \{ p \text{ un } q \} \]

\( \{ G \bullet p \}; S; \bullet q \lor \bullet q \)
\[ \Rightarrow \{ q \text{ un } r \} \]

\( \{ G \bullet p \}; S; (G \bullet q); S; \bullet r \lor (G \bullet q); S; \bullet r \)
\[ \Rightarrow \{ \text{Monotonicity} \} \]

\( \{ G \bullet (p \lor q) \}; S; (G \bullet (p \lor q)); S; \bullet r \lor (G \bullet (p \lor q)); S; \bullet r \)
\[ = \{ (C5) \} \]

\( \{ G \bullet (p \lor q) \}; S; \bullet r \)

Special cases of unless

Here are special cases as given by Misra and Chandy in [4]. They are the notions of stable predicate, designating predicates which remain true once they hold, and of invariants. The use of the latter is unusual because, at the level of temporal properties, no distinction is made between a predicates that holds always and one which is invariant. Predicates that hold always will be obtained from invariants, themselves proved from the text of the programs of interest.

Definition 16.

\[
\begin{align*}
\text{} & [ (st \ p) \equiv (p \text{ un } false) ] & (2.19) \\
\text{} & [ (inv \ p) \equiv G \bullet p ] & (2.20)
\end{align*}
\]

Theorem.

\[
[ (inv \ p) \Leftarrow (st \ p) \land \bullet p ]
\]
Proof.

\[
\text{st } p \wedge \lozenge p \\
= \quad \{ \text{expand definitions } \} \\
\lozenge p \wedge G (\lozenge p \Rightarrow (G \lozenge p); \text{false}) \\
\Rightarrow \quad \{ G \text{ is strengthening } \} \\
\lozenge p \wedge (\lozenge p \Rightarrow (G \lozenge p); \text{false}) \\
\Rightarrow \quad \{ \text{predicate calculus } \} \\
(G \lozenge p); \text{false} \\
\Rightarrow \quad \{ [\text{false} \Rightarrow 1] \} \\
\text{inv } p
\]

\[\square\]

2.1 Progress Properties

This section introduces some operator from UNITY logic for specifying progress properties, namely \textit{leads-to} (\rightarrow), \textit{ensure} (\textit{en}) and \textit{transient} (\textit{tr}). Progress properties assert that the truth of a first predicate is always followed by the truth of a second predicate. It provides us with a means to specify that particular tasks get carried out. This can be specified directly with the \textit{leads-to} operator. Both \textit{ensure} and \textit{transient} operators are special forms of \textit{leads-to}.

2.1.0 The \textit{leads-to} operator

The \textit{leads-to} operator is the most general means to express a progress property in UNITY logic. For programming purposes, it is very useful because, like other temporal properties of the UNITY logic, its statement concern
complete traces but is only built of state predicates. It is therefore interesting to look at leads-to properties as analogous to Hoare triples.

Assignment

\[
\{ p \} \ x := E \{ q \}
\]

The ensures-rule

\[
[p \Rightarrow q] \Leftarrow \ p \text{ en } q
\]

Conditional

\[
\{ p \land (\exists i : b.i) \} \\
\text{if} \ ([i : b.i \rightarrow S.i] \ fi \ \\
\{ q \}
\]\n
\[\Leftarrow (\forall i : \{ p \land b.i \} \ S.i \{ q \})\]

Disjunction

\[
(p \Rightarrow q) \equiv (2.21)
\]

Transitivity

\[
(p \Rightarrow r) \Leftarrow (p \Rightarrow q) \land (q \Rightarrow r)
\]

Sequential composition

\[
\{ p \} \ S0; \ S1 \{ r \} \Leftarrow \{ p \} \ S0 \{ q \} \land \{ q \} \ S1 \{ r \}
\]

Induction

\[
\{ p \} \ \text{do} \ \neg q \rightarrow S \ \text{od} \ \{ p \land q \} \Leftarrow \{ p \land \neg q \} \ S \{ p \} \land \{ p \land \neg q \land M = m \} \ S \{ M < m \}
\]

\[\Leftarrow (p \land M = m \Rightarrow (2.23)) \land (p \land M < m \lor q)\]

It is interesting to notice in the rule for induction that it includes a variant. This is different from what is done in Event-B where the variant is part of the specification, which poses a limitation of one variant per refinement level. In [4], the ensures-rule (which they call basis in [22]), transitivity and disjunction are used as defining axioms for the leads-to operator. In order to take advantage of the our computation calculus framework, we will give leads-to an equivalent formulation and prove that the axioms of Chandy and Misra follow. This way, we can inherit many theorems from UNITY logic.

There is unfortunately a limitation to reaping the benefits of changing the axiomatization of the UNITY logic. Some of the theorems published are in fact meta-theorems: instead of building a deductive proof given the truth of
the premises, their proofs consist of a procedure for building proofs given the
proof of the premises. The procedures do so by reducing all such proofs to a
series of appeals to the axioms. Since we don’t know whether our model is in
fact more powerful than the UNITY logic, we assume it can be and discard
the meta proofs. In their place, we provide calculational proofs.

From the experience gained by reformulating the axioms of the UNITY logic,
we conclude that the computation calculus is much more flexible for working
with temporal properties. This is what we shall use it for. Nevertheless,
we keep the UNITY operators because they are well-suited for programs
to temporal properties. Rather than replacing these operators, we restrict
their usage to what they are most advantageously used for, programming.
In short, UNITY logic and computation calculus nicely complement each
other.

Next, we show a computation calculus formulation for the leads-to operator.
Then we proceed to prove that the UNITY axiomatization follows. Further,
we demonstrate the application of the formulation by proving theorems from
[4] and [22].

**Definition 17** ($\implies$ operator).

\[
[(p \implies q) \equiv \mathbf{G}(\bullet \mathbf{p} \Rightarrow \mathbf{F}; \bullet q)] \tag{2.24}
\]

**Proofs of the UNITY axiomatization**

- Proof of disjunction

\[
(\exists i:: p.i) \implies q
= \{ (2.24) \}
= \mathbf{G}(\bullet (\exists i:: p.i) \Rightarrow \mathbf{F}; \bullet q)
= \{ \bullet \text{ and } (_) \Rightarrow x \text{ over } \exists \}\]
\[
G (\forall i:: \bullet p.i \Rightarrow F; \bullet q)
\]
\[
= \{ G \text{ is universally conjunctive } \}
\]
\[
(\forall i:: G (\bullet p.i \Rightarrow F; \bullet q))
\]
\[
= \{ (2.24) \}
\]
\[
(\forall i:: p.i \mapsto q)
\]

- Proof of transitivity

\[
p \mapsto q \land q \mapsto r
\]
\[
= \{ (2.24) \}
\]
\[
G (\bullet p \Rightarrow F; \bullet q) \land G (\bullet q \Rightarrow F; \bullet r)
\]
\[
= \{ (\land s) \text{ over } G \text{ with persistent } s \} \}
\]
\[
G ( (\bullet p \Rightarrow F; \bullet q) \land G (\bullet q \Rightarrow F; \bullet r) )
\]
\[
\Rightarrow \{ \text{predicate calculus} \}
\]
\[
G (\bullet p \Rightarrow (F; \bullet q \land \text{G ( \bullet q \Rightarrow F; \bullet r)} ))
\]
\[
\Rightarrow \{ \text{substitution} \}
\]
\[
G (\bullet p \Rightarrow F; (\bullet q \land (\bullet q \Rightarrow F; \bullet r)))
\]
\[
\Rightarrow \{ \text{modus ponens and } [F \equiv F; F] \}
\]
\[
G (\bullet p \Rightarrow F; \bullet r)
\]
\[
= \{ (2.24) \}
\]
\[
p \mapsto r
\]

Since the ensures-rule relates to an underlying program, we use an intermediate notion, that of \textit{transient predicates} (to be introduced later), to abstract away from any program. We delay the proof of the base case of the definition of \textit{leads-to} until then.
Theorems about \textit{leads-to}

Statements of Theorems

\begin{itemize}
\item Trading
\[ ((p \to \neg q \lor r) \iff (p \land q \to r)) \quad (2.25) \]
\item PSP
\[ [ (p \land r \to (q \land r) \lor b) \iff (p \to q) \land (r \text{ un} b)] \quad (2.26) \]
\item Cancellation
\[ [ (p \to q \lor r) \iff (p \to q \lor b) \land (b \to r)] \quad (2.27) \]
\item Induction
\[ [ (p \to q) \iff (p \land M = m \to (p \land M < m) \lor q)] \quad (2.28) \]
\end{itemize}

In addition to these, it is easy to see that \textit{leads-to} is antimonotonic in its first argument and monotonic in its second. They will not be proven here.

Proofs of Theorems

\begin{itemize}
\item Trading
\[ \text{G}\{\bullet p \Rightarrow F; (\bullet \neg q \lor \bullet r)\} \]
\[ = \{ ; \text{ over } \lor \} \]
\[ \text{G}\{\bullet p \Rightarrow F; \bullet \neg q \lor F; \bullet r\} \]
\[ \iff \{ [ 1 \Rightarrow F ] \} \]
\[ \text{G}\{\bullet p \Rightarrow \bullet \neg q \lor F; \bullet r\} \]
\end{itemize}
\[
\begin{align*}
&= \quad \{ \text{shunting} \} \\
&G(\bullet p \land \bullet q \Rightarrow F;\bullet r)
\end{align*}
\]

- **PSP** (Progress-Safety-Progress)

Given

\[
\begin{align*}
p &\mapsto q \quad (0) \\
run \ b &\quad (1)
\end{align*}
\]

Prove

\[
p \land b \mapsto (q \land r) \lor b
\]

- \(p \land \bullet r\)
  \[\Rightarrow \quad \{ \text{Hypotheses} \} \]
  \[F;\bullet q \land (G \bullet r);S;\bullet b\]
  \[= \quad \{ (1.11) \} \]
  \[F;\bullet q \land ((G \bullet r \land E) \lor (G \bullet r \land F);S;\bullet b)\]
  \[\Rightarrow \quad \{ \text{weakening then } [F;S \Rightarrow F] \} \]
  \[F;\bullet q \land (G \lor F;\bullet b)\]
  \[\Rightarrow \quad \{ \text{predicate calculus} \} \]
  \[(F;\bullet q \land G \lor F;\bullet b)\]
  \[\Rightarrow \quad \{ \text{substitution} \} \]
  \[F;\bullet (q \land r) \lor F;\bullet b\]
  \[= \quad \{ \text{over } \lor \} \]
  \[F;\bullet ((q \land r) \lor b)\]

- **Cancellation**

Given

\[
p \mapsto q \lor r \quad (0)
\]

53
we want to prove
\[ p \mapsto q \lor b \]

• \( p \mapsto q \lor b \)

\[
\begin{align*}
\Rightarrow & \quad \{ (0) \} \\
F;\bullet(q \lor r) & = \quad \{ \text{; over } \lor \} \\
F;\bullet q \lor F;\bullet r & = \{ (1) \} \\
F;\bullet q \lor F;\bullet b & = \{ F = F;F \} \\
F;\bullet q \lor F;\bullet b & = \quad \{ \text{; over } \lor \} \\
F;\bullet(q \lor b) & 
\end{align*}
\]

• Induction

The following proof is taken from [22]. We have decided to give it a different presentation to show how we could have designed an inductive formulation and it’s proof of validity at the same time.

We want a rule to prove \( p \mapsto q \) by induction. In order to write a proof by induction, we need a domain to induce over. If we have a variant expression, inducing over the value it has in the first argument of the leads-to property seems like a good start. We are therefore going to rewrite our goal as a universal quantification that we will be able to prove by induction.

\[ p \mapsto q \]

54
one point rule, with variant $M$ an expression of the state

\[ p \land (\exists m:: M = m) \rightarrow q \]

\[ \langle \forall m:: p \land M = m \rightarrow q \rangle \]

\[ \langle \forall m:: A.m \rangle \]

\[ A.m \equiv (p \land M = m \rightarrow q) \]

Assuming $m$ ranges over a well founded set, we can prove the last universal quantification by induction. Our proof obligation hence becomes:

\[ \langle \forall m:: A.m \leftrightarrow (\forall n:: n < m:: A.n) \rangle \]

\[ \langle \forall n:: n < m:: A.n \rangle \]

\[ \langle \forall n:: n < m:: A.n \rangle \]

\[ \langle \exists n:: n < m:: p \land M = n \rightarrow q \rangle \]

At this point, if we use our theorems about progress, we need to involve another property in our calculation to land on a formula with $p \land M = m$ as its left-hand side. It also seems reasonable to refuse to introduce a safety property, at least, as long as we haven’t explored other options. This eliminates PSP. The simplest idea is therefore to try transitivity.

\[ \langle \exists n:: n < m:: p \land M = n \rangle \rightarrow q \]

\[ \Rightarrow \{ \text{transitivity with (1)} \} \]
\[ p \land M = m \overset{\Rightarrow}{\iff} q \]
\[ = \{ (0) \} \]
\[ \Lambda.m \]

\[ p \land M = m \overset{\Rightarrow}{\iff} (\exists n: n < m: p \land M = n) \quad (1) \]

(1) is not realistic because it implies that that \( M \) would decrease for infinitely long which is precluded by the fact that \(<\) is well-founded. We can, however, guard the decrease of \( M \) by the falsity of \( q \), as we do in the design of loops in sequential programming.

\[ p \land M = m \land \neg q \overset{\Rightarrow}{\iff} (\exists n: n < m: p \land M = n) \quad (2) \]

\[ (\exists n: n < m: p \land M = n) \overset{\Rightarrow}{\iff} q \]
\[ \Rightarrow \{ \text{transitivity with (2)} \} \]
\[ p \land M = m \land \neg q \overset{\Rightarrow}{\iff} q \]
\[ \Rightarrow \{ \text{trading} \} \]
\[ p \land M = m \overset{\Rightarrow}{\iff} q \]
\[ = \{ (0) \} \]
\[ \Lambda.m \]

Trading is taken from Knapp’s thesis. It is legitimate to wonder if we could not apply trading to (2) before we apply it. Indeed, trading does not change the formula much but it would give us a weaker assumption than (2). That is to say that (2) would still be sufficient but we could prove something weaker and still complete an inductive proof of progress.

\[ p \land M = m \overset{\Rightarrow}{\iff} (\exists n: n < m: p \land M = n) \lor q \quad (3) \]

Transitivity is no longer applicable to make use of (3) in our calcula-
tion; however, it becomes possible to use cancellation instead.

\[(\exists n: n < m: p \land M = n) \mapsto q\]

\[\Rightarrow \{ \text{Cancellation with (3)} \}\]

\[p \land M = m \mapsto q\]

\[= \{ (0) \}\]

\[A.m\]

Being satisfied with (3) as our assumption, we can still simplify it further.

\[(\exists n: n < m: p \land n = M)\]

\[= \{ \text{trading} \}\]

\[(\exists n: n = M: p \land n = M)\]

\[= \{ \text{one point rule} \}\]

\[p \land n = M\]

We find that we only need to assume (4) in order to prove \(p \mapsto q\) inductively.

\[p \land M = m \mapsto (p \land M < m) \lor q\]  \hspace{1cm} (4)

Hence, our rule is:

\[\begin{align*}
(p \mapsto q) & \iff (p \land M = m \mapsto (p \land M < m) \lor q) \\
\end{align*}\]

for \(<\) a well-founded relation.
2.1.1 The ensures operator

In UNITY, ensures properties are a limited form of leads-to properties which can only be implemented with a single action. When they are introduced, they force the design to implement the related progress property in a specific way.

\[ [ (p \rightarrow q) \Leftarrow (p \text{ en } q) ] \]

With the evolution of UNITY logic [22] and the introduction of transient predicates (defined in the next subsection), the mapping between progress properties and programs has been encapsulated one level lower than the ensures operator.

**Definition 18 (en operator).**

\[ [ (p \text{ en } q) \equiv (p \text{ un } q) \land (tr \ p \land q) ] \]

The nice properties of ensures are related to the notion of program union in UNITY. Since we do not have union in Unit-B, they are not properties we are interested in and, instead of stepping through an ensures property when implementing a progress property, we will use directly the following formula.

**The ensures-rule**

\[ [ (p \rightarrow q) \Leftarrow (p \text{ un } q) \land (tr \ p \land q) ] \quad (2.29) \]

2.1.2 The transient operator

A transient predicate is one that is guaranteed to be falsified infinitely often. It is of interest because it does not require any kind of stability to be implemented and its implementation is therefore nicely localized in only in
the scheduling constraint of one action. It is defined as follows.

\[ ([\text{tr } p]) \equiv G F; \bullet \neg p \]  \hspace{1cm} (2.30)

With the above definition, we can proceed to prove the ensures rule. Since the implication in \( p \mapsto q \) and \( p \un q \) have the same antecedent, we start the proof of \( p \mapsto q \) with its antecedent to immediately apply the \textit{unless} property.

\[
\begin{align*}
\bullet p & \\
\Rightarrow & \quad \{ \ p \un q \ \} \\
(G \bullet p); S; \bullet q & = \quad \{ (1.11) \} \\
(G \bullet p \land E) \lor (G \bullet p \land F); S; \bullet q & \Rightarrow \quad \{ \text{weakening with } [S \Rightarrow F] \text{ and } F = F \lor F \} \\
G \bullet p \lor F; \bullet q
\end{align*}
\]

Since no programming model has been introduced yet, proof obligations for the implementation of a transient predicate are left for later.

It is interesting to notice that a special case of \textit{leads-to} property can be translated into an \textit{transient} form more simply than through the \textit{ensures} rule. The proof, done using the computation calculus, is simple and we leave it as an exercise.

\[
\begin{align*}
p \mapsto \neg p & = 1 \mapsto \neg p \\
1 \mapsto \neg p & = \text{tr } p
\end{align*}
\]

\textbf{Special case of the \textit{ensures} rule: induction} \hspace{1cm} Experience has shown us that, in a strengthening chain, applying in succession the induction rule and the \textit{ensures} rule yield complicated formula that can be drastically simplified. Therefore, we have isolated a theorem encapsulating the application both
rules and the ensuing simplifications.

\[ p \mapsto q \]
\[ \equiv \quad \{ \text{induction} \} \]
\[ p \land M = m \mapsto (p \land M < m) \lor q \]
\[ \equiv \quad \{ \text{ensure} \} \]
\[ (p \land M = m \quad \text{un} \quad (p \land M < m) \lor q) \]
\[ \land \quad \text{tr} \quad (p \land M = m \land \sim ( (p \land M < m) \lor q )) \]
\[ = \quad \{ \text{translate un into co (2.8)} \} \]
\[ (p \land M = m \land \sim ( (p \land M < m) \lor q )) \]
\[ \land \quad \text{tr} \quad (p \land M = m \land \sim ( (p \land M < m) \lor q )) \]

The above yields a very unattractive formula. However, we can notice that one of the arguments of the temporal operators is duplicated so we only have two predicates to simplify.

\[ p \land M = m \land \sim ( (p \land M < m) \lor q ) \]
\[ = \quad \{ \text{predicate calculus} \} \]
\[ p \land M = m \land m \leq M \land \sim q \]
\[ = \quad \{ \leq \text{is reflexive} \} \]
\[ p \land M = m \land \sim q \]

\[ (p \land M = m) \lor (p \land M < m) \lor q \]
\[ = \quad \{ \text{predicate calculus and } \leq \} \]
\[ (p \land M \leq m) \lor q \]

\[ p \land M = m \land \sim q \quad \text{co} \quad (p \land M \leq m) \lor q \]
\[
\iffalse
type\{\text{ antimonotonicity of lhs }\}
\]
\[
p \land M \leq m \land \neg q \co (p \land M \leq m) \lor q
\]
\[
= \iffalse\{\text{(2)}\}\fi
p \land M \leq m \un q
\]

We can therefore conclude:

\[
\iffalse\{\text{(p \rightarrow q)}\}\fi
\iffalse\{\text{(p \land M \leq m \un q)}\}\fi \land \iffalse\{\text{(tr p \land M = m \land \neg q)}\}\fi \iffalse(2.32)\fi
\]

61
Chapter 3

The Unit-B Method

This chapter introduce Unit-B method, inspired by UNITY [4] and the Event-B [1] methods. Similar to both fore-mentioned methods, a program in Unit-B represents a transition systems and its execution can be defined using computation calculus. On the one hand, while Event-B is designed for developing systems using refinement preserving properties, it does not handle progress properties. In particular, the refinement notion in Event-B does not necessarily preserve progress properties. On the other hand, temporal properties (both safety and liveness) are included in UNITY, by a comprehensive program logic. However, the stepwise development process using refinement is not present in UNITY. Our proposed Unit-B method intends to combine the strength of Event-B and UNITY, to fill the missing pieces of these methods. The result is a refinement based stepwise development process handling both safety and progress properties.

Being a transition system, an Unit-B program contains a set of variables \( v \), an initialisation represented by a state predicate \( i \), and a set of atomic actions \( A \). As before, an execution of an Unit-B program \( P \) satisfying a safety constraint \( \text{saf}.P \) and a scheduling constraint \( \text{sched}.P \), which can be
expressed as follows.

\[ \text{ex.P} \equiv \text{saf.P} \land \text{sched.P} \], \quad (3.1) \]

Where saf.P is defined as in (1.31).

What makes Unit-B different from UNITY and Event-B is the scheduling constraint, for which some additional annotation will be defined. We progress by reviewing some existing scheduling convention from UNITY, then introduce the actual scheduling constraint for our Unit-B programs depending on the additional annotations.

### 3.0 Existing Scheduling Policies in UNITY

The scheduling constraint enforces the required frequency for the execution of individual actions. Contrarily to [10], they do not constrain what happens in between. Safety constraint are sufficient to rule out the execution of undesired actions. The result is a better separation of concerns and is in line with Misra’s addition of the notion of transient predicates to UNITY [22].

In the literature [22], one can distinguish between two main execution policies. They are weak fairness and strong fairness\(^1\). Although we chose to depart from that practice, our solution is strongly inspired by these notions.

### 3.0.0 Weak Fairness

The execution of actions in UNITY is done according to weakly fair (wf-) schedules. It has the advantage of being relatively easy to implement: it can be implemented naively by testing the guard of each wf-action infinitely often and executing them when their guard hold. For this reason, it can

\(^1\)There is also minimal progress but we will not consider in our context.
be kept in a low level design specification without creating doubts as to its feasibility.

Assuming that an guarded action in UNITY is of the simple form $g \rightarrow a$, the fact that the action is weakly-fair scheduled is captured by $\text{wf.}(g \rightarrow a)$ and is defined as follows.

$$\text{wf.}(g \rightarrow a) \equiv \text{G}(\text{G} \cdot g \Rightarrow \text{G} \cdot F; a; \text{true})$$

(3.2)

From UNITY, we know what premises we can use to prove that an action satisfies a transient predicate $\text{tr } p$ (defined in Section 2.1.2).

**Theorem.**

\[
\text{tr } p
\]

(3.3)

provided

$$\text{wf.}(g \rightarrow a)$$

(0)

$$\left[ (p \land g); a \Rightarrow X; \neg p \right]$$

(1)

$$\text{G} (\bullet p \Rightarrow \bullet g)$$

(2)

**Proof.** We first drop the outer $\text{G}$ (showed in (1.26)) and continue our computation as follows.

$$\text{F}; \neg p; \text{true}$$

$$\Leftarrow \{ \text{unfold } \text{F}, \text{ aiming for (1) } \}$$

$$\text{F}; X; \neg p; \text{true}$$

$$\Leftarrow \{ (1) \}$$

$$\text{F}; (p \land g); a; \text{true}$$

$$\Leftarrow \{ \text{persistence } \}$$

$$\text{F}; a; \text{true} \land \text{G} \cdot g \land \text{G} \cdot p$$

$$\Leftarrow \{ (0) \}$$
As will be seen later, this notation lends itself to the important refinement technique of guard strengthening. It has been pointed out in [21].

\[ G \land G \cdot p = \{ (2) \} \]

3.0.1 Strong Fairness

Strong fairness has been initially left out of UNITY to be lightly covered in [22]. It can be stated as:

\[ [ sf.(g \rightarrow a) \equiv G (GF; \cdot g \Rightarrow GF; g; a; true) ] \quad (3.4) \]

This constraint gives us more flexibility: the guard does not have to hold forever to force the execution of the action. It would be sufficient to use the same proof obligation for \( tr \) as with weak fairness but we would fail to reap the benefits of the strengthened assumption. We can use the shape of this new formula dictate what the proof obligation might look like.

We decide to prove \( tr \ p \) as \( GF; \cdot \sim p \). It reasonable to assume the hypotheses will of the form \( G s \) and to remove \( G \) everywhere.

\[ [ (tr \ p) \leftrightarrow sf.(g \rightarrow a) ] \]

We can already try with the following assumption, for which there does not seem to be much choice.

\[ [ g; a \Rightarrow X; \sim p ] \quad (3.5) \]
A possible attempt is as follows.

\[ F; \neg p; \text{true} \]
\[ \iff \{ \text{unfold } F \text{ heading for (3.5)} \} \]
\[ F; X; \neg p; \text{true} \]
\[ \iff \{ (3.5) \} \]
\[ F; g; a; \text{true} \]
\[ \iff \{ \text{antecedent} \} \]
\[ G F; \bullet g \]

The last formula of the above calculation suggests that we use \( p \mapsto g \) as an additional hypothesis.

\[ p \mapsto g \tag{3.6} \]

Its use leads us to the negation of the formula we started calculating with. This is a form of proof by contradiction.

\[ G F; \bullet g \]
\[ \iff \{ p \mapsto g \text{ (3.6)} \} \]
\[ G \bullet p \]
\[ \iff \{ \text{Dual of } G \} \]
\[ \neg (F; \neg p; \text{true}) \]

The proof obligation we finally discovered is unattractive. The problem is that, although it allows us to prove progress properties using sf-actions, strengthening the guard is very tricky. From a computation calculus point of view, it is explained by the fact that \( g \) appears both in a monotonic and antimonotonic context. From a UNITY logic point of view, we expect that refinement preserves transient predicates but strengthening the guard would
have the consequence of invalidating all the proof of obligations of the kind of (2). Also, for guard h, we would inherit a whole class of proof obligation of the shape \( p \mapsto h \) for all transient \( p \). This is especially tricky because the transient predicates are not encoded as such in the action systems. It is therefore hard to deduce the new proof obligations to maintain transient predicates. To remedy this problem, we chose to bring strong fairness and weak fairness closer together. This is what general scheduling is all about.

### 3.1 General Scheduling

Recall the definition of weak and strong fairness assumptions for a guarded action \( g \rightarrow a \) as follows.

\[
\begin{align*}
\text{s.f.}(g \rightarrow a) & \equiv G GF; \top \Rightarrow GF; g; a; \top \\
\text{w.f.}(g \rightarrow a) & \equiv G G \cdot g \Rightarrow GF; a; \top
\end{align*}
\]

We are going to combine these definitions into a single one, allowing us to manipulate these conditions in a more flexible fashion. Moreover, we want to separate the actual guard from these fairness constraints, so that we can define refinement preserving progress properties.

First of all, we extend our Unit-B notation to include additional annotations on how individual action can be scheduled. For each action \( a \) of an Unit-B programs, we associated with two state predicates: \( c \) standing for coarse-schedule and \( f \) standing for fine-schedule. As a result, each action in our Unit-B programs will be represented by a tuple \((a, c, f)\), containing the atomic action \( a \) and its corresponding schedule information \( c \) and \( f \). Note that the atomic action \( a \) can be guarded (which we will take into account later).

Given an action of the form \((a, c, f)\), we define the general scheduling of the
action denoted by $sc.(a,c,f)$ as follows.

$$[sc.(a,c,f) \equiv G(G \cdot c \land GF;\bullet f \Rightarrow GF;f;a;true)] \quad (3.7)$$

Note that

$$[sc.(a,c,1) \equiv G(G \cdot c \Rightarrow GF;f;a;true)]$$

which is the same as $wf.(c \rightarrow a)$. Similarly,

$$[sc.(a,1,f) \equiv G(GF;\bullet f \Rightarrow GF;f;a;true)]$$

which is the same as $sf.(f \rightarrow a)$.

In the case where $c \mapsto f$, the general schedule is reduce to what we called a magical scheduling $msc.(a,c,f)$.

$$[msc.(a,c,f) \equiv G(G \cdot c \Rightarrow GF;f;a;true)] \quad (3.8)$$

In fact, the following proof ensures that

$$[c \mapsto f \Rightarrow (msc.(a,c,f) \equiv sc.(a,c,f))] \quad (3.9)$$

Proof. We prove $msc.(a,c,f) \equiv sc.(a,c,f)$ under the assumption $c \mapsto f$ by ping-pong argument

• Ping:

$$sc.(a,c,f)$$

$$= \quad \{ \text{definition (3.7)} \}$$

$$G(G \cdot c \land GF;\bullet f \Rightarrow GF;f;a;true)$$

$$\Leftarrow \quad \{ \text{strengthen, } G \text{ is monotonic} \}$$

$$G(G \cdot c \Rightarrow GF;f;a;true)$$

$$= \quad \{ \text{definition (3.8)} \}$$

$$msc.(a,c,f)$$

68
• Pong:

\[ \text{msc.}(a, c, f) \]
\[ = \{ \text{definition (3.8)} \} \]
\[ G (G \cdot c \Rightarrow GF; f; a; \text{true}) \]
\[ = \{ G \cdot c \Rightarrow GF; f, \text{see below} \} \]
\[ G (G \cdot c \land GF; f \Rightarrow GF; f; a; \text{true}) \]
\[ = \{ \text{definition (3.7)} \} \]
\[ \text{sc.}(a, c, f) \]

We now prove that \( G \cdot c \Rightarrow GF; f \)

\[ G \cdot c \Rightarrow GF; f \]
\[ \Leftarrow \{ G \text{ is monotonic} \} \]
\[ \cdot c \Rightarrow F; f \]
\[ \Leftarrow \{ G \text{ is strengthening} \} \]
\[ c \mapsto f \]

\[ \square \]

### 3.2 The Unit-B Notation

This section introduces the “pretty-print” syntax of an Unit-B program, inspired by Event-B. Each Unit-B program has a set of variables \( v \) and a set of actions \( A \). Each action \( a_i \in A \) is of the following form

\[ a_i \doteq \text{when } g_i \text{ during } c_i \text{ upon } f_i \text{ then } A_i \text{ end} \]
where \( g_i \) is the guard (the enable condition), \( c_i \) is the coarse schedule, \( f_i \) is the fine schedule and \( A_i \) is the before-after predicate describing how variables \( v \) are modified. \( g_i, c_i, f_i \) are state predicates over \( v \), whereas \( A_i \) is a predicate over \( v \) and its after value \( v' \). We omit during (resp. upon) when there is no \( c_i \) (resp. \( f_i \)). In this case, i.e. when they are omitted, we assume they are \( \top \). Similarly, when \( g_i \) is omitted, we assume that it is \( \top \) and change the keyword when to begin. According the convention, the following action \( a_i \triangleq \text{begin } A_i \text{ end} \) is scheduled infinitely often.

As mentioned earlier, the execution of an Unit-B program \( P \) is captured by a safety part and a scheduling part, i.e.

\[
\text{[ ex.P } \equiv \text{saf.P } \wedge \text{sched.P }] .
\]  

(3.10)

In particular

\[
\text{[ saf.P } \equiv \bullet A_{\text{init}} \wedge G (\text{step.P } \vee J; \text{true}) ] ,
\]

where \( \text{step.P } \equiv \langle \exists i:: g_i; A_i \rangle \). Note that \( \text{saf.P} \) does not depend on the coarse and fine schedule of individual action.

The scheduling constraint \( \text{sched.P} \) is defined as follows.

\[
\text{[ sched.P } \equiv \langle \forall i:: \text{sc.}(g_i; A_i, c_i, f_i) \rangle ] ,
\]  

(3.11)

i.e. each action is scheduled according to its coarse and fine schedule.

Under additional condition that

\[
\text{[ saf.P } \Rightarrow G \bullet (c_i \wedge f_i \Rightarrow g_i) ] ,
\]  

\text{(FIS)}

the schedule can be simplified to

\[
\text{[ sched.P } \equiv \langle \forall i:: \text{sc.}(A_i, c_i, f_i) \rangle ] ,
\]


3.3 Verification Conditions

This sections provide some verification conditions for ensuring an Unit-B program satisfies certain properties. We adopt the convention in Event-B of naming the conditions for traceability. We are going to give labels to properties and use them in the label of the condition.

The verification conditions are expressed in computation calculus for invariance properties, co properties and tr properties. un properties are translated into co properties and leads to properties are translated into tr and un properties using rules shown in Section 2.

3.3.0 Invariance

The invariance proof principle is well-known and is already mentioned before in Section 1.2. In order to prove that \( I \) is an invariant of a program \( P \), i.e. \( \text{ex.} P \Rightarrow G \bullet I \), we prove \( \text{saf.} P \Rightarrow G \bullet I \) with the following conditions.

\[
\begin{align*}

[A_{\text{init}} &\Rightarrow I ] \quad \text{(3.12)} \\
[I; g_i; A_i &\Rightarrow X; I ], \text{ for every event } a_i \quad \text{(3.13)}
\end{align*}
\]

Typically \( I \) will be a conjunction of several sub-invariants \( I = I_1 \wedge I_2 \wedge \ldots \), we prove the each sub-invariant independently. Let \( \text{inv}_j \) be the label for invariant \( I_j \), the following conditions are

\[
\begin{align*}

[A_{\text{init}} &\Rightarrow I_j ], \text{ for every invariant } I_j \quad \text{(inv}_j/\text{INIT}) \\
[I; g_i; A_i &\Rightarrow X; I_j ], \text{ for every event } a_i, \text{ invariant } I_j \quad \text{(a}_i/\text{inv}_j/\text{INV})
\end{align*}
\]
3.3.1 co Properties

In order to prove that a program $P$ satisfies a co properties of the following form $p \text{ co } q$, i.e. $\text{ex}.P \Rightarrow p \text{ co } q$, we prove $\text{saf}.P \Rightarrow p \text{ co } q$ (i.e. relying on the safety constraint of $P$ only). Let $co$ be the label of the property $p \text{ co } q$, the following proof obligation ensures that program $P$ satisfies $co$.

$$[p; g_i; A_i \Rightarrow X; q], \text{for every event } a_i \quad (a_i/\text{co}/\text{CO})$$

The soundness proof of for this obligation is similar to the proof of invariance principle in Section 1.2 and is left out here.

Moreover, we can make use of any invariant in the proof of co properties. Assume that $I$ is an invariant, i.e. we have $[\text{saf}.P \Rightarrow G \bullet I]$, then the condition $a_i/\text{co}/\text{CO}$ becomes

$$[I; p; g_i; A_i \Rightarrow X; q], \text{for every event } a_i \quad (a_i/\text{co}/\text{CO})$$

3.3.2 Transient Properties

A proof of a transient property relies on both safety and scheduling constraints of the program $P$. Let $tr$ be the label of the property $tr \ p$, we want to use an action $a_i \triangleq \text{when } g_i \text{ during } c_i \text{ upon } f_i \text{ then } A_i \text{ end}$ to implement $tr$. The following proof obligations ensure that program $P$ satisfies $tr$.

$$[\text{ex}.P \Rightarrow G \bullet (p \Rightarrow c_i)] \quad (a_i/\text{tr}/\text{SCH})$$
$$[p; c_i \wedge f_i; A_i \Rightarrow X; \neg p] \quad (a_i/\text{tr}/\text{NEG})$$
$$[\text{ex}.P \Rightarrow c_i \mapsto f_i] \quad (a_i/\text{OP})$$

Condition $a_i/\text{tr}/\text{SCH}$ stating that $p \Rightarrow c_i$ is an invariance property and can be proved by the invariance induction rule or proved that it is derivable from other invariants.
Condition $a_i/\text{tr}/\text{NEG}$ can take into account invariance properties I and can be stated as follows.

$$\left[ I; p; c_i \land f_i; A_i \Rightarrow X; \neg p \right] \quad (a_i/\text{tr}/\text{NEG})$$

Condition $a_i/\text{OP}$ is independent of the transient property tr. Practically, this is usually proved when the action is introduced. In particular, it becomes trivial in the case where $f_i$ is 1 or is the same as $c_i$.

**Proof.** We prove that if a program $P$ satisfies transient property tr $p$, i.e. $[\text{ex.P} \Rightarrow \text{tr} \ p ]$, if there exist and event $a_i \triangleq \text{when g}_i \text{ during c}_i \text{ upon f}_i \text{ then } A_i \text{ end}$ in $P$ satisfying conditions $a_i/\text{tr}/\text{SCH}$, $a_i/\text{tr}/\text{NEG}$, $a_i/\text{OP}$.

First of all, we prove that under $a_i/\text{OP}$, ex.$P$ yields msc.$(A_i, c_i, f_i)$.

$$\text{msc.}(BA_i, c_i, f_i) = \quad \{ a_i/\text{OP} \text{ and (3.9)} \}$$

$$\text{sc.}(BA_i, c_i, f_i) \ \iff \quad \{ 3.11 \}$$

$$\text{sched}.P \ \iff \quad \{ 3.10 \}$$

$$\text{ex}.P$$

We complete the proof by proving $[ \text{msc.}(BA_i, c_i, f_i) \Rightarrow \text{tr} \ p ]$ under the assumption $G \bullet (p \Rightarrow c_i) \ (a_i/\text{tr}/\text{SCH})$ and $[ (p \land c_i \land f_i); BA_i \Rightarrow X; \neg p ]$ $(a_i/\text{tr}/\text{NEG})$. Note that both msc and tr $p$ is formulate with $G$, hence we can use monotonicity, i.e. to prove $F; \neg p; \text{true}$ under assumption $(G \bullet c_i \Rightarrow GF; f_i; BA_i; \text{true})$. Moreover, $G \bullet p = \neg(F; \neg p; \text{true})$ hence we can use $G \bullet p$ as an additional assumption.
\[
F; \neg p; \text{true} \\
\Leftarrow \quad \{ F; X \Rightarrow F \} \\
F; X; \neg p; \text{true} \\
\Leftarrow \quad \{ \text{ai/tr/NEG} \} \\
F; (p \land c_i \land f_i); BA_i; \text{true} \\
= \quad \{ G \bullet p \} \\
F; (c_i \land f_i); BA_i; \text{true} \\
\Leftarrow \quad \{ \text{persistence and } G \text{ is strengthening} \} \\
G F; f_i; BA_i; \text{true} \land G \bullet c_i \\
= \quad \{ \text{msc.}(BA_i, c_i, f_i) \} \\
G \bullet c_i \\
\Leftarrow \quad \{ G \bullet (p \Rightarrow c) \} \\
G \bullet p
\]
Chapter 4

Refinement Laws

This thesis promotes the use of refinement as the main technique for designing programs. We define refinement between specifications and consider programs to be a special case of specifications. Until now, the notion of program has been explored in depth but specifications have been no more than mentioned. For our purposes, we will take a specification to be the conjunction of the execution of one program with a set of temporal properties. When the set of properties is empty, a specification is also a program.

Refinement laws give the directions to developments but they can be applied with a wide variety of styles. Although they could be used as verification conditions checking that a specification refines another specification, this is not what we choose to do. We choose to use the laws to calculate refinements. The nuance is subtle but the consequences are important. The first style uses the laws to answer the question: “Given two specifications, is the first a refinement of the second?” The second style aims at answering the question: “Given a specification, what would a specification refining it would look like?”.

We take implication (⇐) between two specifications to be our refinement
order with the stronger side being the most concrete specification and the weaker side being the most abstract. It is important to realize that the refinement rules are implications and, although it would be understandable to see some terms of the antecedent as side conditions, they should be thought of as being on the strong side of the implication. They can sometimes be discharged immediately—that is, proven to hold about the concrete specification—but it is not necessary to do so. They can be added to the specification and integrated into the program at a later point. This allows us to apply refinement rule to satisfy the “side conditions” of a previous application of a refinement rule.

Specifications are built as conjunctions. It is therefore possible to refine elements in isolation. We can refine the safety constraint on its own, independently of the rest or relying on the properties of the rest of the program or to refine the scheduling constraint of an action on its own or simply to strengthen accompanying temporal properties.

4.0 Safety Refinement

For refinement preserving safety properties, one can strengthen the guard and the assignment of the corresponding action. This results in two proof obligations as follows.

\[
\begin{align*}
\text{ex.}Q \Rightarrow G \cdot (h \Rightarrow g) & \quad \text{(GRD)} \\
\text{ex.}Q \Rightarrow (A \Rightarrow B) & \quad \text{(SIM)}
\end{align*}
\]

It is also possible to add an action, as long as it refines skip. The skip introduce previously is a bit strong to allow it. Instead, we use a family of skips parameterized in the program variables of which they preserve the value. The the variables for which the values are not preserved are left
for $p$ a predicate in variables $v$. In short, new actions have to not modify variables from previous refinements.

### 4.1 Progress Refinement

#### 4.1.0 Weakening Coarse Schedule

Because of it’s position in the scheduling formulation, it is very easy to weaken a coarse schedule. Given

$$G (G \cdot b \land G F; \cdot f \Rightarrow G F; f; A; true)$$

to weaken the coarse schedule $(c)$, we only need to rely on the antimonotonicity of its place in the formula.

#### 4.1.1 Replacing a Coarse Schedule

Some work has been done in UNITY on the problem of strengthening the guard of an action notably by Misra and by Singh [?, 21, 23]. Let’s recall that, in UNITY, the guard and the schedules of an action are the same thing and the difficulties encountered with strengthening a guard in UNITY come from the desire to preserve progress. For us, [21] is the most interesting because, unlike the other two, it allows us to preserve progress properties without knowing about them. Indeed, in the other two notes, the proof obligation is specially tailored to each progress property one wants to preserve. Since our
notion of refinement has to preserve all progress properties, we have to leave them anonymous.

In [21], the theorem presented is defended by a meta-proof. We shall instead provide a proof using our framework. We design our proof by using Misra’s statement of the theorem loosely. By starting with no assumptions, we shall add the ones we make use of.

The goal is to refine

$$G ( G \bullet g \land G F; \bullet f \Rightarrow G F; f; A; true )$$

into

$$G ( G \bullet h \land G F; \bullet f \Rightarrow G F; A; true ) \quad (0)$$

$$G \bullet g \land G F; \bullet f$$

$$\Rightarrow \quad \{ (1) \text{ see below } \}$$

$$F; ( G \bullet h ) \land G F; \bullet f$$

$$\Rightarrow \quad \{ \text{ persistence } \}$$

$$F; ( G \bullet h \land G F; \bullet f )$$

$$\Rightarrow \quad \{ (0) \}$$

$$F; ( G F; f; A; true )$$

$$= \quad \{ (C13) \}$$

$$G F; f; A; true$$
We can see that we only needed two more assumptions to complete the proof as opposed to what is shown in [21] where they need also that the new schedule be stronger than the old one. We don’t use it because we have decoupled the notion of schedule and guard and our schedule to safety only
weakly.

\[
g \mapsto h \quad (2)
\]

\[
h \un \sim g \quad (3)
\]

4.1.2 Strengthening a Fine Schedule

It is possible, at no small costs, to strengthen the fine schedule of an action. Given

\[
G (G \cdot c \land G F; \cdot f \Rightarrow G F; f; A; true)
\]

We want to refine it into:

\[
G (G \cdot c \land G F; \cdot g \Rightarrow G F; g; A; true)
\]

\[
G (G \cdot c \land G F; \cdot f \Rightarrow G F; f; A; true) \quad \Leftarrow \quad \{ \text{strengthening} \}
\]

\[
G (G \cdot c \Rightarrow G F; f; A; true)
\]

\[
\Leftarrow \quad \{ G \cdot (g \Rightarrow f) \}
\]

\[
G (G \cdot c \Rightarrow G F; g; A; true)
\]

\[
\Leftarrow \quad \{ \text{remove magic} \}
\]

\[
G (G \cdot c \land G F; \cdot g \Rightarrow G F; g; A; true) \land (c \mapsto g)
\]
4.1.3 Removing a Fine Schedule

\[ G (G \bullet c \land G F;\bullet f \Rightarrow G F;f;A;true) \]

We want to refine it into:

\[ G (G \bullet c \Rightarrow G F;A;true) \]

To allow us to do so, we need:

\[ G \bullet(c \Rightarrow f) \]

\[
\begin{align*}
G \bullet c \land G F;\bullet f & \Rightarrow G F;f;A;true \\
= & \{ G \bullet(c \Rightarrow f) \} \\
G \bullet c \land G \bullet f \land G F;\bullet f & \Rightarrow G F;f;A;true \\
= & \{ \text{substitution} \} \\
G \bullet c \land G \bullet f & \Rightarrow G F;A;true \\
= & \{ G \bullet(c \Rightarrow f) \} \\
G \bullet c & \Rightarrow G F;A;true
\end{align*}
\]

4.1.4 Mergin Actions

It can be useful to merge two actions into one. For example, if no fair scheduler can be relied upon, it is necessary that the proper coordination of
actions be assured. It would be the case if the program under development was itself a scheduler. In this case, all that we can assume from the processor executing the program is that it will choose and execute actions infinitely often but without regards for fairness. In such a situation, if there is only one action to choose and that it implements a proper coordination, an unfair scheduler will be good enough.

The effect of merging two actions for the safety constraint is nil since a demonic choice is expressed as a disjunction, the same operator used to unite any pair of actions in said constraint.

The scheduling constraint, however, requires us to do more work.

For simplicity, we will ignore the case where the actions have fine schedules. It does not seem like a useful case to consider because, from a methodological perspective, in the advent that it would be, it is always possible to come back and prove a new form of this refinement law.

Below is the scheduling constraint of the two actions under consideration.

\[
\begin{align*}
&G (G \bullet g \Rightarrow G F; A; true) \quad (0) \\
&G (G \bullet h \Rightarrow G F; B; true) \quad (1)
\end{align*}
\]

We will form their common scheduling constraint by first finding a common formulation to the consequents of the above implications and then by merging the antecedents.

We chose to impose a strong restriction on the schedules to allow us to effectively merge the actions. It is required for them to be disjoint.
\[ G \bullet (\neg g \lor \neg h \lor (\neg e \land \neg f)) \] (2)

\[
f; A
= \{ G \bullet g \}
(f \land g); A
= \{ \text{identity of } \lor \}
(f \land g); A \lor \text{false}
= \{ \text{left zero of } ; \}
(f \land g); A \lor \text{false}; B
= \{ G \bullet g \text{ and (2) as } g \Rightarrow (e \land h \equiv \text{false}) \}
(f \land g); A \lor (e \land h); B
\]
\[
\Leftarrow \{ e \text{ and } f \text{ are state predicates} \}
(f \lor e) ; ( (f \land g); A \lor (e \land h); B )
\]

A similar calculation allows us to transform the scheduling constraint of B. Since it does not appear in the remainder of the calculations, we label the consequent we have obtained with P and substitute it in the formulae we will manipulate.

\[
(G \bullet g \land GF; \bullet f \Rightarrow P) \land (G \bullet h \land GF; \bullet e \Rightarrow P)
\]
\[
\Leftarrow \{ \text{predicate calculus} \}
(G \bullet g \lor G \bullet h) \land (GF; \bullet f \lor GF; \bullet e) \Rightarrow P
\]
\[
\Leftarrow \{ \text{monotonicity} \}
\]

83
This tells us that provided the schedules are disjoint, we can merge two actions by guarding their effects with their respective guards. The action resulting from merging $A$ and $B$ is therefore $(g \land f); A \lor (h \land e); B$, the new coarse schedule is taken to be the disjunction of that of the respective actions and similarly for the fine schedule.

### 4.1.5 Special Case of Schedule Strengthening

We have seen in the previous section that, in order to merge to actions, we need their coarse schedule to be disjoint. At first, sight, one might think that it is possible to change a program to make the schedules disjoint by strengthening the schedules. Unfortunately, the rule we provided for replacing a coarse schedule makes sure that the new schedule holds for at least as long as the one it replaces. In the case of an action for which the schedule holds forever, the replacement rule would not allow us to disable the action to enable another action.

For that purpose, we have designed the round-robin rule. The first purpose for doing so is to make it possible to implement our fair transition systems using an unfair implementation. However, with this rule, we wish to convey the message that there are many possibilities for refining programs and that one does not have to limit oneself to the refinement rules we have provided.

The idea behind this rule is that, although we can change a coarse schedule to make it interruptible, it must still force the execution of the corresponding actions with the same frequency. To do so, we require the new schedule to
hold infinitely often in the conditions where the old one would hold forever. Additionally, in order to still rely on weak fairness only, we require that the only way to falsify the new schedule when the old one holds is to execute the associated actions. In other words, provided the old schedule holds forever, either the action is executed infinitely often or the new schedule holds forever.

\[ GF; A; true \lor F; G (p \Rightarrow (G \cdot p); \cdot \sim g) \]  

\[ g \mapsto p \]  

\[ G \cdot g \]

\[ = \{ (0) assuming \neg GF; A; true \} \]

\[ G \cdot g \land F; G (p \Rightarrow (G \cdot p); \cdot \sim g) \]

\[ \Rightarrow \{ \text{persistence rule} \} \]

\[ F; (G \cdot g \land G (p \Rightarrow (G \cdot p); \cdot \sim g)) \]

\[ = \{ G \text{ is strengthening} \} \]

\[ F; (G \cdot g \land F \cdot p \land G (p \Rightarrow (G \cdot p); \cdot \sim g)) \]

\[ \Rightarrow \{ \text{persistence rule} \} \]

\[ F; (G \cdot g \land F; (G \cdot p); \cdot \sim g) \]

\[ \Rightarrow \{ \text{persistence and (1.9)} \} \]

\[ F; (F; (G \cdot p) \land E) \]

\[ \Rightarrow \{ (2) and weakening \} \]

\[ F; (GF; A; true) \]

\[ = \{ (C13) \} \]
\[
G \cdot p \Rightarrow G \cdot F \cdot A \cdot \text{true}
\]

(2)

We would like to know under what conditions a program fulfills condition (0). Because of the second disjunct, we will treat it as safety property. To make it easier to prove \([p \Rightarrow q \lor s]\), we will shunt it and prove \([p \land \neg q \Rightarrow s]\) starting with its left-hand side. We will use \(B\) as the disjunction of all the actions that are not \(A\). We can therefore use \(G\ (A \lor B); \text{true}\) as our safety constraint.

\[
G\ (A \lor B); \text{true} \land \neg G \cdot F \cdot A ; \text{true}
\]

\[
\Rightarrow \quad \{ \text{dual of } F \text{ and } G \} \\
G\ (A \lor B); \text{true} \land F; (G \cdot \neg (A; \text{true}))
\]

\[
\Rightarrow \quad \{ \text{persistence rule} \} \\
F; (G\ (A \lor B); \text{true} \land G \cdot \neg (A; \text{true}))
\]

\[
\Rightarrow \quad \{ \text{over } \lor \text{ and } G \text{ over } \land \} \\
F; (G \ (A; \text{true} \lor B; \text{true}) \land \neg (A; \text{true}))
\]

\[
\Rightarrow \quad \{ \text{predicate calculus} \} \\
F; G\ (\bullet p \Rightarrow (G\ \bullet p); \bullet \sim g)
\]

\[
[G\ B; \text{true} \Rightarrow G\ (\bullet p \Rightarrow (G\ \bullet p); \bullet \sim g)]
\]

(3)
The innovation with our new rule is therefore that, as opposed to the schedule replacement rule that we saw previously, we only have to have the safety property satisfied by the other actions than the one we are refining. It is in this respect a weaker property that we need to defend. We have to use it carefully, however, because we should not the mistake to confuse (0) for $p \un \sim g$.

In our example, we won’t be using this refinement law but we still suggest a notation to distinguish between the two notions. We could refer to (0) by $p \un \sim g \except A$. 
Chapter 5

Example. Mutual Exclusion

In this chapter, we illustrate the use of the Unit-B method by designing a formal specification for the mutual exclusion problem and deriving a solution.

5.0 Description

Mutual exclusion is a typical problem of concurrent programming. More precisely, it is a problem of synchronization. The problem occurs when separate processes share a variable and need independently to write to it. Unless their writing operations are very simple, the processes can interfere with each other when they try to write at once. This is why the usual solution is to separate the code of the processes into critical and non-critical sections and to have the critical sections exclude each other in time. The notion of mutual exclusion is actually a safety property. In the literature, the problem comes with various progress properties. It is common to see solutions checked for the absence of deadlock or for the absence of individual starvation.
The absence of deadlock assert that if some processes are waiting to get into their critical sections, one will eventually get in. It does not exclude starvation: a process waiting forever as other processes are repeatedly granted access to their critical section. This kind of situation can be problematic and it is excluded by the absence of individual starvation.

In our solution, we start one level of abstraction higher. We are given a specification where some actions, supposedly taken from different processes, have to write to a shared variable, $x$, within a finite delay of being requested to. At the first level of abstraction, we have no mentions of mutual exclusion or of critical sections but they will be introduced in the first refinement. Because of the properties of the abstract specification, the solution we are looking for has to exclude individual starvation but it has not been written as such in the abstract specification. It only comes as a consequence of the necessity for the jobs to be carried through eventually.

5.1 Conventions

In the formal development, we adopt the following conventions for our development.

Firstly, we use the following conventions for labelling our properties.

- Progress properties start with $P$.
- Safety properties start with $S$.
- Each property associated with two numbers, representing the level of refinement and the sequence number within the refinement (starting from 0).
For example $P_{1\_0}$ means that this is the first progress properties within the first refinement.

Secondly, we label actions in each refinement with the level of refinement. We use similar convention to label the other

Thirdly, we use before-after predicates representing the update of variable during actions, with the assumption that the frame of assignment are all variable of the models. For example, if the state has two variables $x,y$, an assignment $x' = 1$ states that $x$ is assigned 1 and $y$ can change arbitrary, where $x' = 1 \land y' = y$ states additionally that $y$ is unchanged.

Finally, during the development, we interleave the introduction of properties and actions of the program. This allows us to introduce the details of the model when it is necessary. In particular, we are going to strengthen the guard and assignment of the action during the development. During this kind of elaboration, proofs that we have done, e.g. invariant maintenance or satisfying a progress property, are maintained.

### 5.2 Formal Development

We adopt the following refinement strategy:

**Initial Model**  Progress property.

**First Refinement**  Mutual exclusion.

**Second Refinement**  Implementation using queue.
5.2.0 Initial Model

In this initial program called Mutex0, we are initially interested in a set of processes which are capable of taking care of classes of requests. For any process \( P.i \), \( p.i \) tells us that a request for updating \( x \) corresponding to \( P.i \) is pending. We are concerned with requests to be serviced in a finite time.

\[
p.i \mapsto \neg p.i \quad (P0_0)
\]

Each process \( P.i \) services a pending request by applying his own function \( f.i \) to a shared variable \( x \).

\[
p.i \land x = k \quad \text{co} \quad x = f.i.k \lor p.i \quad (S0_0)
\]

Note that both \( P0_0 \) and \( S0_0 \) represent a set of properties, i.e. universally quantified over \( i \) (the processes).

From (2.31), we know that \( P0_0 \) is equivalent to \( \text{tr} \ p.i \), which could be implemented by a single action. As a result, we are going to implement \( P0_0 \) by the following action \( A0 \). The notation \( .i \) indicates that this is multiple event, i.e. it represents several events, one for each value of \( i \).

```
A0.i
during
A0_cs1:  p.i
begin
A0_as1:  p'.i = false
end
```

According to Section 3.3.2, to ensure that \( A0.i \) helps to implement \( P0_0 \),
we have to prove the following conditions.

\[(p.i \Rightarrow p.i) \text{ is invariant} \quad (A0/P0_0/SCH)\]
\[p.i \land A0.i \Rightarrow \neg p'.i \quad (A0/P0_0/NEG)\]
\[p.i \mapsto \text{true} \quad (A0/OP)\]

These condition are trivial, in particular, for \(A0/P0_0/NEG\) holds since
the action ensures \(p.i\) is set to false.

\[A0\text{ satisfies } P0_0\]

\(A0\) also need to preserves the safety property \(S0_0\), we add the following
assignments to \(A0\).

\[A0\_as2: \quad (\forall j: i \neq j: p'.j = p.j)\]
\[A0\_as3: \quad x' = f.i.x\]

Here, we have to consider prove that \(A0.i\) satisfies \(S0_0.i\) and \(A0.i\) satisfies
\(S0_0.j\) for \(i \neq j\). According to Section 3.3.1, we have to prove the following
conditions.

\[(p.i \land x = k) \land A0.i \Rightarrow (x' = f.i.k \lor p'.i) \quad (A0.i/S0_0.i/CO)\]
\[(p.j \land x = k) \land A0.i \Rightarrow (x' = f.j.k \lor p'.j) \quad (A0.i/S0_0.j/CO)\]

The proof for \(A0.i/S0_0.i/CO\) is as follows

\[x' = f.i.k \lor p'.i\]
\[\Leftrightarrow \{ A0\_as1, A0\_as3 \}\]
\[f.i.x = f.i.k \lor \text{false}\]
\[\Leftrightarrow \{ \text{ Strengthening } \}\]
\[p.i \land x = k\]
The proof for $A_0.i/S_{0_0}.j/CO$ is as follows

$$x' = f.j.k \lor p'.j$$

$$\Leftarrow \{ A_0\_as2 \text{ for } j \neq i \}$$

$$x' = f.j.k \lor p.j$$

$$\Leftarrow \{ \text{Strengthening} \}$$

$$p.j \land x = k$$

Now, we can conclude that

$$A_0 \text{ satisfies } S_{0_0}$$

It can be noticed that an assignment to $p.i$ setting it to $true$, while leaving the other $p.j$ unchanged will satisfy safety property $S_{0_0}$. It is included because it is felt that the safest way to use the design under construction is to superpose a program on top of it and possibly weaken the guards.

```plaintext
C.i
  during
  C0\_cs1 : false
  begin
    C0\_as1 : p.i = true
    C0\_as2 : (\forall j: i \neq j: p'.j = p.j)
  end
```
We can conclude that

$$\mathcal{C}_0 \text{ satisfies } \mathcal{S}_0\{0\}$$

5.2.1 First Refinement

The initial program is abstract, in the sense that it allows read and write to shared variable $x$ atomically. In this refinement, we impose the atomicity constraints on accessing $x$, so that only atomic read and atomic write are allowed.

In order to remove $x$ from the right hand side of the assignment of $A$, we need to add a series of variables local to the various processes, namely $y.i$ and $b.i$, where $y.i$ will be given the values of $f.i.x$ when $b.i$ is true. We add an invariant that will specify when the local variables will be consistent with the global variable:

$$b.i \Rightarrow y.i = f.i.x \text{ is invariant } \quad (J1\{0\})$$

In order to rewrite the action $A0$, we need to strengthen its guard. Since we require that the schedule of an action be stronger than its guard, we are going to strengthen the coarse schedule too.
\[
A_1.i
\]

\textbf{when}\n\[A_1\text{-}gd1 : \ b.i\]
\textbf{during}\n\[A_0\text{-}cs1 : \ p.i\]
\[A_1\text{-}cs2 : \ b.i\]
\textbf{then}\n\[A_0\text{-}as1 : \ p'.i = \text{false}\]
\[A_0\text{-}as2 : \ \langle \forall j : i \neq j : p'.j = p.j \rangle\]
\[A_1\text{-}as3 : \ x' = y.i\]
\[A_1\text{-}as4 : \ \langle \forall j : y'.j = y.j \rangle\]
\textbf{end}\n
Note that \(A_1\) inherits some elements from \(A_0\) (those with the same labels).

We first consider the fact that \(A_1\) is a safety refinement of \(A_0\), i.e. the conditions \texttt{GRD} and \texttt{SIM}.

\[
(h \Rightarrow g) \ \text{is invariant} \quad \quad (A_1/\text{GRD})
\]
\[
A_1.i \Rightarrow A_0.i \quad \quad (A_1/\text{SIM})
\]

This is sufficient to ensure that the rewriting is a correct refinement for safety but it has the ability to falsify \(J_{1-0}\). We could basically take two directions from here. Either we strengthen the precondition to require that no other “\(b\)” is true or we add an invariant that allows us to deduce from \(b.i\) that all other \(b\) are false. The former is not usable because, if two \(b\) come to be true at the same time, neither of the corresponding agent can proceed. We’re better off imposing mutual exclusion of \(b.i\).

\[
\langle \forall i : b.i : \langle \forall j : i \neq j : \neg b.j \rangle \rangle \ \text{is invariant} \quad \quad (J_{1-1})
\]

It can be formulated more symmetrically as:

\[
\langle \forall i,j : i \neq j : \neg b.i \lor \neg b.j \rangle \ \text{is invariant} \quad \quad (J_{1-1})
\]
The former allows us to take advantage of $J_{1 \_1}$ in the invariance of $J_{1 \_0}$ by $A_1$ and the latter is simply more elegant.

$$A_1 \text{ satisfies } \quad J_{1 \_0}$$

Thanks to $J_{1 \_1}$, $A_1$ now preserves $J_{1 \_0}$. In order to make sure that $A_1$ preserves the newly introduced invariant $J_{1 \_1}$, we added the following assignments to make sure that it does not change any $b.j$ other than $b.i$.

$$A_1\_as5 : \ (\forall j: i \neq j: b.j = b.j)$$

With the additional assignment, we can conclude

$$A_1 \text{ satisfies } \quad J_{1 \_1}$$

We now have to take care of the proof obligations incurred by strengthening the coarse schedule of $A_1$ (Section 4.1.1). They will draw the outline of the rest of the development. They requires the properties to hold for this first refinement.

$$p.i \implies p.i \land b.i \quad (P1\_0)$$
$$p.i \land b.i \un\neg p.i \quad (S1\_0)$$

Before we take care of a new progress property, we make sure that the safety property $S1\_0$ is reasonable. For $A_1$, it results in the following condition (after translating $\un$ property into $p.i \land b.i \ \co \neg p.i \lor b.i$). Once again,
we consider the properties in different combination of events and properties.

\[(p.i \land b.i) \land b.i \land A1.i \Rightarrow (\neg p'.i \lor b'.i), \quad (A1.i/S1_0.i/CO)\]

\[(p.j \land b.j) \land b.i \land A1.i \Rightarrow (\neg p'.j \lor b'.j), \quad (A1.i/S1_0.j/CO)\]

The condition are again trivial to prove.

\[A_1 \text{ satisfies } S_1_0\]

We can proceed with the progress property. We use the ensures rule 2.29 to prove \(P1_0\), which result into proving two different conditions: a co property and a tr property.

\[p.i \rightarrow p.i \land b.i\]

\[\Leftarrow \{ 2.29 \}\]

\[(p.i \text{ un } p.i \land b.i) \land (\text{tr } p.i \land \neg(p.i \land b.i))\]

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

\[(p.i \text{ un } p.i \land b.i) \land (\text{tr } p.i \land \neg b.i)\]

\[= \{ 2.8 \}\]

\[(p.i \land \neg(p.i \land b.i) \quad \text{co } p.i \lor (p.i \land b.i)) \land (\text{tr } p.i \land \neg b.i)\]

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

\[(p.i \land \neg b.i \quad \text{co } p.i) \land (\text{tr } p.i \land \neg b.i)\]

The properties are copied here for future reference.

\[p.i \land \neg b.i \text{ co } p.i \quad (S1_1)\]

\[\text{tr } p.i \land \neg b.i \quad (P1_1)\]
The fact that \( A_1 \) satisfies \( S_{1_1} \) is captured by the following conditions (by combining properties and events)

\[
(p.i \land \neg b.i) \land b.i \land A_1.i \Rightarrow p'.i \quad (A_1.i/S_{1_1}.i/CO)
\]
\[
(p.j \land \neg b.j) \land b.i \land A_1.i \Rightarrow p'.j \quad (A_1.i/S_{1_1}.j/CO)
\]

which are trivial to discharge.

\[ A_1 \text{ satisfies } S_{1_1} \]

We can take care of \( P_{1_1} \) with the following new event and its companion safety property.

\[
\begin{array}{c}
\text{B1.i} \\
\text{during} \\
\text{B1_cs1} : \ p.i \\
\text{begin} \\
\text{B1_as1} : \ b'.i = \text{true} \\
\text{B1_as2} : \ p'.i = \text{true} \\
\text{end}
\end{array}
\]

The proof that \( B_1 \) implements \( P_{1_1} \) requires to prove the conditions.

\[
(p.i \land \neg b.i \Rightarrow p.i) \text{ is invariant} \quad (B_1/P_{1_1}/SCH)
\]
\[
(p.i \land \neg b.i) \land p.i \land B_1 \Rightarrow -(p'.i \land \neg b'.i) \quad (B_1/P_{1_1}/NEG)
\]
\[
p.i \Rightarrow \text{true} \quad (B_1/OP)
\]
There conditions are trivial to prove in our context.

\[ B_1 \text{ satisfies } P_{1_1} \]

We can see that \( B \) risks destroying the invariance of \( J_{1_0} \) so we augment it with the following assignment to \( y_s \):

\[ B_1_{\text{as}3} : \ y_i' = f.i \cdot x \]
\[ B_1_{\text{as}4} : \langle \forall j: i \neq j : y_j' = y_j \rangle \]

Similarly, in order to satisfy \( S_{1_0} \), we need the following additional action to ensure that no \( b_j \) other than \( b_i \) are changed.

\[ B_1_{\text{as}5} : \langle \forall j: i \neq j : b_j' = b_j \rangle \]

Finally, \( S_{1_1} \) is satisfied by \( B_1 \) trivially.

\[ B_1 \text{ satisfies } J_{1_0}, S_{1_0}, S_{1_1} \]

However, \( J_{1_1} \) can be falsified by \( B \) if \( b_j \) is true for \( i \neq j \). We can, and do, strengthen the guard with the necessity for all other \( b_j \) for \( i \neq j \) to be false.

\[ B_1_{\text{gd}1} : \langle \forall j: i \neq j : \neg b_j \rangle \]

It immediately satisfies \( J_{1_1} \) but we are left with another problem on our hands: the guard is now stronger than the schedule, which constitutes an inconsistency.

\[ B_1 \text{ satisfies } J_{1_1} \]
The usual solution, strengthening the coarse schedule, e.g. adding \( \forall j: i \neq j: \neg b.j \) to the coarse schedule, is of no help. An attempt to prove that the change is sensible, we apply the rules for changing coarse schedule. This results in proving two conditions.

\[
\begin{align*}
& p.i \rightarrow (\forall j: i \neq j: \neg b.j) \land p.i \\
& (\forall j: i \neq j: \neg b.j) \land p.i \textbf{ un } \neg p.i
\end{align*}
\]

The transformation for the \textbf{ un } condition into \textbf{ co } is as follows.

\[
(\forall j: i \neq j: \neg b.j) \land p.i \textbf{ un } \neg p.i
= \quad \{ \text{ definition 2.8 } \}
(\forall j: i \neq j: \neg b.j) \land p.i \textbf{ co } (\forall j: i \neq j: \neg b.j) \lor \neg p.i
\]

This \textbf{ co } condition would not be possible to meet by B1. If two requests were to be issued one after the other, while no b was true, none of the b could ever become true. It would basically preclude any of the other processes to be admitted in their critical section if p.i holds. As a consequence, p holding for two different i’s would result in a deadlock.

With general scheduling, we are rather lucky because we can strengthen the guard and get only the good half of the proof obligation. This almost strengthening is more precisely the introduction of a schedule which differs from the guard. For brevity’s sake, since the schedule and the precondition are the same, we will mention only one. We choose to express only the schedule.
We now have to prove the following condition

\[ p.i \mapsto (\forall j: i \neq j: \neg b.j) \, . \, \quad (P1_2) \]

This corresponding to the well-definedness of the action B1, (condition a_i/OP), and also allows us to prove that the additional of the fine schedule does no harm as in Section 4.1.2.

We will transform it to make it easier to implement.

\[ p.i \mapsto (\forall j: i \neq j: \neg b.j) \]
\[ \Leftarrow \quad \{ \text{(Anti)monotonicity for symmetry} \} \]
\[ \text{true} \mapsto (\forall j:: \neg b.j) \]
\[ = \quad \{ \text{U0} \} \]
\[ \neg (\forall j:: \neg b.j) \mapsto (\forall j:: \neg b.j) \]
\[ = \quad \{ \text{predicate calculus} \} \]
\[ \langle \exists j :: b.j \rangle \mapsto \langle \forall j :: \neg b.j \rangle \]
\[ = \quad \{ \text{Disjunction} \} \]
\[ \langle \forall i :: b.i \mapsto \langle \forall j :: \neg b.j \rangle \rangle \]

As a result, we add the following progress property into our model.

\[ b.i \mapsto \langle \forall j :: \neg b.j \rangle \] \hspace{1cm} (P1_3)

\[ \textbf{P1}_3 \quad \text{imply} \]
\[ \textbf{P1}_2 \]

We use the ensures rule (2.29) to prove \textbf{P1}_3, which result into proving two different conditions: a co property and a tr property.

\[ b.i \leftrightarrow \langle \forall j :: \neg b.j \rangle \]
\[ \leftarrow \quad \{ \text{2.29} \} \]
\[ (b.i \ \textbf{un} \ \langle \forall j :: \neg b.j \rangle) \land (\textbf{tr} \ b.i \land \neg \langle \forall j :: \neg b.j \rangle) \]
\[ = \quad \{ \text{predicate calculus} \} \]
\[ (b.i \ \textbf{un} \ \langle \forall j :: \neg b.j \rangle) \land (\textbf{tr} \ b.i) \]

The two properties are listed below as follows.

\[ b.i \ \textbf{un} \ \langle \forall j :: \neg b.j \rangle \] \hspace{1cm} (S1_2)
\[ \textbf{tr} \ b.i \] \hspace{1cm} (P1_4)

\[ \textbf{S1}_2, \textbf{P1}_4 \quad \text{imply} \]
\[ \textbf{P1}_3 \]

To implement \textbf{P1}_4, we can use \textbf{A1}.i which does not assign to \textbf{b}.i yet by adding the following assignment.
A1_as6: \( b'.i = \text{false} \)

We have three conditions to prove.

\[
\begin{align*}
(b.i \Rightarrow (p.i \land b.i)) & \quad \text{is invariant} \quad (A1.i/P1_3/SCH) \\
b.i \land (p.i \land b.i) \land A1.i & \Rightarrow \neg b'.i \quad (A1.i/P1_3/NEG) \\
(p.i \land b.i \Rightarrow \text{true}) & \quad (A1.i/OP)
\end{align*}
\]

Condition A1.i/P1_3/NEG is trivial because of the assignment A1_as6. Condition A1.i/OP is also trivial to prove. For A1.i/P1_3/SCH, we would need for b.i be stronger than the coarse schedule of A1. For that purpose, we can introduce J1_2.

\[
b.i \Rightarrow p.i \quad \text{is invariant} \quad (J1_2)
\]

A1 and B1 already maintain J1_2. S1_2 is satisfied by A1 with the support from J1_1. With respect to B1 and S1_2, B1 satisfies a stronger property b.i un false.

\[
\begin{align*}
\text{A1} & \quad \text{satisfies} \\
P1_4, J1_2, S1_2
\end{align*}
\]

\[
\begin{align*}
\text{B1} & \quad \text{satisfies} \\
J1_2, S1_2
\end{align*}
\]

Finally, we have to prove that C0 maintains all safety properties introduced in this refinement. This is ensured by adding the following assignments to make sure that C0 does not change any bs or ys.
This concludes the first refinement.

5.2.2 Second Refinement

Using techniques that we have already seen, we separate the actions which deal with synchronization from those which deal with the shared variable. It would be convenient for the remainder of the developments if we had a proper decomposition mechanism but it falls outside the scope of this thesis. Instead, we assume that the actions dealing with the shared variables are still around and we implicitly modify them as we go so that they preserve the value of the variables we will introduce. This will effectively make them satisfy both the new safety and the new progress properties. In the remainder of the development, we will ignore the actions that are not relevant.

We continue our development with the following three stripped-down actions:

```plaintext
A_1.i
when
A0_gd1 : b.i
during
A0_cs1 : p.i
A1_cs2 : b.i
then
A0_as1 : p'.i = false
A0_as2 : (∀ j: i ≠ j: p'.j = p.j)
A1_as4 : b'.i = false
A1_as5 : (∀ j: i ≠ j: b'.j = b.j)
end
```
We want to subsume remove \( B_1_{_{\text{fs2}}} \) by assimilating it to the coarse schedule of \( B \). Since we know that the fine schedule will not be part of a stable condition, we find a stronger predicate, one that we can put as the left argument of an unless property and one which we can establish inductively:

\[
  v.i = 0 \implies \langle \forall j: i \neq j: \neg b.j \rangle \quad \text{is invariant} \quad \text{(J2\_0)}
\]

With the above invariant, we can strengthen the coarse schedule of \( B \) in such a way as to imply \( B_1_{_{\text{fs2}}} \). We simply have to add to the coarse schedule \( v.i = 0 \).

\[
  B_2_{_{\text{cs2}}} : \quad v.i = 0
\]
which gives us the task to prove

- Strengthening of B’s coarse schedule:

\[
\begin{align*}
  p_i &\implies p_i \land v_i = 0 \\
  p_i \land v_i = 0 &\implies \neg p_i
\end{align*}
\]  

(P2_0)  
(S2_0)

We strengthen P2_0 with the induction rule to obtain a stronger progress property and an invariant, to ensure that the relation < over v.i be well-founded.

\[
\begin{align*}
  p_i \land v.i = k &\implies (p_i \land v.i < k) \lor (p_i \land v.i = 0) \\
  p_i &\implies 0 \leq v.i \text{ is invariant}
\end{align*}
\]  

(P2_1)  
(J2_1)

P2_1, J2_1 imply P2_0

If we directly apply our ensures-rule (2.32) to (P2_1), we obtain as our transient predicate:

\[
\text{tr } p_i \land v.i = k \land v.i \neq 0
\]  

(P2_2)

The above suggest that we use an action which decreases v.i. Since decreasing v.i can nullify it, we are forced to choose an action that establishes the consequent of (J2_0). In the previous refinement, it has been proved that A establishes

\[\langle \forall i:: \neg b.i\rangle\]

in one step. This suggest to us the choice of A for decreasing v.i while
preserving $J_{2\_0}$.

A satisfies $J_{2\_0}$

At this point, we have a problem: the predicate in $P_{2\_2}$ is not stronger than the coarse schedule of A. Strengthening the predicate of $P_{2\_2}$ would do the job but we can’t do it directly because $tr$ is antimonotonic. We are forced to revise our treatment of $P_{2\_1}$. The presence of $b.j$ in $P_{2\_2}$, for any $j$, would have solved our problem. We use transitivity to add it to an intermediate predicate between the two arguments of $P_{2\_1}$.

$$p.i \land v.i = k \iff (\exists j:: b.j) \land v.i = k \land p.i \quad (P_{2\_3})$$

$$\langle \exists j:: b.j \rangle \land v.i = k \land p.i \iff (p.i \land v.i < k) \lor (p.i \land v.i = 0)$$

$$\langle \exists j:: b.j \rangle \land v.i = k \land p.i \iff (p.i \land v.i < k) \lor (p.i \land v.i = 0)$$

= \{ disjunction \}

$$\langle \forall j:: b.j \land v.i = k \land p.i \iff (p.i \land v.i < k) \lor (p.i \land v.i = 0) \rangle$$

$\iff \{ trading \}$

$$\langle \forall j:: b.j \land v.i = k \land p.i \land v.i \neq 0 \iff p.i \land v.i < k \rangle$$

$$P_{4}: b.j \land v.i = k \land p.i \land v.i \neq 0 \iff p.i \land v.i < k \quad (P_{2\_4})$$

$P_{2\_3}, P_{2\_4}$ imply $P_{2\_1}$

To $P_{4}$, we can apply the ensures rule and obtain:

$$tr \quad b.j \land v.i = k \land p.i \land \neg v.i = 0 \quad (P_{2\_5})$$
\( b.j \land v.i = k \land p.i \land \neg v.i = 0 \) \( \text{un} \) \( p.i \land v.i < k \) \hspace{1cm} (S2_1)

**P2_5, S2_1** imply **P2_4**

Together with \( J1_2.j \), from the previous refinement, the predicate of \( P2_5.j \) is stronger than the coarse schedule of \( A.j \). Additionally, we can augment it with a statement decreasing all the \( v.i \), thereby taking care of both **P2_5** and **S2_1**.

\[ A2 \text{ as6} : \ [v' = v - 1] \]

\[ A2 \text{ satisfies } \quad P2_5, S2_1 \]

Since \( B \) does not change the value of \( p \) and does not falsify any \( b \), it is reasonable to augment it so as to preserve the truth of the left-hand side of **S2_1** by preserving the value of \( v \).

\[ B2 \text{ as6} : \ [v' = v] \]

\[ B2 \text{ satisfies } \quad S2_1 \]

If we turn to **P2_3**, we can expect that a single application of the ensures-rule allows us to implement it.

\[ \text{tr} \ p.i \land v.i = k \land \langle \forall j:: \neg b.j \rangle \] \hspace{1cm} (P2_6)

\[ p.i \land v.i = k \text{ un} \langle \exists j:: b.j \rangle \land v.i = k \land p.i \] \hspace{1cm} (S2_2)

\[ p.i \land v.i = k \land \langle \forall j:: \neg b.j \rangle \text{ co} \ p.i \land v.i = k \]

108
P2_6, S2_2 imply P2_3

Our strategy to implement P6 is to assign true to one b.j and B qualifies perfectly for this. We then have to make sure that the predicate of P6 is stronger than the coarse guard of B. We don’t have to compare it to B’s fine guard because the proper relation between the schedules of B have been established in the first refinement.

\[
\langle \exists j:: p.i \Rightarrow v.j = 0 \land p.j \rangle = \{ \text{J2_2 to remove } p.j \} \langle \exists j:: p.i \Rightarrow v.j = 0 \rangle = \{ \Rightarrow \text{ over } \exists \} p.i \Rightarrow \langle \exists j:: v.j = 0 \rangle \\
\Leftarrow \begin{cases}
\text{antimonotonicity, we take ‘tl’ as the number of waiting processes and use } (p.i \Rightarrow 0 < tl) \\
to \text{state that the queue is non empty}
\end{cases}
\]
\[0 < tl \Rightarrow \langle \exists j:: v.j = 0 \rangle \]
\[\Leftarrow \{ \text{instantiation with } n := 0 \} \langle \forall n: 0 \leq n < tl: \langle \exists j:: v.j = n \rangle \rangle \]

\[0 \leq v.i \Rightarrow p.i \text{ is invariant (J2_2)} \]
\[\langle \forall n: 0 \leq n < tl: \langle \exists j:: v.j = n \rangle \rangle \text{ is invariant (J2_3)} \]

B2 satisfies P2_6

109
We can easily see that $B_2$ also satisfies $S_{2\_2}$.

$B_2$ satisfies $S_{2\_2}$

$S_3$ is also satisfied by $A$ because the left-hand side of its co formulation contradicts the guard of $A$.

The progress properties being addressed, we focus on the remaining safety properties. We will do so one action at a time.

$A.i$ satisfies $J_{2\_1}.i$ because it falsifies $p.i$. However, we need to verify the interaction between $A.i$ and $J_{2\_1}.j$ with $i \neq j$:

\[
\begin{align*}
p.j' & \Rightarrow 0 \leq v.j' \\
= & \quad \{ A2.i \text{ with } j \neq i \} \\
p.j & \Rightarrow 0 < v.j \\
= & \quad \{ \langle \rangle \} \\
p.j & \Rightarrow 0 \leq v.j \land 0 \neq v.j \\
= & \quad \{ J_{2\_1}.j \} \\
p.j & \Rightarrow 0 \neq v.j \\
\leftarrow & \quad \{ A0._gd1 \text{ and } J_{2\_4}; \text{strengthening} \} \\
v.i \neq v.j
\end{align*}
\]

\[
\begin{align*}
b.i & \Rightarrow v.i = 0 \quad \text{is invariant} \quad \text{(J2_4)} \\
(\forall i, j: i \neq j: v.i \neq v.j) & \quad \text{is invariant} \quad \text{(J2_5)}
\end{align*}
\]

$A_2$ satisfies $J_{2\_1}$

$J_{2\_1}$
A preserves J2 trivially because it terminates with b being false everywhere. Since A decreases v everywhere, J3 is preserved. Now, the easiest way to preserve J3 would be to decrease all the v by one. As for J6.j, with \( i \neq j \), A.i maintains it because p.j stays unchanged and the predicate \( 0 \leq v.j \) is strengthened with the execution of A.i.

\[ \text{A2 satisfies} \quad J2_4, J2_5, J2_2 \]

\[ \langle \exists j :: v'.j = n \rangle \]
\[ = \quad \{ \text{A2} \} \]
\[ \langle \exists j :: v.j = n + 1 \rangle \]
\[ \Leftarrow \quad \{ J2_3 \text{ with } n := n + 1 \} \]
\[ 0 \leq n + 1 < tl \]
\[ = \quad \{ \text{algebra} \} \]
\[ -1 \leq n < tl - 1 \]
\[ \Leftarrow \quad \{ \text{A2\_as7 : } tl' = tl - 1 \} \]
\[ 0 \leq n < tl' \]

\[ \text{A2\_as7} : \quad tl' = tl - 1 \]

\[ \text{A2 satisfies} \quad J2_3 \]

All that is left to verify for A is S2_0. It can be reformulated as:

\[ p.i \land v.i = 0 \quad \text{co} \quad v.i = 0 \lor \neg p.i \]

We focus on A.j and start with a case analysis around \( i = j \). If we execute A.j, with \( i \neq j \), we know that v.i = 0 does not hold because of J2_4, J2_5.
and \( A_0_{gd1} \) — in short, \( v.j = 0 \) holds and all values of \( v \) are distinct. Hence, \( A.j \) satisfies \( S2\_0.i \) provided \( i \neq j \). In the case of \( i = j \), \( p.i \) is falsified and \( S2\_0.i \) satisfied too.

\[
A2 \quad \text{satisfies} \\
S2\_0
\]

We now turn to \( B \) to see if it satisfies all the safety properties. Let’s start with the invariants.

Since \( B.j \) sets \( b.j \) to true, it can falsify a \( J2\_0.i \) for \( i \neq j \). To avoid it, we guard \( B.j \) with \( \langle \forall i: i \neq j: \neg v.i = 0 \rangle \) which, thanks to \( J2\_5 \), follows from \( v.j = 0 \). This condition is already part of the coarse schedule. As a consequence, we know we are not introducing an inconsistency. In the case where \( i = j \), \( B2\_as6 \) preserves the value of the antecedent of \( J2\_0.i \) and \( B2\_as6 \) preserves the value of its consequence. Hence, \( B2 \) preserves \( J2\_0 \).

\[
B2_{gd2}: \quad v.i = 0
\]

\[
B2 \quad \text{satisfies} \\
J2\_0
\]

It turns out that this new guard also helps maintain \( J2\_4 \). Together with \( J2\_0.i \), the new guard falsifies the antecedent of \( J2\_4.j \), for \( i \neq j \). In the case of \( i = j \), since the value of \( v \) is everywhere preserved, \( v.i = 0 \land b.i \) hold after the execution of \( B.i \).

\[
B2 \quad \text{satisfies} \\
J2\_4
\]

As for \( J2\_1 \) and \( J2\_5 \), the variables that they contain are left unchanged.
by B2. It is also the case of J2_2 and S2_0.

B2 satisfies

\[ J2_1, J2_5, J2_2, S2_0 \]

In the case of J2_3, it contains one new variable, tl, that is so far unconstrained by B2. Since B2 preserves the value of v, the other variable appearing in J2_3, it is reasonable for B to preserve the value of tl also.

\[ B2_{as7} : \quad tl' = tl \]

B2 satisfies

\[ J2_3 \]

It is finally the turn of C. All invariants but J2_1 would be well served if C preserved the value of v. However, because of C0_{as1.i}, J2_1.i requires us to change v.i to make it non-negative. In addition, J2_5 requires us to choose a value which is distinct from that of any other image of v. We could choose tl for the new value of v.i since we have already started giving it the role of the tail of a queue of processes. Our invariants so far are still insufficient to guarantee that assigning tl to v.i will guarantee unicity. We will add two invariants to make of tl a proper choice.

\[ 0 \leq tl \quad \text{is invariant} \quad (J2_6) \]
\[ v.i < tl \quad \text{is invariant} \quad (J2_7) \]

In passing, we note that A2 and B2 satisfy the newly introduced invariants.

A2 satisfies

\[ J2_6, J2_7 \]
B2 satisfies $\mathbf{J}_{2\,6}, \mathbf{J}_{2\,7}$

We next adapt the specification of $\mathbf{C}_2$ to meet the newly introduced invariants and the others.

$\mathbf{C}_2$ \_as4\,: $v'.i = tl$

$\mathbf{C}_2$ \_as5\,: $(\forall j: i \neq j: v'.j = v.j)$

$\mathbf{C}_2$ satisfies $\mathbf{J}_{2\,1}, \mathbf{J}_{2\,2}$

$\mathbf{C}_2$ \_as6\,: $tl' = tl + 1$

$\mathbf{C}_2$ satisfies $\mathbf{J}_{2\,6}, \mathbf{J}_{2\,7}$

We present only a few proofs of invariance in detail. The simpler ones will be left out.

- Invariance of $\mathbf{J}_{2\,5}$:

  $$(\forall j: i \neq j: v'.j \neq v'.i)$$

  $= \{ \mathbf{C}_2$ \_as4 \} $$

  $$(\forall j: i \neq j: v.j \neq tl)$$

  $\Leftarrow \{ \text{strengthening} \}$$

  $$(\forall j: i \neq j: v.j < tl)$$

  $\Leftarrow \{ \text{range weakening} \}$$

  $$(\forall j: v.j < tl)$$

114
\[ \{ \text{J2}_6 \} \]

true

C2 satisfies
\[ \text{J2}_7 \]

- Invariance of \( \text{J2}_0 \):

\[ \langle \forall j: i \neq j: \neg b'.j \rangle \]
\[ = \{ C1 \} \]
\[ \langle \forall j: i \neq j: \neg b.j \rangle \]
\[ \Leftarrow \{ \text{J2}_4 \} \]
\[ \langle \forall j: i \neq j: v.j \neq 0 \rangle \]
\[ = \{ v.i = 0 \} \]
\[ \langle \forall j: i \neq j: v.j \neq v.i \rangle \]
\[ = \{ \text{J2}_5 \} \]

true

C2 satisfies
\[ \text{J2}_0 \]

- Invariance of \( \text{J2}_4 \):

\[ v'.i = 0 \Leftarrow b'.i \]
\[ = \{ C1 \} \]
\[ tl = 0 \Leftarrow b.i \]
\[ = \{ \text{C2}_gd1: \neg b.i \} \]
true

$C2_{gd1} : \neg b.i$

$C2$ satisfies

$J2_4$

• Invariance of $J7$:

$0 \leq n < tl' \Rightarrow \langle \exists j :: v'.j = n \rangle$

$= \{ \text{split range} \}$

$0 \leq n < tl' \Rightarrow \langle \exists j : i \neq j : v'.j = n \rangle \lor v'.i = n$

$= \{ C2 \}$

$0 \leq n < tl + 1 \Rightarrow \langle \exists j : i \neq j : v.j = n \rangle \lor tl = n$

$= \{ \text{shunting} \}$

$0 \leq n < tl + 1 \land tl \neq n \Rightarrow \langle \exists j : i \neq j : v.j = n \rangle$

Since we don't need to use the whole formula that we obtained, we simply start calculating with one side, to transform it into the other side.

$0 \leq n < tl + 1 \land tl \neq n$

$= \{ \text{algebra} \}$

$0 \leq n < tl$

$\Rightarrow \{ J2_3 \}$

$\langle \exists j :: v.j = n \rangle$

$= \{ \text{split range} \}$

$\langle \exists j : i \neq j : v.j = n \rangle \lor v.i = n$

We are almost at the shape we were aiming for. All we need is to remove the last disjunct of the above formula. Equating it to the
identity of $\lor$, false, would solve the problem.

\[
v.i = n \\
\Rightarrow \quad \{ 0 \leq n \} \\
0 \leq v.i \\
\Rightarrow \quad \{ J2_2 \} \\
p.i \\
= \quad \{ \neg p.i \} \\
\text{false}
\]

(end of proofs of invariance)

We see that in order to preserve $J2_3$, we have needed $\neg p.i$ which we can include as a guard of $C2$.

\[
C2_{gd2} : \neg p.i
\]

$C2$ satisfies $J2_3$

And it so happens that it also takes care of $S2_0$, $S2_1$ and $S2_2$.

$C2$ satisfies $S2_0$, $S2_1$, $S2_2$

The addition of the two new guards make sense: processes don’t need to be added to the queue of processes when they are already in the queue. However, even without a model of modular composition, we can see that strengthening the guard of $C2$, which was to serve as a method for using the service of the present program, we limit its availability and violate the principle of substitution. It can, however, be repaired. So far, the guards
were not needed but, in the situations where they didn’t hold, C2 would not have an effect. Since skip preserves any invariant and, in this case, that it refines C2.i provided \( \neg \text{p.i} \), we add a guarded alternative to C.

\[
\begin{align*}
\text{D}_1.i \\
& \text{when} \\
& \text{D2}_{\text{gd1}} : \text{p.i} \\
& \text{then} \\
& \text{D2}_{\text{as1}} : \text{skip} \\
& \text{end}
\end{align*}
\]

At this point, the conclusion is only tentative but it would make sense to take care that any action that can be called as a procedure don’t get its guard strengthened. Doing so would not violate the feasibility of the immediate program but might violate that of a superposed program, if it is assigned a conflicting schedule.

As a final note, we stress that we haven’t introduced a queue per se in this refinement but such a data refinement would be routine work in the context of Event-B and there is no reasons it wouldn’t be just as easy in Unit-B. It would suffice to take the following glueing invariant:

\[
\langle \forall i: \text{p.i}: \text{q.v.i} = i \rangle \tag{J2_8}
\]

with \( \text{q} \) a sequence of processes.

Showing such a refinement would not serve the purpose of demonstrating the novelty of Unit-B and we leave it out.
Chapter 6

Conclusions

**Summary**  By using computations calculus, we have managed to disentangle the notions of temporal properties and program executions and allowed us to reason about them in isolation. This led to the introduction of a notion of specification to which the notion of program is only a special case. The generalization is what enabled the use of calculation for designing specifications and for their refinement. This, we believe, reduces considerably the amount of guesswork needed in the design of programs.

We believe that, instead of adding to the development burden, the requirement for progress has provided yet another heuristic to lead us to proper solution. At the moment, this is somewhat speculative, but we have a strongly belief that the use of progress properties in the process of design would have been much harder, probably impractically so, if we had not separated the concern of proper coordination of the actions from that of the actions’ contributions to progress. Indeed, the coordination of actions seem to require heavy machinery in Event-B and, in our design, all the related details would have been unhelpful.

Obviously, all a priori work on progress in a design would be going to waste
if the notion of refinement used did not guarantee that all progress properties were preserved. Some proposals \[?, 23\] aim at preserving targeted progress properties. We do not believe this approach to be suitable because it makes the complexity of a proof of refinement proportional to the number of progress properties. In a large system, it is to be expected that numerous progress properties, be them high level or low level ones, would have to be used. Additionally, if we decompose a system, it is desirable for the refinement of a module that it does not violate the properties of the potential modules it could be composed with. This denies a priori consideration of the properties of said modules beyond the fact that they are compatible.

**Future work** For the duration of the project, we have kept the concern for modularity in mind. Although we have managed to achieve some measure of modularity on the small scale of one algorithm, it is not sufficient for putting large programs together. This is therefore the next goal in the development of our theory; we would like a notion of module which makes composition monotonic with respect to refinement.

It would be very helpful also for pointing out shortcomings in the application of the method to try and apply it to various examples. We think mostly of examples of parallel algorithms and synchronization protocols but the design of discrete control systems would be very enlightening too. It would be interesting to compare the simplicity of solutions designed in Unit-B with that of other methods. Candidates would be Event-B, UNITY, TLA and Action Systems.

Finally, a note on automatic tools. It has been largely ignored in this thesis because our purpose is to provide an intellectual discipline for the design of correct programs. Although there are various ways that computer programs could assist in the application of the present method but it is a vast subject to investigate and it is largely beyond the scope of this thesis. Additionally, there are some ideas about the role we should give to tools in the application
of the Unit-B method but the development of tools is expected to be very time consuming and to yield little insights into the theory and the method. For the moment, we prefer to defer their development to a time where we have a clearer idea of the benefits to expect.

**Concluding remark**  When we set out on this project, we had the hunch that a better treatment for progress could be done in the context of Event-B. For one thing, in Event-B, refinement does not preserves progress properties and, as a consequence, the concern for progress is postponed until the last refinement of a system. This deprives the rest of the development from any input coming from the requirement of progress and it is compensated by a vague intuition of what would help progress supplemented by some incomplete proof obligations for convergence.

This lack of feedback made it very hard in the design of Event-B models to let the symbols do the work, to calculate the models, so to speak. The verification of refinement is the best that can be hoped for under such circumstances.

The goal was therefore to find a way to derive programs from temporal properties. UNITY gave us part of the answer and, armed with Rutger M. Dijkstra’s computation calculus, we managed to put together a powerful notion of refinement which we used to derive a mutual exclusion algorithm.
Appendix A

Some Predicate Calculus Techniques

Indirect Equality

\[ x \equiv y \equiv (\forall z :: (z \Rightarrow x) \equiv (z \Rightarrow y)) \]

\[ x \equiv y \equiv (\forall z :: (x \Rightarrow z) \equiv (y \Rightarrow z)) \]

Indirect Inequality

\[ x \Rightarrow y \equiv (\forall z :: (z \Rightarrow x) \Rightarrow (z \Rightarrow y)) \]

\[ x \Rightarrow y \equiv (\forall z :: (x \Rightarrow z) \Leftarrow (y \Rightarrow z)) \]

Notice that, in the quantification, in the free variable appearing in consequent of the main implication appears on the same side of its implication
Appendix B

Fixpoint Calculus

Fix-point induction

\[
\begin{align*}
[ f.x \Rightarrow x ] & \Rightarrow [ \langle \mu x :: f.x \rangle \Rightarrow x ] \\
[ x \Rightarrow f.x ] & \Rightarrow [ x \Rightarrow \langle \nu x :: f.x \rangle ] \quad \text{(fix-point induction)}
\end{align*}
\]

Fix-point rolling

\[
\begin{align*}
f.\langle \mu x :: g.(f.x) \rangle & = \langle \mu x :: f.(g.x) \rangle \\
f.\langle \nu x :: g.(f.x) \rangle & = \langle \nu x :: f.(g.x) \rangle \quad \text{(fix-point rolling)}
\end{align*}
\]

Fix-point (un-) folding

\[
\begin{align*}
\langle \mu x :: f.x \rangle & = f.\langle \mu x :: f.x \rangle \\
\langle \nu x :: f.x \rangle & = f.\langle \nu x :: f.x \rangle \quad \text{(fix-point (un-)folding)}
\end{align*}
\]
Appendix C

Iteration

Iteration rolling

\[ *(s; t); s = s; *(t; s) \quad \text{ (iteration rolling)} \]
\[ \infty s; t = s\infty t; s \]

Proof. We show only the proof for finite iterator. The proof of the infinite iterator follows the same principle and is simpler.

\[ *(s; t); s \]
\[ = \{ \text{tail recursion} \} \]
\[ \langle \mu x :: s; t; x \lor s \rangle \]
\[ = \{ ; \over \lor \} \]
\[ \langle \mu x :: s; (t; x \lor 1) \rangle \]
\[ = \begin{cases} \text{fixpoint rolling with } f.x = s; x \\ \text{and } g.x = t; x \lor 1 \end{cases} \]
\[ s; (\mu x:: t; s; x \lor 1) \]
\[ = \{ \text{definition of } * \_ \} \]
\[ s; ^* (t; s) \]
Bibliography


