Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

CML Definition 1

Deliverable Number: D23.2

Version: 1.0

Date: 4 September 2012

Public Document

http://www.compass-research.eu

Contributors:

Victor Bandur, York
Jeremy Bryans, Newcastle
Ana Cavalcanti, York
Andy Galloway, York
Jim Woodcock, York

Editors:

Jeremy Bryans, Newcastle
Andy Galloway, York
Jim Woodcock, York

Reviewers:

Joey Coleman, Aarhus
Simon Perry, Atego
Marcel Oliveira, UFPE

Document History

Ver	Date	Author	Description
0.1	15-08-2012	Jim Woodcock	Initial document version
0.2	31-08-2012	Jim Woodcock	First proper draft
0.3	04-09-2012	Jim Woodcock	First draft for comments
0.4	07-09-2012	Jim Woodcock	Draft including comments from Jeremy Bryans and Victor Bandur
1.0	28-09-2012	Jeremy Bryans	Version including comments from internal reviewers

Abstract

This report contains the first version of the semantics of the COMPASS Modelling Language (*CML*) in Hoare & He's Unifying Theories of Programming (UTP). This language has been constructed as a modelling language for systems of systems. An introduction to the syntax was given in D23.1 [34], and this document is meant to be read in conjunction with that, and the reader is assumed to have read that document. The language described in that document is the one considered here. This draft is intended for discussion and will be updated as part of the next deliverable (D23.3). We start with a summary of the relevant theories from UTP: the alphabetised relational calculus and the theory of designs. Next, we give a semantics to a restricted subset of *CML*. This is based on Lowe & Ouaknine's timed testing traces semantics for CSP. Following this, we extend the language subset to include stateful reactive processes and give it a timed reactive design semantics. This is novel work. Our semantics covers the core *CML* language, so the correspondence between additional language features are treated either as a shallow embedding of *CML* expression language in UTP, or as derived operators. We then describe a unifying theory for monotonic partial logics with undefined constructs to give a sound basis for choosing a treatment of undefinedness in *CML*. This theory also justifies using different verification tools for proving facts about *CML* specifications. We give the first version of the operational semantics for the kernel language. Finally, we give an overview of the semantics for the object-oriented features of *CML*, which will be treated formally in the next version of the language.

Contents

Chapter 1

Preface

This document is COMPASS Deliverable Number D23.2, and produced as output to Task 2.3.1 within Work Package 23 [7]. The objective of Task 2.3.1 is to produce a complete definition of the *CML* language. The complete definition will be a sound notation for SoS modelling and reasoning that will integrate existing notations and semantic foundations to cover contracts, concurrency, communication, object-orientation, time, and mobility. This document contains a behavioural semantic definition of the *CML* kernel, as well as a discussion of derived operators. It also discusses some of the issues arising around the question of undefinedness, and gives a (tentative) operational semantics and an initial treatment of a copy semantics for the object-oriented features of *CML*. Subsequent outputs from this task will update and expand this deliverable.

In this deliverable, *CML0* refers to the syntax definition for *CML*, found in [34]. *CML1* refers to the semantic definitions found in this deliverable. The *kernel* refers to that restricted subset of *CML* which only contains language constructs that cannot be defined in terms of simpler constructs, and is discussed in Chapter 4. The *core* language further excludes imperative features – this subset is considered separately in order to explore the correspondence between *CML* and one of the established denotational models of CSP in the literature [19].

Inputs to this task include the work within T1.1.2 *Requirements for Methods and Tools* on the common requirements base. the work within T2.1.1 on *Guidelines for Requirements Specification for SoS* and work within T2.1.2 on *Guidelines for System Architectures for SoS*. This task will output to tasks within Theme Three on tools work. Feedback from these tasks will be taken into account in subsequent deliverables.

The current document is intended to be read in conjunction with COMPASS Deliverable Number D23.1 "*CML* Definition 0" which contains the complete initial syntactic design for CML [34]. Deliverable D23.1 will be updated and re-issued once feedback has been gathered from the COMPASS work on tools underway in Theme 3.

Semantic approach The semantic approach taken is that set out by Hoare & He in their book Unifying Theories of Programming [14]. They set out there a long-term research agenda, which has as its goal a comprehensive treatment of the relationships between all programming theories and pragmatic programming paradigms.

Review of Progress Deliverable 23.1 contains the complete syntax and for *CML*, referred to as *CML0*.

This deliverable contains a kernel semantics for *CML*. It contains the contract language, explicit functionality, concurrency and communication. We have also added a real-time semantics for *CML*. To meet the requirements of other parts of the project, we have also developed an operational semantics, although the formal proof of equivalence with the denotational semantics remains to be completed. The next deliverable from this task (D23.3) will contain a revision of D23.2 denotational and operational semantics, together with further proofs of properties and consistency of the semantics. We will also include the copy semantics for object orientation and a Hoare logic for the semantics of *CML*.

Chapter 2

Introduction

CML is the *COMPASS Modelling Language*, the first language specifically designed for modelling and analysing systems of systems. It is based on the following baseline languages: VDM [20], CSP [25], and *Circus* [15]. The first version of the language, dubbed *CML0*, contains only the syntactic description of the language [34]. The main objective of work package WP 23 of the COMPASS project is to provide a complete design for *CML*, including integration of the baseline notation's syntax and semantics. This will be used as the basis for the development of analysis techniques in Theme 2 and prototype tools development in Theme 3.

We describe the initial report on *CML* as a version for discussion [34]: this second version of the language is dubbed *CML1*. This current document (D23.2) is intended as a living document that will be updated and extended in the future Deliverables of Work Package 23.

Our chosen formalism for this work is Hoare & He's Unifying Theories of Programming (UTP) [14]. In Chapter 3, we give a detailed introduction to UTP, which we have chosen as a semantic technique for its systematic notation, methods, and emerging tools. We describe two UTP theories. In Section 3.3, we describe UTP's fundamental theory: the alphabetised relational calculus. In Section 3.5, we describe the theory of designs that underpins the use of preconditions and postconditions in VDM and the refinement calculus. These two theories form the foundations of our approach to *CML*'s semantics.

We describe the denotational semantics for a kernel of *CML* in Chapter 4. This restricted subset excludes language components that can be defined in terms of simpler constructs. The semantics is a combination of two complementary approaches. It is a shallow embedding of *CML*'s expression language in UTP; for example, sets, sequences, and mappings are all part of UTP and are not further defined. (Issues of undefinedness are dealt with explicitly in Chapter 6.) Other constructs are given as deep embeddings in UTP; for example, the process algebraic constructs must all be given a detailed semantic model as they have no analogue in UTP. In Section 4.1, we give a semantics to an even more restricted subset of *CML*: one in which there are no imperative features and no sequential composition. This impoverished subset is chosen as a vehicle to study the semantic domain and the meaning of the basic timed process algebraic features. It is based on Lowe & Ouaknine's timed testing traces semantics for CSP [19], which also lacks state and sequential composition. Lowe & Ouaknine use a closed presentation for the semantics, following an established tradition. In Section 4.2, we discuss their axioms and posit most of them as theorems of *CML*'s basic semantics. *CML* is strictly more powerful that Lowe & Ouaknine's language in the sense that it allows specifications for processes, which, if they are feasible, may then be refined into process constructs. An appendix contains some of the proofs of theorems corresponding to Lowe & Ouaknine's axioms.

We extend the basic language subset in Section 4.3 to include imperative reactive processes. The major changes are to add sequential composition and a specification statement that corresponds to a VDM operation. This new semantics is built from the previous one by adding preconditions and applying healthiness functions and the result is a timed reactive design semantics. This semantics covers the kernel *CML1* language. It does not include derived constructs, such as while-loops or particular kinds of parallelism that can be defined directly in terms of other constructs. So, in Chapter 5, we describe some of these derived operators with their definitions.

In Chapter 6 we address the issue of undefined expressions in *CML*. We describe a unifying theory for monotonic partial logics with undefined constructs as a sound basis for choosing a treatment of undefinedness in *CML*. This theory justifies using different verification tools for proving facts about *CML* specifications.

In Section 7, we give the first version of the operational semantics for the kernel language. This is tentative work, with no proofs of soundness as yet. The COMPASS DoW requires this work later in the project, but we have brought it forward in order to meet requests from the tooling side of the project.

In Section 8, we give an overview of the copy semantics for the proposed object-oriented features of *CML*. This language extension was required in this phase by the COMPASS DoW, but formal treatment had to be postponed until the next version of the language in order to accommodate the request for an initial operational semantics.

Chapter 3

Unifying Theories of Programming

3.1 Background

Unifying Theories of Programming is originally the work of Hoare & He [14]. It is a long-term research agenda, which can be summarised as follows. Researchers have proposed many different programming theories and practitioners have proposed many different pragmatic programming paradigms. How do we understand the relationship between all of these?

UTP can trace its origins back to the work on predicative programming, which was started by Hehner; see [12] for a summary. It gives three principal ways to study such relationships: 1. by computational paradigm; 2. by level of abstraction; and 3. by method of presentation.

Computational Paradigms UTP groups programming languages according to a classification by computational model; for example, structured, object-oriented, functional, or logical. The technique is to identify common concepts and deal separately with additions and variations. It uses two fundamental scientific principles: (i) simplicity of presentation and (ii) separation of concerns.

Abstraction Orthogonal to organising by computational paradigm, languages could be categorised by their level of abstraction within a particular paradigm. For example, the lowest level of abstraction may be the platform-specific technology of an implementation. At the other end of the spectrum, there might be a very high-level description of overall requirements and how they are captured and analysed. In between, there will be descriptions of components and descriptions of how they will be organised into architectures. Each of these levels will have interfaces specified by contracts of some kind. UTP gives ways of mapping between these levels based on a formal notion of refinement that provides guarantees of correctness all the way from requirements to code.

Presentation The third classification is by the method chosen to present a language definition. There are three scientific methods. (i) *Denotational*, in which each syntactic phrase is given a single mathematical meaning, specification is just a set of denotations, and refinement is a simple correctness criterion of inclusion: every program behaviour is also a specification behaviour. (ii) *Algebraic*, where no direct meaning is given to the language, but instead equalities relate different programs with the same meaning. (iii) *Operational* (most useful for engineers) where programs are defined by how they execute on an idealised abstract mathematical machine, giving a useful guide for compilation, debugging, and testing. As Hoare & He point out, a comprehensive account of a programming theory needs all three kinds of presentation, and the UTP technique allows us to study differences and mutual embeddings, and to derive each from the others by mathematical definition, calculation, and proof.

The UTP Research Agenda has as its ultimate goal to cover all the interesting paradigms of computing, including both declarative and procedural, hardware and software. It presents a theoretical foundation for understanding software and systems engineering, and has been already been exploited in areas such as hardware ([24, 36]), hardware/software co-design ([4]) and component-based systems ([35]). But it also presents an opportunity in constructing new languages, especially ones with heterogeneous paradigms and techniques. Having studied the variety of existing programming languages and identified the major components of programming languages and theories, we can select theories for new, perhaps special-purpose languages. The analogy here is of a theory supermarket, where you shop for exactly those features you need while being confident that the theories plug-and-play together.

A key concept in UTP is the *design*: the familiar precondition-postcondition pair that describes the contract between a programmer and a client. We make great use of this construct in the semantics of *CML*, so we take the opportunity to give an introduction to the theory, which we will then use later in this deliverable. This introduction is adapted from [31].

3.2 Introduction

The book by Hoare & He [14] sets out a research programme to find a common basis in which to explain a wide variety of programming paradigms: unifying theories of programming (UTP). Their technique is to isolate important language features, and give them a denotational semantics. This allows different languages and paradigms to be compared.

The semantic model is an alphabetised version of Tarski's relational calculus, presented in a predicative style that is reminiscent of the schema calculus in the Z [32] notation. Each programming construct is formalised as a relation between an initial and an intermediate or final observation. The collection of these relations forms a *theory* of the paradigm being studied, and it contains three essential parts: an alphabet, a signature, and healthiness conditions.

The *alphabet* is a set of variable names that gives the vocabulary for the theory being studied. Names are chosen for any relevant external observations of behaviour. For instance, programming variables x, y, and z would be part of the alphabet. Also, theories for particular programming paradigms require the observation of extra information; some examples are a flag that says whether the program has started (ok); the current time ($clock$); the number of available resources (res); a trace of the events in the life of the program (tr); a set of refused events (ref) or a flag that says whether the program is waiting for interaction with its environment ($wait$). The *signature* gives the rules for the syntax for denoting objects of the theory. *Healthiness conditions* identify properties that characterise the theory.

Each healthiness condition embodies an important fact about the computational model for the programs being studied.

Example 3.2.1 (Healthiness conditions)

1. *The variable clock gives us an observation of the current time, which moves ever onwards. The predicate B specifies this.*

$$B \mathrel{\widehat=} clock \leq clock'$$

If we add B to the description of some activity, then the variable clock describes the time observed immediately before the activity starts, whereas $clock'$ describes the time observed immediately after the activity ends. If we suppose that P is a healthy program, then we must have that $P \Rightarrow B$.

2. *The variable ok is used to record whether or not a program has started. A sensible healthiness condition is that we should not observe a program's behaviour until it has started; such programs satisfy the following equation.*

$$P = (ok \Rightarrow P)$$

If the program has not started, its behaviour is not described. □

Healthiness conditions can often be expressed in terms of a function ϕ that makes a program healthy. There is no point in applying ϕ twice, since we cannot make a healthy program even healthier. Therefore, ϕ must be idempotent: $P = \phi(P)$; this equation characterises the healthiness condition. For example, we can turn the first healthiness condition above into an equivalent equation, $P = P \wedge B$, and then the following function on predicates $and_B \mathrel{\widehat=} \lambda X \bullet P \wedge B$ is the required idempotent.

The relations are used as a semantic model for unified languages of specification and programming. Specifications are distinguished from programs only by the fact that the latter use a restricted signature. As a consequence of this restriction, programs satisfy a richer set of healthiness conditions.

Unconstrained relations are too general to handle the issue of program termination; they need to be restricted by healthiness conditions. The result is the theory of designs, which is the basis for the study of the other programming paradigms in [14]. Here, we present the general relational setting, and the transition to the theory of designs.

In the next section, we present the most general theory of UTP: the alphabetised predicates. In the following section, we establish that this theory is a complete lattice. Section 3.5 restricts the general theory to designs. Next, in Section 3.6, we present an alternative characterisation of the theory of designs using healthiness conditions. Finally, we conclude with a summary and a brief account of related work.

3.3 The alphabetised relational calculus

The alphabetised relational calculus is similar to Z's schema calculus, except that it is untyped and rather simpler. An *alphabetised predicate* ($P, Q, \ldots,$ ***true***) is an alphabet-predicate pair, where the predicate's free variables are all members of the alphabet. Relations are predicates in which the alphabet is composed of undecorated variables (x, y, z, $\ldots$) and dashed variables (x', a', $\ldots$); the former represent initial observations, and the latter, observations made at a later intermediate or final point. The alphabet of an alphabetised predicate P is denoted αP, and may be divided into its before-variables ($in\alpha P$) and its after-variables ($out\alpha P$). A *homogeneous relation* has $out\alpha P = in\alpha P'$, where $in\alpha P'$ is the set of variables obtained by dashing all variables in the alphabet $in\alpha P$. A *condition* ($b, c, d, \ldots, true$) has an empty output alphabet.

Standard predicate calculus operators can be used to combine alphabetised predicates. Their definitions, however, have to specify the alphabet of the combined predicate. For instance, the alphabet of a conjunction is the union of the alphabets of its components: $\alpha(P \wedge Q) = \alpha P \cup \alpha Q$. Of course, if a variable is mentioned in the alphabet of both P and Q, then they are both constraining the same variable.

A distinguishing feature of UTP is its concern with program development, and consequently program correctness. A significant achievement is that the notion of program correctness is the same in every paradigm in [14]: in every state, the behaviour of an implementation implies its specification.

If we suppose that $\alpha P = \{a, b, a', b'\}$, then the *universal closure* of P is simply $\forall\, a, b, a', b' \bullet P$, which is more concisely denoted as $[\,P\,]$. The correctness of a program P with respect to a specification S is denoted by $S \sqsubseteq P$ (S is refined by P), and is defined as follows.

$$S \sqsubseteq P \quad \textbf{\textit{iff}} \quad [\,P \Rightarrow S\,]$$

Example 3.3.1 (Refinement) *Suppose we have the specification $x' > x \wedge y' = y$, and the implementation $x' = x + 1 \wedge y' = y$. The implementation's correctness is argued as follows.*

$$\begin{aligned}
& x' > x \wedge y' = y \;\sqsubseteq\; x' = x + 1 \wedge y' = y && \text{[definition of } \sqsubseteq\text{]}\\
=\; & [\, x' = x + 1 \wedge y' = y \Rightarrow x' > x \wedge y' = y \,] && \text{[universal one-point rule, twice]}\\
=\; & [\, x + 1 > x \wedge y = y \,] && \text{[arithmetic and reflection]}\\
=\; & true &&
\end{aligned}$$

And so, the refinement is valid. □

As a first example of the definition of a programming constructor, we consider conditionals. Hoare & He use an infix syntax for the conditional operator, and define it as follows.

$$\begin{aligned}
& P \lhd b \rhd Q \;\widehat{=}\; (b \wedge P) \vee (\neg\, b \wedge Q) && \textbf{\textit{if}}\ \alpha b \subseteq \alpha P = \alpha Q\\
& \alpha(P \lhd b \rhd Q) \;\widehat{=}\; \alpha P &&
\end{aligned}$$

Informally, $P \lhd b \rhd Q$ means P if b else Q.

The presentation of conditional as an infix operator allows the formulation of many laws in a helpful way.

L1	$P \lhd b \rhd P = P$	*idempotence*
L2	$P \lhd b \rhd Q = Q \lhd \neg\, b \rhd P$	*symmetry*
L3	$(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R)$	*associativity*
L4	$P \lhd b \rhd (Q \lhd c \rhd R) = (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$	*distributivity*
L5	$P \lhd \mathit{true} \rhd Q = P = Q \lhd \mathit{false} \rhd P$	*unit*
L6	$P \lhd b \rhd (Q \lhd b \rhd R) = P \lhd b \rhd R$	*unreachable branch*
L7	$P \lhd b \rhd (P \lhd c \rhd Q) = P \lhd b \vee c \rhd Q$	*disjunction*
L8	$(P \odot Q) \lhd b \rhd (R \odot S) = (P \lhd b \rhd R) \odot (Q \lhd b \rhd S)$	*interchange*

In the Interchange Law (***L8***), the symbol $\odot$ stands for any truth-functional operator.

For each operator, Hoare & He give a definition followed by a number of algebraic laws as those above. These laws can be proved from the definition. As an example, we present the proof of the Unreachable Branch Law (***L6***).

Example 3.3.2 (Proof of Unreachable Branch (*L6*))

$$
\begin{aligned}
& (P \lhd b \rhd (Q \lhd b \rhd R)) && \text{[L2]}\\
=\ & ((Q \lhd b \rhd R) \lhd \neg\, b \rhd P) && \text{[L3]}\\
=\ & (Q \lhd b \wedge \neg\, b \rhd (R \lhd \neg\, b \rhd P)) && \text{[propositional calculus]}\\
=\ & (Q \lhd \mathit{false} \rhd (R \lhd \neg\, b \rhd P)) && \text{[L5]}\\
=\ & (R \lhd \neg\, b \rhd P) && \text{[L2]}\\
=\ & (P \lhd b \rhd R) && \square
\end{aligned}
$$

Implication is, of course, still the basis for reasoning about the correctness of conditionals. We can, however, prove refinement laws that support a compositional reasoning technique.

Law 3.3.1 (Refinement to conditional)

$$P \sqsubseteq (Q \lhd b \rhd R) \;=\; (P \sqsubseteq b \wedge Q) \wedge (P \sqsubseteq \neg\, b \wedge R) \qquad \square$$

This result allows us to prove the correctness of a conditional by a case analysis on the correctness of each branch. Its proof is as follows.

Proof of Law 3.3.1

$$
\begin{aligned}
& P \sqsubseteq (Q \lhd b \rhd R) && \text{[definition of } \sqsubseteq\text{]}\\
=\ & [\, (Q \lhd b \rhd R) \Rightarrow P \,] && \text{[definition of conditional]}\\
=\ & [\, b \wedge Q \vee \neg\, b \wedge R \Rightarrow P \,] && \text{[propositional calculus]}\\
=\ & [\, b \wedge Q \Rightarrow P \,] \wedge [\, \neg\, b \wedge R \Rightarrow P \,] && \text{[definition of } \sqsubseteq\text{, twice]}
\end{aligned}
$$

$$= (P \sqsubseteq b \wedge Q) \wedge (P \sqsubseteq \neg\, b \wedge R) \qquad \Box$$

A compositional argument is also available for conjunctions.

Law 3.3.2 (Separation of requirements)

$$((P \wedge Q) \sqsubseteq R) = (P \sqsubseteq R) \wedge (Q \sqsubseteq R) \qquad \Box$$

We can prove that an implementation satisfies a conjunction of requirements by considering each conjunct separately. The omitted proof is left as an exercise for the interested reader.

Sequence is modelled as relational composition. Two relations may be composed, providing that the output alphabet of the first is the same as the input alphabet of the second, except only for the use of dashes.

$$\begin{array}{lcl} P(v')\,;\,Q(v) & \mathrel{\widehat=} & \exists\, v_0 \bullet P(v_0) \wedge Q(v_0) \qquad \textbf{if } out\alpha P = in\alpha Q' = \{v'\} \\ in\alpha(P(v')\,;\,Q(v)) & \mathrel{\widehat=} & in\alpha P \\ out\alpha(P(v')\,;\,Q(v)) & \mathrel{\widehat=} & out\alpha Q \end{array}$$

Composition is associative and distributes backwards through the conditional.

$$\begin{array}{lll} \textbf{\textit{L1}} & P\,;\,(Q\,;\,R) = (P\,;\,Q)\,;\,R & \textit{associativity} \\ \textbf{\textit{L2}} & (P \lhd b \rhd Q)\,;\,R = ((P\,;\,R) \lhd b \rhd (Q\,;\,R)) & \textit{left distribution} \end{array}$$

The simple proofs of these laws, and those of a few others in the sequel, are omitted for the sake of conciseness.

The definition of assignment is basically equality; we need, however, to be careful about the alphabet. If $A = \{x, y, \ldots, z\}$ and $\alpha e \subseteq A$, where αe is the set of free variables of the expression e, the assignment $x :=_A e$ of expression e to variable x changes only x's value.

$$\begin{array}{lcl} x :=_A e & \mathrel{\widehat=} & (x' = e \wedge y' = y \wedge \cdots \wedge z' = z) \\ \alpha(x :=_A e) & \mathrel{\widehat=} & A \cup A' \end{array}$$

There is a degenerate form of assignment that changes no variable: it's called "skip", and has the following definition.

$$\begin{array}{lcl} I\!I_A & \mathrel{\widehat=} & (v' = v) \qquad \textbf{if } A = \{v\} \\ \alpha I\!I_A & \mathrel{\widehat=} & A \cup A' \end{array}$$

Skip is the identity of sequence.

$$\textbf{\textit{L5}} \quad P\,;\,I\!I_{\alpha P} = P = I\!I_{\alpha P}\,;\,P \qquad \textit{unit}$$

We keep the numbers of the laws presented in [14] that we reproduce here.

In theories of programming, nondeterminism may arise in one of two ways: either as the result of run-time factors, such as distributed processing; or as the under-specification of implementation choices. Either way, nondeterminism is modelled by choice; the semantics is simply disjunction.

$$P \sqcap Q \;\widehat{=}\; P \vee Q \qquad \textbf{\textit{if}}\ \alpha P = \alpha Q$$

$$\alpha(P \sqcap Q) \;\widehat{=}\; \alpha P$$

The alphabet must be the same for both arguments.

The following law gives an important property of refinement: if P is refined by Q, then offering the choice between P and Q is immaterial; conversely, if the choice between P and Q behaves exactly like P, so that the extra possibility of choosing Q does not add any extra behaviour, then Q is a refinement of P.

Law 3.3.3 (Refinement and nondeterminism)

$$P \sqsubseteq Q \;=\; (P \sqcap Q = P) \qquad \square$$

Proof

$$\begin{aligned}
& P \sqcap Q = P && \textit{[antisymmetry]}\\
={} & (P \sqcap Q \sqsubseteq P) \wedge (P \sqsubseteq P \sqcap Q) && \textit{[definition of } \sqsubseteq\textit{, twice]}\\
={} & [\, P \Rightarrow P \sqcap Q \,] \wedge [\, P \sqcap Q \Rightarrow P \,] && \textit{[definition of } \sqcap\textit{, twice]}\\
={} & [\, P \Rightarrow P \vee Q \,] \wedge [\, P \vee Q \Rightarrow P \,] && \textit{[propositional calculus]}\\
={} & \mathit{true} \wedge [\, P \vee Q \Rightarrow P \,] && \textit{[propositional calculus]}\\
={} & [\, Q \Rightarrow P \,] && \textit{[definition of } \sqsubseteq\textit{]}\\
={} & P \sqsubseteq Q && \square
\end{aligned}$$

Another fundamental result is that reducing nondeterminism leads to refinement.

Law 3.3.4 (Thin nondeterminism)

$$P \sqcap Q \sqsubseteq P \qquad \square$$

The proof is immediate from properties of the propositional calculus.

Variable blocks are split into the commands ***var*** x, which declares and introduces x in scope, and ***end*** x, which removes x from scope. Their definitions are presented below, where A is an alphabet containing x and x'.

$$\textbf{\textit{var}}\ x \;\widehat{=}\; (\exists\, x \bullet I\!I_A) \qquad \alpha(\textbf{\textit{var}}\ x) \;\widehat{=}\; A \setminus \{x\}$$

$$\textbf{\textit{end}}\ x \;\widehat{=}\; (\exists\, x' \bullet I\!I_A) \qquad \alpha(\textbf{\textit{end}}\ x) \;\widehat{=}\; A \setminus \{x'\}$$

The relation ***var*** x is not homogeneous, since it does not include x in its alphabet, but it does include x'; similarly, ***end*** x includes x, but not x'.

The results below state that following a variable declaration by a program Q makes x local in Q; similarly, preceding a variable undeclaration by a program Q makes x' local.

$$(\,\textbf{\textit{var}}\ x \,;\, Q\,) = (\,\exists\, x \bullet Q\,)$$

$$(\,Q \,;\, \textbf{\textit{end}}\ x\,) = (\,\exists\, x' \bullet Q\,)$$

More interestingly, we can use **var** x and **end** x to specify a variable block.

$$(\,\textbf{var}\ x\ ;\ Q\ ;\ \textbf{end}\ x\,) = (\,\exists\, x, x' \bullet Q\,)$$

In programs, we use **var** x and **end** x paired in this way, but the separation is useful for reasoning.

The following laws are representative.

$$\textit{L6} \quad (\,\textbf{var}\ x\ ;\ \textbf{end}\ x\,) = \mathit{II}$$

$$\textit{L8} \quad (\,x := e\ ;\ \textbf{end}\ x\,) = (\,\textbf{end}\ x\,)$$

Variable blocks introduce the possibility of writing programs and equations like that below.

$$\begin{aligned} &(\,\textbf{var}\ x\ ;\ x := 2 * y\ ;\ w := 0\ ;\ \textbf{end}\ x\,) \\ &\quad = \ (\,\textbf{var}\ x\ ;\ x := 2 * y\ ;\ \textbf{end}\ x\,)\ ;\ w := 0 \end{aligned}$$

Clearly, the assignment to w may be moved out of the scope of the the declaration of x, but what is the alphabet in each of the assignments to w? If the only variables are w, x, and y, and suppose that $A = \{w, y, w', y'\}$, then the assignment on the right has the alphabet A; but the alphabet of the assignment on the left must also contain x and x', since they are in scope. There is an explicit operator for making alphabet modifications such as this: *alphabet extension*. If the right-hand assignment is $P \mathrel{\widehat{=}} w :=_A 0$, then the left-hand assignment is denoted by P_{+x}.

$$\begin{aligned} P_{+x} &\mathrel{\widehat{=}} P \wedge x' = x && \text{for } \ x, x' \notin \alpha P \\ \alpha(P_{+x}) &\mathrel{\widehat{=}} \alpha P \cup \{x, x'\} \end{aligned}$$

If Q does not mention x, then the following laws hold.

$$\begin{aligned} &\textit{L1} \quad \textbf{var}\ x\ ;\ Q_{+x}\ ;\ P\ ;\ \textbf{end}\ x \ = \ Q\ ;\ \textbf{var}\ x\ ;\ P\ ;\ \textbf{end}\ x \\ &\textit{L2} \quad \textbf{var}\ x\ ;\ P\ ;\ Q_{+x}\ ;\ \textbf{end}\ x \ = \ \textbf{var}\ x\ ;\ P\ ;\ \textbf{end}\ x\ ;\ Q \end{aligned}$$

Together with the laws for variable declaration and undeclaration, the laws of alphabet extension allow for program transformations that introduce new variables and assignments to them.

3.4 The complete lattice

The refinement ordering is a partial order: reflexive, anti-symmetric, and transitive. Moreover, the set of alphabetised predicates with a particular alphabet A is a complete lattice under the refinement ordering. Its bottom element is denoted $\bot_{\mathrm{A}}$, and is the weakest predicate ***true***; this is the program that aborts, and behaves quite arbitrarily. The top element is denoted $\top^A$, and is the strongest predicate ***false***; this is the program that performs miracles and implements every specification. These properties of abort and miracle are captured in the following two laws, which hold for all P with alphabet A.

$$\textit{L1} \quad \bot_{\mathrm{A}} \sqsubseteq P \qquad\qquad \textit{bottom element}$$

L2 $P \sqsubseteq \top_A$ *top element*

The least upper bound is not defined in terms of the relational model, but by the law ***L1*** below. This law alone is enough to prove laws ***L1A*** and ***L1B***, which are actually more useful in proofs.

L1 $P \sqsubseteq (\bigsqcap S)$ ***iff*** $(\, P \sqsubseteq X$ ***for all*** X ***in*** $S\,)$ *unbounded nondeterminism*

L1A $(\bigsqcap S) \sqsubseteq X$ ***for all*** X ***in*** S *lower bound*

L1B ***if*** $P \sqsubseteq X$ ***for all*** X ***in*** S, ***then*** $P \sqsubseteq (\bigsqcap S)$ *greatest lower bound*

These laws characterise basic properties of least upper bounds.

A function F is *monotonic* if and only if $P \sqsubseteq Q \Rightarrow F(P) \sqsubseteq F(Q)$. Operators like conditional and sequence are monotonic; negation and conjunction are not. There is a class of operators that are all monotonic.

Example 3.4.1 (Disjunctivity and monotonicity) *Suppose that $P \sqsubseteq Q$ and that $\odot$ is disjunctive, or rather, $R \odot (S \sqcap T) = (R \odot S) \sqcap (R \odot T)$. From this, we can conclude that $P \odot R$ is monotonic in its first argument.*

$P \odot R$ *[assumption ($P \sqsubseteq Q$) and Law 3.3.3]*

$= (P \sqcap Q) \odot R$ *[assumption ($\odot$ disjunctive)]*

$= (P \odot R) \sqcap (Q \odot R)$ *[thin nondeterminism]*

$\sqsubseteq Q \odot R$

A symmetric argument shows that $P \odot Q$ is also monotonic in its other argument. In summary, disjunctive operators are always monotonic. The converse is not true: monotonic operators are not always disjunctive. □

Since alphabetised relations form a complete lattice, every construction defined solely using monotonic operators has a fixed-point. Even more, a result by Tarski says that the set of fixed-points form a complete lattice themselves. The extreme points in this lattice are often of interest; for example, $\top$ is the strongest fixed-point of $X = P \,;\, X$, and $\bot$ is the weakest.

The weakest fixed-point of the function F is denoted by $\mu\, F$, and is simply the greatest lower bound (the *weakest*) of all the fixed-points of F.

$$\mu\, F \;\widehat{=}\; \bigsqcap\{\, X \mid F(X) \sqsubseteq X \,\}$$

The strongest fixed-point νF is the dual of the weakest fixed-point.

Hoare & He use weakest fixed-points to define recursion. They write a recursive program as $\mu\, X \bullet \mathcal{C}(X)$, where $\mathcal{C}(X)$ is a predicate that is constructed using monotonic operators and the variable X. As opposed to the variables in the alphabet, X stands for a predicate itself, and we call it the recursive variable. Intuitively, occurrences of X in $\mathcal{C}$ stand for recursive calls to $\mathcal{C}$ itself. The definition of recursion is as follows.

$\mu\, X \bullet \mathcal{C}(X) \;\widehat{=}\; \mu\, F$ ***where*** $F \;\widehat{=}\; \lambda\, X \bullet \mathcal{C}(X)$

The standard laws that characterise weakest fixed-points are valid.

L1 $\mu\, F \sqsubseteq Y$ if $F(Y) \sqsubseteq Y$ *weakest fixed-point*

$$\textbf{\textit{L2}} \quad [\, F(\mu\, F) = \mu\, F \,] \qquad \textit{fixed-point}$$

L1 establishes that $\mu\, F$ is weaker than any fixed-point; ***L2*** states that $\mu\, F$ is itself a fixed-point. From a programming point of view, ***L2*** is just the copy rule.

Proof of *L1*

$$\begin{array}{ll}
\quad F(Y) \sqsubseteq Y & \textit{[set comprehension]} \\
= Y \in \{\, X \mid F(X) \sqsubseteq X \,\} & \textit{[lattice law \textbf{L1A}]} \\
\Rightarrow \bigsqcap\{\, X \mid F(X) \sqsubseteq X \,\} \sqsubseteq Y & \textit{[definition of } \mu\, F\textit{]} \\
= \mu\, F \sqsubseteq Y & \square
\end{array}$$

Proof of *L2*

$$\begin{array}{ll}
\quad \mu\, F = F(\mu\, F) & \textit{[mutual refinement]} \\
= \mu\, F \sqsubseteq F(\mu\, F) \land F(\mu\, F) \sqsubseteq \mu\, F & \textit{[fixed-point law \textbf{L1}]} \\
\Longleftarrow\ F(F(\mu\, F)) \sqsubseteq F(\mu\, F) \land F(\mu\, F) \sqsubseteq \mu\, F & \textit{[}F\textit{ monotonic]} \\
\Longleftarrow\ F(\mu\, F) \sqsubseteq \mu\, F & \textit{[definition]} \\
= F(\mu\, F) \sqsubseteq \bigsqcap\{\, X \mid F(X) \sqsubseteq X \,\} & \textit{[lattice law \textbf{L1B}]} \\
\Longleftarrow\ \forall\, X \in \{\, X \mid F(X) \sqsubseteq X \,\} \bullet F(\mu\, f) \sqsubseteq X & \textit{[comprehension]} \\
= \forall\, X \bullet F(X) \sqsubseteq X \Rightarrow F(\mu\, F) \sqsubseteq X & \textit{[transitivity of } \sqsubseteq\textit{]} \\
\Longleftarrow\ \forall\, X \bullet F(X) \sqsubseteq X \Rightarrow F(\mu\, F) \sqsubseteq F(X) & \textit{[}F\textit{ monotonic]} \\
\Longleftarrow\ \forall\, X \bullet F(X) \sqsubseteq X \Rightarrow \mu\, F \sqsubseteq X & \textit{[fixed-point law \textbf{L1}]} \\
= \mathit{true} & \square
\end{array}$$

Iteration The while loop is written $b * P$: while b is true, execute the program P. This can be defined in terms of the weakest fixed-point of a conditional expression.

$$b * P \ \widehat{=}\ \mu\, X \bullet (\, (P \,;\, X) \lhd b \rhd I\!I\,)$$

Example 3.4.2 (Non-termination) *If b always remains true, then obviously the loop $b * P$ never terminates, but what is the semantics for this non-termination? The simplest example of such an iteration is $\mathit{true} * I\!I$, which has the semantics $\mu\, X \bullet X$.*

$$\begin{array}{ll}
\quad \mu\, X \bullet X & \textit{[definition of least fixed-point]} \\
= \bigsqcap\{\, Y \mid (\lambda\, X \bullet X)(Y) \sqsubseteq Y \,\} & \textit{[function application]} \\
= \bigsqcap\{\, Y \mid Y \sqsubseteq Y \,\} & \textit{[reflexivity of } \sqsubseteq\textit{]} \\
= \bigsqcap\{\, Y \mid \mathit{true} \,\} & \textit{[property of } \bigsqcap\textit{]} \\
= \bot & \square
\end{array}$$

A surprising, but simple, consequence of Example 3.4.2 is that a program can recover from a non-terminating loop!

Example 3.4.3 (Aborting loop) *Suppose that the sole state variable is x and that c is a constant.*

$$
\begin{aligned}
& (b * P);\ x := c && \text{[Example 3.4.2]}\\
=\ & \bot;\ x := c && \text{[definition of } \bot\text{]}\\
=\ & \textbf{\textit{true}};\ x := c && \text{[definition of assignment]}\\
=\ & \textbf{\textit{true}};\ x' = c && \text{[definition of composition]}\\
=\ & \exists\, x_0 \bullet \textbf{\textit{true}} \land x' = c && \text{[predicate calculus]}\\
=\ & x' = c && \text{[definition of assignment]}\\
=\ & x := c && \Box
\end{aligned}
$$

Example 3.4.3 is rather disconcerting: in ordinary programming, there is no recovery from a non-terminating loop. It is the purpose of *designs* to overcome this deficiency in the programming model; we return to this in Section 3.5.

3.5 Designs

The problem pointed out in Section can be explained as the failure of general alphabetised predicates P to satisfy the equation below.

$$\textbf{\textit{true}}\ ;\ P \ = \ \textbf{\textit{true}}$$

In particular, in Example 3.4.3 we presented a non-terminating loop which, when followed by an assignment, behaves like the assignment. Operationally, it is as though the non-terminating loop could be ignored.

The solution is to consider a subset of the alphabetised predicates in which a particular observational variable, called *ok*, is used to record information about the start and termination of programs. The above equation holds for predicates P in this set. As an aside, we observe that *false* cannot possibly belong to this set, since $\textbf{\textit{false}} = \textbf{\textit{false}}\ ;\ \textbf{\textit{true}}$.

The predicates in this set are called designs. They can be split into precondition-postcondition pairs, and are in the same spirit as specification statements used in refinement calculi. As such, they are a basis for unifying languages and methods like B [1], VDM [16], Z [32], and refinement calculi [21, 3, 22].

In designs, *ok* records that the program has started, and ok' records that it has terminated. These are auxiliary variables, in the sense that they appear in a design's alphabet, but they never appear in code or in preconditions and postconditions.

In implementing a design, we are allowed to assume that the precondition holds, but we have to fulfill the postcondition. In addition, we can rely on the program being started, but we must ensure that the program terminates. If the precondition does not hold, or the program does not start, we are not committed to establish the postcondition nor even to make the program terminate.

A design with precondition P and postcondition Q, for predicates P and Q not containing *ok* or ok', is written $(\ P \vdash Q\ )$. It is defined as follows.

$$(\ P \vdash Q\ ) \ \widehat{=}\ (\ ok \land P \Rightarrow ok' \land Q\ )$$

If the program starts in a state satisfying P, then it will terminate, and on termination Q will be true.

Abort and miracle are defined as designs in the following examples. Abort has precondition *false* and is never guaranteed to terminate.

Example 3.5.1 (Abort)

$$
\begin{array}{ll}
\quad \mathbf{false} \vdash \mathbf{false} & \text{[definition of design]} \\
= ok \wedge \mathbf{false} \Rightarrow ok' \wedge \mathbf{false} & \text{[}\mathbf{false}\text{ zero for conjunction]} \\
= \mathbf{false} \Rightarrow ok' \wedge \mathbf{false} & \text{[vacuous implication]} \\
= \mathbf{true} & \text{[vacuous implication]} \\
= \mathbf{false} \Rightarrow ok' \wedge \mathbf{true} & \text{[}\mathbf{false}\text{ zero for conjunction]} \\
= ok \wedge \mathbf{false} \Rightarrow ok' \wedge \mathbf{true} & \text{[definition of design]} \\
= \mathbf{false} \vdash \mathbf{true} & \Box
\end{array}
$$

Miracle has precondition *true*, and establishes the impossible: *false*.

Example 3.5.2 (Miracle)

$$
\begin{array}{ll}
\quad \mathbf{true} \vdash \mathbf{false} & \text{[definition of design]} \\
= ok \wedge \mathbf{true} \Rightarrow ok' \wedge \mathbf{false} & \text{[}\mathbf{true}\text{ unit for conjunction]} \\
= ok \Rightarrow \mathbf{false} & \text{[contradiction]} \\
= \neg\, ok & \Box
\end{array}
$$

A reassuring result about a design is the fact that refinement amounts to either weakening the precondition, or strengthening the postcondition in the presence of the precondition. This is established by the result below.

Law 3.5.1 *Refinement of designs*

$$P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2 \;=\; [\, P_1 \wedge Q_2 \Rightarrow Q_1 \,] \wedge [\, P_1 \Rightarrow P_2 \,] \qquad \Box$$

Proof

$$
\begin{array}{ll}
\quad P_1 \vdash Q_1 \;\sqsubseteq\; P_2 \vdash Q_2 & \text{[definition of } \sqsubseteq\text{]} \\
= [\, (P_2 \vdash Q_2) \Rightarrow (P_1 \vdash Q_1) \,] & \text{[definition of design, twice]} \\
= [\, (ok \wedge P_2 \Rightarrow ok' \wedge Q_2) \Rightarrow (ok \wedge P_1 \Rightarrow ok' \wedge Q_1) \,] & \\
 & \text{[case analysis on } ok\text{]} \\
= [\, (P_2 \Rightarrow ok' \wedge Q_2) \Rightarrow (P_1 \Rightarrow ok' \wedge Q_1) \,] & \text{[case analysis on } ok'\text{]} \\
= [\, ((P_2 \Rightarrow Q_2) \Rightarrow (P_1 \Rightarrow Q_1)) \wedge (\neg P_2 \Rightarrow \neg P_1) \,] & \text{[propositional calculus]} \\
= [\, ((P_2 \Rightarrow Q_2) \Rightarrow (P_1 \Rightarrow Q_1)) \wedge (P_1 \Rightarrow P_2) \,] & \text{[predicate calculus]} \\
= [\, P_1 \wedge Q_2 \Rightarrow Q_1 \,] \wedge [\, P_1 \Rightarrow P_2 \,] & \Box
\end{array}
$$

The most important result, however, is that abort is a zero for sequence. This was, after all, the whole point for the introduction of designs.

$$\mathit{L1} \qquad \mathbf{true} \,;\, (P \vdash Q) = \mathbf{true} \qquad \textit{left-zero}$$

Proof

$$
\begin{array}{lr}
\quad \mathbf{true} \,;\, (P \vdash Q) & \textit{[property of sequential composition]} \\
= \exists\, ok_0 \bullet \mathbf{true} \,;\, (P \vdash Q)[ok_0/ok] & \textit{[case analysis]} \\
= (\, \mathbf{true} \,;\, (P \vdash Q)[true/ok] \,) \lor (\, \mathbf{true} \,;\, (P \vdash Q)[false/ok] \,) & \textit{[property of design]} \\
= (\, \mathbf{true} \,;\, (P \vdash Q)[true/ok] \,) \lor (\, \mathbf{true} \,;\, \mathbf{true} \,) & \textit{[relational calculus]} \\
= (\, \mathbf{true} \,;\, (P \vdash Q)[true/ok] \,) \lor \mathbf{true} & \textit{[propositional calculus]} \\
= \mathbf{true} & \square
\end{array}
$$

In this new setting, it is necessary to redefine assignment and skip, as those introduced previously are not designs.

$$(x := e) \;\widehat{=}\; (\, \mathbf{true} \vdash x' = e \land y' = y \land \cdots \land z' = z \,)$$

$$\mathit{I\!I}_{\mathbf{D}} \;\widehat{=}\; (\, \mathbf{true} \vdash \mathit{I\!I} \,)$$

Their existing laws hold, but it is necessary to prove them again, as their definitions changed.

$$
\begin{array}{ll}
\boldsymbol{L2} & (v := e \,;\, v := f(v)) = (v := f(e)) \\
\boldsymbol{L3} & (v := e \,;\, (P \lhd b(v) \rhd Q)) = ((v := e \,;\, P) \lhd b(e) \rhd (v := e \,;\, Q)) \\
\boldsymbol{L4} & (\mathit{I\!I}_{\mathbf{D}} \,;\, (P \vdash Q)) = (P \vdash Q)
\end{array}
$$

As as an example, we present the proof of ***L2***.

Proof of *L2*

$$
\begin{array}{lr}
\quad v := e \,;\, v := f(v) & \textit{[definition of assignment, twice]} \\
= (\, \mathbf{true} \vdash v' = e \,) \,;\, (\, \mathbf{true} \vdash v' = f(v) \,) & \textit{[case analysis on } ok_0\textit{]} \\
= (\, (\, \mathbf{true} \vdash v' = e \,)[true/ok'] \,;\, (\, \mathbf{true} \vdash v' = f(v) \,)[true/ok] \,) \lor & \\
\quad \neg\, ok \,;\, \mathbf{true} & \textit{[definition of design]} \\
= (\, (\, ok \Rightarrow v' = e \,) \,;\, (\, ok' \land v' = f(v) \,) \,) \lor \neg\, ok & \textit{[relational calculus]} \\
= ok \Rightarrow (\, v' = e \,;\, (\, ok' \land v' = f(v) \,) \,) & \textit{[assignment composition]} \\
= ok \Rightarrow ok' \land v' = f(e) & \textit{[definition of design]} \\
= (\, \mathbf{true} \vdash v' = f(e) \,) & \textit{[definition of assignment]} \\
= v := f(e) & \square
\end{array}
$$

If any of the program operators are applied to designs, then the result is also a design. This follows from the laws below, for choice, conditional, sequence, and recursion. The choice between two designs is guaranteed to terminate when they both are; since either of them may be chosen, then either postcondition may be established.

$$\boldsymbol{T1} \quad (\, (P_1 \vdash Q_1) \sqcap (P_2 \vdash Q_2) \,) = (\, P_1 \land P_2 \vdash Q_1 \lor Q_2 \,)$$

If the choice between two designs depends on a condition b, then so do the precondition and the postcondition of the resulting design.

$$\textbf{\textit{T2}} \quad ((P_1 \vdash Q_1) \lhd b \rhd (P_2 \vdash Q_2)) \\ = ((P_1 \lhd b \rhd P_2) \vdash (Q_1 \lhd b \rhd Q_2))$$

A sequence of designs $(P_1 \vdash Q_1)$ and $(P_2 \vdash Q_2)$ terminates when P_1 holds, and Q_1 is guaranteed to establish P_2. On termination, the sequence establishes the composition of the postconditions.

$$\textbf{\textit{T3}} \quad ((P_1 \vdash Q_1) ; (P_2 \vdash Q_2)) \\ = ((\neg (\neg P_1 ; \textbf{\textit{true}}) \wedge (Q_1 \textbf{\textit{ wp }} P_2)) \vdash (Q_1 ; Q_2))$$

where $Q_1 \textbf{\textit{ wp }} P_2$ is the weakest precondition under which execution of Q_1 is guaranteed to achieve the postcondition P_2. It is defined in [14] as

$$Q \textbf{\textit{ wp }} P = \neg (Q ; \neg P)$$

Preconditions can be relations, and this fact complicates the statement of Law ***T3***; if the P_1 is a condition instead, then the law is simplified as follows.

$$\textbf{\textit{T3'}} \quad ((p_1 \vdash Q_1) ; (P_2 \vdash Q_2)) = (p_1 \wedge (Q_1 \textbf{\textit{ wp }} P_2)) \vdash (Q_1 ; Q_2))$$

A recursively defined design has as its body a function on designs; as such, it can be seen as a function on precondition-postcondition pairs (X, Y). Moreover, since the result of the function is itself a design, it can be written in terms of a pair of functions F and G, one for the precondition and one for the postcondition.

As the recursive design is executed, the precondition F is required to hold over and over again. The strongest recursive precondition so obtained has to be satisfied, if we are to guarantee that the recursion terminates. Similarly, the postcondition is established over and over again, in the context of the precondition. The weakest result that can possibly be obtained is that which can be guaranteed by the recursion.

$$\textbf{\textit{T4}} \quad (\mu X, Y \bullet (F(X, Y) \vdash G(X, Y))) = (P(Q) \vdash Q) \\ \textbf{\textit{where }} P(Y) = (\nu X \bullet F(X, Y)) \textbf{\textit{ and }} Q = (\mu Y \bullet P(Y) \Rightarrow G(P(Y), Y))$$

Further intuition comes from the realisation that we want the least refined fixed-point of the pair of functions. That comes from taking the strongest precondition, since the precondition of every refinement must be weaker, and the weakest postcondition, since the postcondition of every refinement must be stronger.

Like the set of general alphabetised predicates, designs form a complete lattice. We have already presented the top and the bottom (miracle and abort).

$$\top_D \mathrel{\widehat{=}} (\textbf{\textit{true}} \vdash \textbf{\textit{false}}) = \neg ok$$

$$\bot_D \mathrel{\widehat{=}} (\textbf{\textit{false}} \vdash \textbf{\textit{true}}) = \textbf{\textit{true}}$$

The least upper bound and the greatest lower bound are established in the following theorem.

Theorem 3.5.1 *Meets and joins*

$$\textstyle\bigsqcap_i (\, P_i \vdash Q_i \,) = (\, \bigwedge_i P_i \,) \vdash (\, \bigvee_i Q_i \,)$$

$$\textstyle\bigsqcup_i (\, P_i \vdash Q_i \,) = (\, \bigvee_i P_i \,) \vdash (\, \bigwedge_i P_i \Rightarrow Q_i \,)$$

As with the binary choice, the choice $\bigsqcap_i (\, P_i \vdash Q_i \,)$ terminates when all the designs do, and it establishes one of the possible postconditions. The least upper bound models a form of choice that is conditioned by termination: only the terminating designs can be chosen. The choice terminates if any of the designs does, and the postcondition established is that of any of the terminating designs.

3.6 Healthiness conditions

Another way of characterising the set of designs is by imposing healthiness conditions on the alphabetised predicates. Hoare & He identify four healthiness conditions that they consider of interest: ***H1*** to ***H4***. We discuss each of them.

3.6.1 ***H1***: unpredictability

A relation R is ***H1*** healthy if and only if $R = (ok \Rightarrow R)$. This means that observations cannot be made before the program has started. A consequence is that R satisfies the left-zero and unit laws below.

$$\textbf{true} \,;\, R = \textbf{true} \quad \text{and} \quad \mathit{II}_{\mathbf{D}} \,;\, R = R$$

We now present a proof of these results.

Designs with left-units and left-zeros are *H1*

$$
\begin{aligned}
& R && \text{[assumption ($\mathit{II}_{\mathbf{D}}$ is left-unit)]} \\
&= \mathit{II}_{\mathbf{D}} \,;\, R && \text{[$\mathit{II}_{\mathbf{D}}$ definition]} \\
&= (\, \textbf{true} \vdash \mathit{II}_{\mathbf{D}} \,) \,;\, R && \text{[design definition]} \\
&= (\, ok \Rightarrow ok' \wedge \mathit{II} \,) \,;\, R && \text{[relational calculus]} \\
&= (\, \neg\, ok \,;\, R \,) \vee (\, \mathit{II} \,;\, R \,) && \text{[relational calculus]} \\
&= (\, \neg\, ok \,;\, \textbf{true} \,;\, R \,) \vee (\, \mathit{II} \,;\, R \,) && \text{[assumption (\textbf{true} is left-zero)]} \\
&= \neg\, ok \vee (\, \mathit{II} \,;\, R \,) && \text{[assumption (II is left-unit)]} \\
&= \neg\, ok \vee R && \text{[relational calculus]} \\
&= ok \Rightarrow R && \square
\end{aligned}
$$

H1 designs have a left-zero

$$
\begin{array}{ll}
\quad \mathbf{true} \,;\, R & \text{[assumption (} R \text{ is } \mathbf{H1}\text{)]} \\
= \mathbf{true} \,;\, (\, ok \Rightarrow R \,) & \text{[relational calculus]} \\
= (\, \mathbf{true} \,;\, \neg\, ok \,) \lor (\, \mathbf{true} \,;\, R \,) & \text{[relational calculus]} \\
= \mathbf{true} \lor (\, \mathbf{true} \,;\, R \,) & \text{[relational calculus]} \\
= \mathbf{true} & \square
\end{array}
$$

H1 designs have a left-unit

$$
\begin{array}{ll}
\quad I\!I_{\mathbf{D}} \,;\, R & \text{[definition of } I\!I_{\mathbf{D}}\text{]} \\
= (\, \mathbf{true} \vdash I\!I_{\mathbf{D}} \,) \,;\, R & \text{[definition of design]} \\
= (\, ok \Rightarrow ok' \land I\!I \,) \,;\, R & \text{[relational calculus]} \\
= (\, \neg\, ok \,;\, R \,) \lor (\, ok \land R \,) & \text{[relational calculus]} \\
= (\, \neg\, ok \,;\, \mathbf{true} \,;\, R \,) \lor (\, ok \land R \,) & \text{[}\mathbf{true} \text{ is left-zero]} \\
= (\, \neg\, ok \,;\, \mathbf{true} \,) \lor (\, ok \land R \,) & \text{[relational calculus]} \\
= \neg\, ok \lor (\, ok \land R \,) & \text{[relational calculus]} \\
= ok \Rightarrow R & \text{[} R \text{ is } \mathbf{H1}\text{]} \\
= R & \square
\end{array}
$$

This means that we could use the left-zero and unit laws to characterise ***H1***.

3.6.2 *H2*: possible termination

The second healthiness condition is $[\, R[false/ok'] \Rightarrow R[true/ok'] \,]$. This means that if R is satisfied when ok' is $false$, it is also satisfied then ok' is $true$. In other words, R cannot *require* nontermination, so that it is always possible to terminate.

The designs are exactly those relations that are ***H1*** and ***H2*** healthy. First we present a proof that relations that are ***H1*** and ***H2*** healthy are designs.

H1 and H2 healthy relations are designs Let $R^f = R[false/ok']$ and $R^t = R[true/ok']$.

$$
\begin{array}{ll}
\quad R & \text{[assumption (} R \text{ is } \mathbf{H1}\text{)]} \\
= ok \Rightarrow R & \text{[propositional calculus]} \\
= ok \Rightarrow (\, \neg\, ok' \land R^f \,) \lor (\, ok' \land R^t \,) & \text{[assumption (} R \text{ is } \mathbf{H2}\text{)]} \\
= ok \Rightarrow (\, \neg\, ok' \land R^f \land R^t \,) \lor (\, ok' \land R^t \,) & \text{[propositional calculus]} \\
= ok \Rightarrow (\, (\, (\, \neg\, ok' \land R^f \,) \lor ok' \,) \land R^t \,) & \text{[propositional calculus]} \\
= ok \Rightarrow (\, (\, R^f \lor ok' \,) \land R^t \,) & \text{[propositional calculus]} \\
= ok \Rightarrow (\, R^f \land R^t \,) \lor (\, ok' \land R^t \,) & \text{[assumption (} R \text{ is } \mathbf{H2}\text{)]} \\
= ok \Rightarrow R^f \lor (\, ok' \land R^t \,) & \text{[propositional calculus]} \\
= ok \land \neg\, R^f \Rightarrow ok' \land R^t & \text{[design definition]}
\end{array}
$$

$= \neg R^f \vdash R^t$ $\square$

It is very simple to prove that designs are ***H1*** healthy; we present the proof that designs are ***H2*** healthy.

Designs are *H2*

$$
\begin{array}{ll}
(P \vdash Q)[\mathit{false}/ok'] & \text{[definition of design]} \\
= (ok \wedge P \Rightarrow \textbf{\textit{false}}) & \text{[propositional calculus]} \\
\Rightarrow (ok \wedge P \Rightarrow Q) & \text{[definition of design]} \\
= (P \vdash Q)[\mathit{true}/ok'] & \square
\end{array}
$$

While ***H1*** characterises the rôle of ok, ***H2*** characterises ok'. Therefore, it should not be a surprise that, together, they identify the designs.

3.6.3 *H3*: dischargeable assumptions

The healthiness condition ***H3*** is specified as an algebraic law: $R = R \,;\, I\!I_{D}$. A design satisfies ***H3*** exactly when its precondition is a condition. This is a very desirable property, since restrictions imposed on dashed variables in a precondition can never be discharged by previous or successive components. For example, $x' = 2 \vdash \mathit{true}$ is a design that can either terminate and give an arbitrary value to x, or it can give the value 2 to x, in which case it is not required to terminate. This is a rather bizarre behaviour.

A design is *H3 iff* its assumption is a condition

$$
\begin{array}{ll}
((P \vdash Q) = ((P \vdash Q) \,;\, I\!I_{D})) & \text{[definition of design-skip]} \\
= ((P \vdash Q) = ((P \vdash Q) \,;\, (\textbf{\textit{true}} \vdash I\!I_{D}))) & \text{[sequence of designs]} \\
= ((P \vdash Q) = (\neg(\neg P \,;\, \textbf{\textit{true}}) \wedge \neg(Q \,;\, \neg \textbf{\textit{true}}) \vdash Q \,;\, I\!I_{D})) & \text{[skip unit]} \\
= ((P \vdash Q) = (\neg(\neg P \,;\, \textbf{\textit{true}}) \vdash Q)) & \text{[design equality]} \\
= (\neg P = \neg P \,;\, \textbf{\textit{true}}) & \text{[propositional calculus]} \\
= (P = P \,;\, \textbf{\textit{true}}) & \square
\end{array}
$$

The final line of this proof states that $P = \exists\, v' \bullet P$, where v' is the output alphabet of P. Thus, none of the after-variables' values are relevant: P is a condition only on the before-variables.

3.6.4 *H4*: feasibility

The final healthiness condition is also algebraic: $R \,;\, \textbf{\textit{true}} = \textbf{\textit{true}}$. Using the definition of sequence, we can establish that this is equivalent to $\exists\, v' \bullet R$, where v' is the output alphabet of R. In words, this means that for *every* initial value of the observational variables on the input alphabet, there exist final values for the variables of the output alphabet: more concisely, establishing a final state is feasible. The design $\top_{D}$ is not ***H4*** healthy, since miracles are not feasible.

Chapter 4

UTP Semantics for ***CML1***

We give in this Chapter the semantics for *CML1*, the language of timed imperative reactive processes that combines VDM with discrete-time CSP. We focus on the kernel subset of the language that describes actions rather than process-level combinators; the latter are closely related to a subset of the former. Imperative features of *CML* are represented by assignment and specification statements. Other programming-language features are derived from more basic control structures. For example, the ***while*** loop is derived from a combination of recursion, conditional, and sequential composition. In this semantics, we use the notation of UTP rather than the syntax of VDM; Chapter 5 gives the correspondence between the two.

The CSP timed part of *CML* is given a semantics closely related to Lowe & Ouaknine's Timed Testing Traces [19], and this in turn is related to the standard semantics for CSP. The fundamental notions here are those of events, traces and refusals.

An *event* is an atomic and instantaneous interaction between a CSP process and its environment. This might be the observation of a synchronisation event, or the observation of a communication of a value on a channel.

A *trace* of a CSP process is a sequence of events recorded by an observer. This trace may be either finite or infinite, the latter being necessary for a complete treatment of unbounded nondeterminism. In our semantics we restrict ourselves to finite traces.

Consider the following CSP process: $a \to b \to STOP$. Its behaviour is to engage in the two events a and b, in that order. The meaning of this process is given by its possible traces, and there are exactly three of these: (i) $\langle\rangle$, (ii) $\langle a \rangle$, and (iii) $\langle a, b \rangle$. Each trace represents an observation that can be made of the process. The first is the observation before anything happens; the second after the a has occurred, but before the b; and the third after both the a and b events have happened.

A *refusal* of a process is an experiment, where the process refuses to engage in a set of events offered by its environment. In our example process, $a \to b \to STOP$, we can conduct this kind of experiment at different points in the evolution of the process. We could, for instance, conduct it before anything has happened at all. Suppose that the set of possible events is $\{a, b, c\}$. If we were to offer the entire set to the process, then it could not refuse to engage in a, but it could refuse both b and c. If we were to make a meaner offer (that is, a subset of our original offer), say only $\{b, c\}$, then it would still refuse. Here are all the refusals:

1. After the trace $\langle\rangle$: $\emptyset$, $\{b\}$, $\{c\}$, $\{b, c\}$
2. After the trace $\langle a\rangle$: $\emptyset$, $\{a\}$, $\{c\}$, $\{a, c\}$
3. After the trace $\langle a, b\rangle$: $\emptyset$, $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{a, c\}$, $\{b, c\}$, $\{a, b, c\}$

Of course, each refusal is sensitive to the point at which the experiment is made; that is, it is sensitive to the value of the trace that describes what has been observed. This trace-refusal pair is known as a *failure*.

As well as being able to see a process's failures, an observer can also detect the passage of time. In our semantics, this is represented by observing a global clock advancing—the *tock* event marks the end of a granule of time. Refusal experiments can be made only at this granularity of time.

The language that we are considering consists of the following actions.

- Assignment: $x := e$.
- Specification statement: $w : [\, pre, post \,]$.
- Deadlocked action: $STOP$.
- Successful termination: $SKIP$.
- Sequential composition: $P \,;\, Q$.
- Prefixed action: $a \rightarrow P$.
- Internal choice: $P \sqcap Q$.
- External choice: $P \mathrel{\Box} Q$.
- Timeout: $P \overset{n}{\rhd} Q$.
- Parallel composition: $P \parallel_{cs} Q$.
- Hiding: $P \setminus A$.
- Recursion: $\mu\, X \bullet P(X)$.

4.1 Timed Testing Traces

Our first semantics assumes, as a temporary restriction, that *CML* programs do not use sequential composition or imperative state. This simplification allows us to ignore assignments, specification statements, and successful termination.

Let Σ be the universe of events and let the clock event be $tock \notin \Sigma$. Our semantic domain consists of traces with embedded refusal sets. For example, the trace

$$\langle a, b, \{b, c\}, tock, \emptyset, tock, c\rangle$$

represents the observation:

- The trace $\langle a, b\rangle$ occurred in the first time interval.

- At the end of this trace, the process refused the set of events $\{b, c\}$.
- No events were observed during the second time interval.
- The third time interval is incomplete, but the trace $\langle c \rangle$ was observed so far.

Traces bearing this structure are drawn from the following set:

Definition 4.1.1

$$timedTrace \mathrel{\widehat{=}} (\Sigma + \mathbb{P}(\Sigma).tock)^*$$

This definition uses a variation on the standard notation for regular expressions, where the dot is to be understood as concatenation. Refusal sets immediate precede *tock* events and these pairs are separated by sequences of events; just what we need.

Notice that timed testing traces are able to record quite subtle information. Consider the behaviour of an action P, with a universe of events including only a and b. P never offers to engage in b, but P offers to engage in a during every other time interval. Here is a possible trace:

$$\langle \{a, b\}, tock, \{b\}, tock, \{a, b\}, tock, \{b\}, tock, \{a, b\}, tock \rangle$$

Timed traces encode all observations we wish to make about particular executions of *CML* processes: the trace of events marked out by the passage of time and the refusal experiments that can be made during execution. So we introduce a variable tt' to record such an observation. The variable tt' records observations made in the intermediate or final states of an action. If P is a relation describing the behaviour of an action, then there is no need for the complementary observation tt describing the value of the the trace before P, since there is no sequential composition, there was nothing before P. So all predicates describing timed testing traces have the alphabet $\{tt'\}$ and satisfy the following healthiness condition:

Definition 4.1.2

$$\boldsymbol{T0}(P) = P \wedge tt' \in timedTrace$$

As a conjunctive function, ***T0*** is a monotonic idempotent. Applying ***T0*** gives us a way of type-checking the variable tt'.

We define some simple operators on sequences. The function *squash* compacts a finite function $f : \mathbb{N} \twoheadrightarrow X$ to produce a sequence (the function is taken from Z [29]). For example, $squash(\{2 \mapsto a, 3 \mapsto b, 10 \mapsto c\}) = \langle a, b, c \rangle$. This allows us to construct a simple function to filter a sequence against a set. For example, $\langle a, b, c, d, e \upharpoonright \{b, d\} = \langle a, c, e \rangle$.

Definition 4.1.3

$$squash(\emptyset) = \langle \rangle$$
$$squash(f) = \langle f(min(\operatorname{dom} f)) \rangle \frown squash(\{min(\operatorname{dom} f)\} \lhd f)$$
$$t \upharpoonright S = squash(t \rhd S)$$

Now we can define functions to extract information from a trace. The function $trace(t)$ throws away the refusal sets. The function $refsduring(t)$ collects together the refusal set in the trace, throwing away the trace of events. The function $refusals(t)$ calculates all the events being refused at different points during the trace.

Definition 4.1.4

$$trace(t) = t \upharpoonright \Sigma^{tock}$$
$$refsduring(t) = \operatorname{ran}(t \rhd \mathbb{P}(\Sigma))$$
$$refusals(t) = \bigcup refsduring(t)$$

The following lemma gives rules for calculating the trace of events from a testing trace:

Lemma 4.1.1 (Trace extraction)

$$\begin{array}{rcll} trace(\langle\rangle) & = & \langle\rangle & \\ trace(\langle a\rangle \frown t) & = & \langle a\rangle \frown trace(t) & \textit{if } a \in \Sigma^{tock} \\ trace(\langle X\rangle \frown t) & = & trace(t) & \textit{if } X \in \mathbb{P}(\Sigma) \end{array}$$

We define an order relation on traces: $s \preceq t$ holds when s contains less information that t.

Definition 4.1.5 (Testing trace precedence) *Let* $a \in \Sigma \wedge X \subseteq Y$, *then*

$$\begin{array}{rcll} \langle\rangle & \preceq & u & \\ \langle a\rangle \frown t & \preceq & \langle a\rangle \frown u & \textit{if } t \preceq u \\ \langle X, tock\rangle \frown t & \preceq & \langle Y, tock\rangle \frown u & \textit{if } t \preceq u \end{array}$$

This is a stronger relation than the usual prefix relation on traces, $\_ \leq \_$:

Lemma 4.1.2 (Precedence traces)

$$t \preceq u \Rightarrow trace(t) \leq trace(u)$$

Proof *by induction on* t.

A similar result holds for the refusals over testing traces:

Lemma 4.1.3 (Precedence refusals)

$$t \preceq u \wedge a \in refusals(t) \Rightarrow a \in refusals(u)$$

4.1.1 STOP

Our first language construct is the deadlocked action: $STOP$. This action never engages in any events, so we must have that $trace(tt') \upharpoonright \Sigma = \langle\rangle$: no events are ever observed. $STOP$ deadlocks events but it cannot deadlock the clock, so $tock$ events can happen freely. Finally, every refusal experiment must fail. This is all captured by the simple specification $trace(tt') \in tock^*$, where x^* is the regular expression that describes all the finite sequences containing only the event x (the Kleene closure). We do not care about the value of the refusal sets and so leave them unconstrained. All this makes sense as the semantics of $STOP$ only if tt' is a testing trace, and this is guaranteed by an application of the ***T0*** healthiness condition.

Definition 4.1.6 ($STOP$ testing trace)

$$STOP \mathrel{\widehat=} \mathbf{T0}(trace(tt') \in tock^*)$$

4.1.2 Prefix

The prefixed action $a \rightarrow P$ is determined on engaging in the event a and nothing else; after engaging in a it behaves like P. This is formalised as follows.

The first case is when nothing has been observed in the trace, except $tock$ events marking the passage of time: $trace(tt') \in tock^*$. Then, over this period, the event a must not be refused: $a \notin refusals(tt')$.

In the second case, an event has been observed and it must have been the a-event: the first non-$tock$ event must be a. To specify this property we need an auxiliary definition.

The idle prefix of a timed testing trace t is denoted $idleprefix(t)$ and describes the longest prefix of t containing only $tock$ events. For example, the trace

$$\langle \emptyset, tock, \{a\}, tock, b, c, \{a, c\}, tock \rangle$$

has the idle prefix $\langle \emptyset, tock, \{a\}, tock \rangle$. The idle suffix of t is the remainder of the trace once the idle prefix has been removed. In this example, the idle suffix is $\langle b, c, \{a, c\}, tock \rangle$. These definitions are formalised as follows:

Definition 4.1.7 (idleprefix and idlesuffix)

$$\begin{array}{l}
idleprefix(t) \leq t \\
trace(idleprefix(t)) \in tock^* \\
\forall\, t : TimedTrace \bullet \\
\qquad trace(u) \in tock^* \wedge trace(u) \leq trace(t) \\
\qquad\qquad \Rightarrow trace(u) \leq idleprefix(trace(t)) \\
idlesuffix(t) = t - idleprefix(t)
\end{array}$$

Continuing with our second case, the a-event must be the first non-$tock$ event in the trace: $head(trace(idlesuffix(tt')))$. The event a must not be refused during the idle prefix: $a \notin refusals(idleprefix(tt'))$.

Finally, the action will continue as P behaves: $P[tail(idlesuffix(tt'))/tt']$. Note that the use of $head$ and $tail$ are both defined, since $trace(tt') \notin tock^*$. All this is formalised as follows:

Definition 4.1.8 (Prefix)

$$
\begin{array}{l}
a \to P \;\widehat{=} \\
\qquad \mathit{TO} \left(\begin{array}{l}
a \notin \mathit{refusals}(tt') \\
\qquad \lhd\, \mathit{trace}(tt') \in \mathit{tock}^* \rhd \\
\qquad\qquad a = \mathit{head}(\mathit{trace}(\mathit{idlesuffix}(tt'))) \\
\qquad\qquad \wedge\; a \notin \mathit{refusals}(\mathit{idleprefix}(tt')) \\
\qquad\qquad \wedge\; P[\mathit{tail}(\mathit{idlesuffix}(tt'))/tt']
\end{array} \right)
\end{array}
$$

4.1.3 Internal Choice

The internal choice between P and Q is modelled simply as disjunction.

Definition 4.1.9 (Internal choice)

$$P \sqcap Q \;\widehat{=}\; P \vee Q$$

4.1.4 External Choice

In the external choice between P and Q, the two actions are run in parallel until something observable occurs: one of the actions performs a visible event or one of the actions terminates. At that point the other action is discarded and the choice is made. Clearly, the two actions must agree on how long to wait, and this is formalised as $(P \wedge Q)[\mathit{idleprefix}(tt')/tt']$. Subsequent behaviour is described by $(P \vee Q)$.

Definition 4.1.10 (External choice)

$$P \,\Box\, Q \;\widehat{=}\; (P \wedge Q)[\mathit{idleprefix}(tt')/tt'] \wedge (P \vee Q)$$

The difference between internal and external choice can be seen by comparing the two processes JW: Good idea. How about simply

$$a \to \mathit{STOP} \sqcap b \to \mathit{STOP}$$

and

$$a \to \mathit{STOP} \,\Box\, b \to \mathit{STOP}$$

The latter process cannot initially refuse an offer of a or b, but the former can refuse either. The latter is a refinement of the former.

4.1.5 Parallel Composition

The parallel composition $P \parallel_A Q$ specifies the set of events that require synchronisation between the two actions P and Q; outside this set events happen independently, without needing the participation of the other action. Parallel composition is then a form of restricted conjunction, where each action's behaviour is seen as a projection of the overall trace.

Definition 4.1.11 (Parallel composition)

$$P \parallel_A Q \;\widehat{=}\; \exists\, t, u \bullet P[t/tt'] \wedge Q[u/tt'] \wedge tt' \in t \parallel_A u$$

The definition uses a semantic operator on traces. To define this, we start by defining an intersection operator for refusal sets. Suppose that P has a refusal set X and Q has a refusal set Y. Our intersection operator $X \cap_A Y$ tells us what the refusal set will be for the parallel composition. There are three cases:

1. $X \cap A$: the set of synchronisation events refused by P.
2. $Y \cap A$: the set of synchronisation events refused by Q.
3. $X \cap Y$: the set of independent events refused by both by P and by Q.

Any subset of the union of these three sets is a refusal of the parallel composition of P and Q.

Definition 4.1.12

$$X \cap_A Y \;\widehat{=}\; \mathbb{P}((X \cap A) \cup (Y \cap A) \cup (X \cap Y))$$

Now we are ready to define our semantic operator on timed testing traces.

Definition 4.1.13 (Trace interleaving)

Let $t, u \in \mathit{timedTrace};\; a, b \in A;\; c, d \notin A;\; S, T \in \mathbb{P}\,\Sigma$

$$\begin{array}{rl}
t \parallel_A u & = u \parallel_A t \\
\langle\rangle \parallel_A \langle\rangle & = \{\langle\rangle\} \\
\langle\rangle \parallel_A \langle b\rangle \frown u & = \{\} \\
\langle\rangle \parallel_A \langle d\rangle \frown u & = \{\, \langle d\rangle \frown v \mid v \in \langle\rangle \parallel_A u \,\} \\
\langle\rangle \parallel_A \langle T, \mathit{tock}\rangle \frown u & = \{\} \\
\langle a\rangle \frown t \parallel_A \langle a\rangle \frown u & = \{\, \langle a\rangle \frown v \mid v \in t \parallel_A u \,\} \\
\langle a\rangle \frown t \parallel_A \langle b\rangle \frown u & = \{\} \\
\langle a\rangle \frown t \parallel_A \langle d\rangle \frown u & = \{\, \langle d\rangle \frown v \mid v \in \langle a\rangle \frown t \parallel_A u \,\} \\
\langle a\rangle \frown t \parallel_A \langle T, \mathit{tock}\rangle \frown u & = \{\} \\
\langle c\rangle \frown t \parallel_A \langle d\rangle \frown u & = \{\, \langle c\rangle \frown v \mid v \in t \parallel_A \langle d\rangle \frown u \,\} \cup \\
 & \quad\; \{\, \langle d\rangle \frown v \mid v \in \langle c\rangle \frown t \parallel_A u \,\} \\
\langle c\rangle \frown t \parallel_A \langle T, \mathit{tock}\rangle \frown u & = \{\, \langle c\rangle \frown v \mid v \in t \parallel_A \langle T, \mathit{tock}\rangle \frown u \,\} \\
\langle S, \mathit{tock}\rangle \frown t \parallel_A \langle T, \mathit{tock}\rangle \frown u & = \{\, \langle U, \mathit{tock}\rangle \frown v \mid U = S \cap_A T \wedge v \in t \parallel_A u \,\}
\end{array}$$

Note that traces must always agree on *tock* events: *tock* is implicitly assumed to be in A. Further, the traces formed by merging a pair of timed testing traces are maximal: none is a prefix of any other.

Lemma 4.1.4 (Minimality of trace composition)

$$r \in t \parallel_A u \Rightarrow \neg\, \exists\, s, w \bullet ((s \prec t \vee w \prec u) \wedge r \in s \parallel_A w)$$

Proof: *By induction on the cases of the trace interleaving definition.*

4.1.6 Hiding

The hiding operator provides a way to abstract processes by internalising some events, thus making them unobservable by the environment. An assumption of maximal progress requires that no time may elapse whilst hidden events are on offer: hidden events happen as soon as they become available. Once more, the definition is given using semantic functions:

Definition 4.1.14 (Hiding)

$$P \setminus A \;\widehat{=}\; \exists\, t \bullet P[t/tt'] \land A \;\mathsf{urgent}\; t \land (tt' = t \setminus A)$$

The assumption of maximal progress is modelled by considering only the A-urgent traces of P: the traces where every event in A is refused before a *tock* event. These traces represent states in which no further internal progress is possible using events from the set A: all possible occurrences of those events must already have happened internally.

Definition 4.1.15 (Urgency)

$$A \;\mathsf{urgent}\; t \;\widehat{=}\; \forall\, s, X \bullet s \frown \langle X, tock \rangle \le t \Rightarrow A \subseteq X$$

The semantic hiding operator is then defined inductively:

Definition 4.1.16 (Trace hiding)

$$\begin{array}{rcl}
\multicolumn{3}{l}{\mathit{For}\ S \subseteq \Sigma;\ a \in A;\ b \notin A} \\
\langle\rangle \setminus A & = & \langle\rangle \\
(\langle S, tock \rangle \frown tt) \setminus A & = & \langle S \setminus A, tock \rangle \frown (tt \setminus A) \\
(\langle a \rangle \frown tt) \setminus A & = & tt \setminus A \\
(\langle b \rangle \frown tt) \setminus A & = & \langle b \rangle \frown (tt \setminus A)
\end{array}$$

4.1.7 Timeout

The timeout process $P \overset{n}{\rhd} Q$ initially offers to act like P for n time units; however, if P has failed to communicate any visible event within this time period, then the process silently changes to behave like Q. This operator is strict in the sense of Lowe & Ouaknine: events of P cannot be performed by $P \overset{n}{\rhd} Q$ after the nth tock. A non-strict operator would permit the events of P to be available unstably after the nth, but before the $n + 1$th, *tock*. This non-strict operator can be derived from other operators in the language, but the strict one cannot.

The operator is defined in two cases. In one case, at least n time units have passed without a visible event: $tock^n \le trace(tt')$. To account for this behaviour, P must have been able to wait for this period without engaging in any external events; the subsequent trace is then a behaviour of Q. The other case is the complement: fewer than n time units have passed, say m, without a visible event: $\neg\, tock^n \le trace(tt')$. Now, if the idle suffix is empty, then it must be possible for P to wait for m time units. On the other hand, if the idlesuffix is non-empty, then it must also have been possible for P to wait m time units and then perform the idle suffix. Either way, the trace is a behaviour of P.

Definition 4.1.17 *Timeout*

$$
\begin{array}{l}
P \overset{n}{\triangleright} Q \;\widehat{=} \\
\quad (\exists\, u \bullet u \leq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt']) \\
\qquad \lhd\, tock^n \leq trace(tt') \,\rhd \\
\qquad\quad P
\end{array}
$$

4.1.8 Recursion

Recursion is defined as the least fixed-point, as usual.

Definition 4.1.18 (Recursion)

$$\mu\, F = \sqcap\{\, P \mid F(P) \sqsubseteq P \,\}$$

4.2 Lowe & Ouaknine's Axioms

Our semantic domain is inspired by that of Lowe & Ouaknine. They start with five axioms, some of which we can consider as theorems of our definitions.

4.2.1 Well Foundedness

The first axiom states that the empty trace is a possible behaviour of every process.

Definition 4.2.1 (T1: Well foundedness)

$$\textbf{\textit{T1}}(P) = P[\langle\rangle/tt']$$

Theorem 4.2.1 (Well foundedness) *Every CML operator preserves **T1**-healthiness.*

Proof 4.2.1 *See Appendix.*

4.2.2 Prefix Closure

The second axiom states that the traces of every process are prefix closed: if tt' is a trace of P, then so is every prefix of P. This ensures that the history of a system evolves in a smooth way, event by event.

Definition 4.2.2 (T2: Prefix closure)

$$\textbf{\textit{T2}} \qquad [\, P \land t \preceq tt' \Rightarrow P[t/tt']\,]$$

Theorem 4.2.2 (Prefix closure) *Every CML operator preserves **T2**-healthiness.*

Proof 4.2.2 *See Appendix.*

4.2.3 Refusals

An event in the process alphabet can always be either performed or refused. Informally, the axiom states that if at any point in an observation, a process can refuse the set A and cannot perform the event a, then it can refuse a as well as A.

Definition 4.2.3 (T3: Refusals)

$$\mathbf{T3}(P) = P \wedge (P[tt' \frown \langle A, tock\rangle / tt'] \wedge \neg\, P[tt' \frown \langle a\rangle / tt'] \Rightarrow P[tt' \frown \langle A \cup \{a\}, tock\rangle / tt'])$$

Theorem 4.2.3 (Refusals) *Every CML operator preserves **T3**-healthiness.*

4.2.4 Timelock Freedom

A process can always allow time to pass.

Definition 4.2.4 (T4: Timelock freedom)

$$P \Rightarrow P[tt' \frown \langle \emptyset, tock\rangle / tt']$$

Theorem 4.2.4 (Timelock freedom) *Every CML operator preserves **T4**-healthiness.*

4.2.5 Zeno Freedom

Lowe & Ouaknine have a bounded-speed condition as an axiom for their processes: there is a bound n on the number of events that can be performed in the first k time units. Note $\#s$ is the length of the sequence s.

Definition 4.2.5 (T5: Zeno freedom)

$$\mathbf{T5}(P) = P \wedge (\#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(tt')) \leq n)$$

We say that a recursive process is time-guarded if it cannot recurse without time passing. The Zeno-freedom axiom is satisfied by *CML* processes made up from *CML* operators that contain only time-guarded recursions.

Theorem 4.2.5 (Prefix closure) *Suppose that P is a time-guarded process, then for every k there is an n, such that P is **T5**-healthy.*

Proof 4.2.3 *See Appendix.*

4.3 Timed Imperative Sequential Reactive Processes

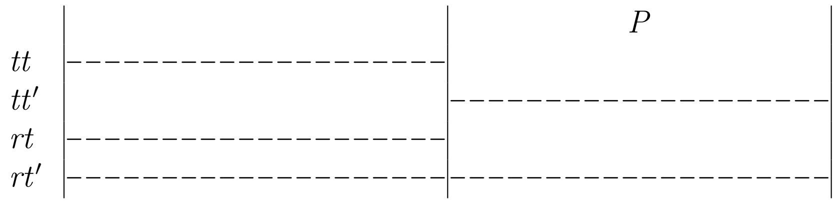
In this section, we extend our treatment of *CML* by including sequential composition and imperative state. We introduce three new observations:

- ok, ok': These are the observation variables from designs [14, Chapter 3]. The observation ok describes the situation in which a process has been started in a stable state, whilst ok' describes the situation in which a process has reached a stable state.

- $wait, wait'$: These are the observation variables from reactive processes [14, Chapter 3]. The observation $wait$ describes the situation in which a process occupies a waiting state of its sequential predecessor, whilst $wait'$ describes the situation in which the process has reached a waiting state. The combination of ok and $wait$ and their dashed counterparts allow sequential combination to be defined as relational composition.

- tt: In Section 4.1, there is a single observation tt' of the trace of a process. Having added sequential composition, we need to make our relations homogeneous, so we add the before version of tt': the value of the trace before the behaviour of the current process.

- rt, rt': These are the observations of the trace of the previous process (rt) and the current process (rt'). The following diagram describes the relationship between the four trace variables, where the dotted lines represent the traces in process P's behaviour:



The behaviour of P is represented by the trace tt'. The behaviour of the predecessors of P is represented by the trace rt, which in this diagram is equal to the trace tt (although, we shall not need to refer to tt again). Finally, the trace of the entire system, including the behaviour of P and its predecessors, is given by rt'.

4.3.1 Healthiness Conditions

There are six new healthiness conditions. The first requirement is that both rt and rt' are properly structured as timed traces.

Definition 4.3.1 (*RT0*)

$$\textbf{\textit{RT0}}(P) = \textbf{\textit{T0}}(P) \land (rt \in timedTrace) \land (rt' \in timedTrace)$$

Next, there should be the relationship we indicated in the diagram between rt, tt, and rt'.

Definition 4.3.2 (*RT1*)

$$\textbf{\textit{RT1}}(P) = P \land (rt' = rt \frown tt')$$

Our next healthiness condition is similar to ***R2*** in Hoare & He's theory of reactive processes (see [14, p.195]). It controls the use of the trace variable to make sure that P is not sensitive to the behaviour of its predecessors. For example, it cannot depend on certain events already having taken place, or for a particular amount of time having elapsed under its predecessor's control.

Definition 4.3.3 (*RT2*)

$$\mathbf{RT2}(P) = P[\langle\rangle, tt'/rt, rt']$$

Our fourth healthiness condition is similar to ***R3*** in the theory of reactive processes (see [14, p.196]). Reactive processes visit a series of states after starting their execution. These states are either stable final states, where $ok' \land \neg\, wait'$ holds, or they are intermediate states where $ok' \land wait'$ holds, and in which the process is waiting for interaction with its environment. In the sequence $P\ ;\ Q$, if P has reached an intermediate state, then we have to describe what Q's behaviour will be. Of course, since P is waiting, Q will do nothing at all: it will behave like a right identity for the sequential operator. This requirement is captured by the following healthiness condition.

Definition 4.3.4 (*RT3*)

$$\mathbf{RT3}(P) = (I\!I_{\mathbf{RT}} \lhd wait \rhd P)$$

where $I\!I_{\mathbf{RT}} = \mathbf{RT}(\mathbf{true} \vdash I\!I)$

where $\alpha I\!I$ is $\{tt, tt', rt, rt', v, v'\}$, where v and v' are the initial and final observations of the program variables. Our fifth healthiness condition corresponds to ***CSP1*** in Hoare & He's theory of CSP (see [14, p.208]). If P's predecessor is in an unstable state, then P will not be started and we have $\neg\, ok$. What contribution will P now make to the divergent behaviour of its predecessor? It is allowed to behave almost arbitrarily: it cannot destroy the structure of the testing traces, nor can it interfere with the relationship that binds them together.

Definition 4.3.5 (*RT4*)

$$\mathbf{RT4}(P) = \mathbf{RT01} \circ \mathbf{TT0}(\neg\, ok) \lor P$$

where $\mathbf{RT01} = \mathbf{RT0} \circ \mathbf{RT1}$.

Finally, P must be monotonic in the value of the ok' variable, just like a design: P cannot demand instability and nontermination.

Definition 4.3.6 (*RT5*)

$$\mathbf{RT5}(P) = P\ ;\ J$$

where $J = (ok \Rightarrow ok') \land (rt' = rt) \land (tt' = tt) \land (v' = v)$

Notice that ***RT4*** and ***RT5*** are the timed reactive versions of ***H1*** and ***H2***, respectively.

Lemma 4.3.1 (*RT* functions are commuting monotonic idempotents)

1. ***RT0–RT5*** *are all monotonic idempotents.*

2. ***RT0–RT5*** *all commute.*

Definition 4.3.7 (*RT*)

$$\boldsymbol{RT} \mathrel{\widehat{=}} \boldsymbol{RT0} \circ \boldsymbol{RT1} \circ \boldsymbol{RT2} \circ \boldsymbol{RT3} \circ \boldsymbol{RT4} \circ \boldsymbol{RT5}$$

We can now proceed to redefine our process combination and to add a few more. We define processes as timed reactive designs in the style of *Circus* (for an introduction to this style, see [6]).

4.3.2 Sequential Composition

Sequential composition was deliberately not mentioned in Section 4.1 so that we could introduce other operators in a simple fashion. It is simply relational composition, given our healthiness conditions.

Definition 4.3.8 (Sequential composition)

$$P \;;_{\boldsymbol{RT}}\; Q = P \;;\; Q$$

4.3.3 Assignment

For the assignment $x := e$, we make the simplifying assumption that the expression e is well defined (we address this assumption in Chapter 6). The assignment takes place immediately and the process then terminates. This process has precondition $true$ and a postcondition (which guarantees stability) that it has terminated $(\neg\, wait)$ in zero time without any visible events $(trace(tt') = \langle\rangle)$, but having completed the assignment $(v' = e)$. Notice that the use of the $trace$ function allows the refusal sets in tt' to be arbitrary. This design is then made healthy with ***RT0*** $\circ$ ***RT1*** $\circ$ ***RT3***, which we abbreviate to ***RT013***. Actually, it is by construction ***RT2***-healthy (it does not constrain rt inappropriately, and we therefore do not need to enforce it) and ***RT4*** and ***RT5***-healthy (it is a reactive design).

Definition 4.3.9 (Assignment)

$$(x :=_{\boldsymbol{RT}} e) = \boldsymbol{RT013}(\boldsymbol{true} \vdash (tt' = \langle\rangle) \wedge \neg\, wait' \wedge (v' = e))$$

4.3.4 STOP

Our old definition of $STOP$ is almost what we need, but we need to make it aware of our new observational variables and turn it into a design. First, it is a design with precondition true; second, it is perpetually waiting; third, it is ***RT013***-healthy.

Definition 4.3.10 (Deadlock)

$$STOP_{\boldsymbol{RT}} = \boldsymbol{RT013}(\boldsymbol{true} \vdash STOP_{\boldsymbol{TT}} \wedge wait')$$

4.3.5 SKIP

We define *SKIP* to be the vacuous assignment.

Definition 4.3.11 (Termination)

$$SKIP_{\mathbf{RT}} = (v :=_{\mathbf{RT}} v)$$

4.3.6 Prefix

We begin with a notational shorthand introduced in [6]:

Definition 4.3.12

$$P_b^c = P[b, c/wait, ok']$$

where t b and c range over the boolean values $\{t, f\}$.

An event-prefixed process $a \to P$ is able to diverge if P diverges, but we know that this can happen only after an a event. So the precondition for the process is $\neg\ P_f^f[\langle a \rangle \frown tt'/tt]$: this is P's divergent behaviour. So, $P_f^f[\langle a \rangle \frown tt'/tt]$ is P's divergent behaviour on any trace starting with the event a. The precondition is the negation of this. The postcondition is given by the testing traces prefix operator.

Definition 4.3.13 (Prefix)

$$a \to_{\mathbf{RT}} P = \mathbf{RT013}(\neg\ P_f^t[\langle a \rangle \frown tt'/tt] \vdash a \to_{\mathbf{TT}} P_f^f)$$

4.3.7 Internal Choice

Internal choice is simply disjunction, as usual.

Definition 4.3.14 (Internal choice)

$$P \sqcap_{\mathbf{RT}} Q = P \vee Q$$

4.3.8 External Choice

External choice is, of course, more involved than internal choice. The process $P \square_{\mathbf{RT}} Q$ diverges whenever either of its operands diverges (it is strict). Its postcondition is simply the external choice operator of testing traces.

Definition 4.3.15 (External choice)

$$P \square_{\mathbf{RT}} Q = \mathbf{RT013}(\neg\ (P \vee Q)_f^f \vdash P_f^t \square_{\mathbf{TT}} Q_f^t)$$

4.3.9 Timeout

The precondition for the timeout process $P \overset{n}{\triangleright}_{\mathbf{RT}} Q$ comes in two parts. The first case deals with the case where the process has waited up to n time units without any visible event. We can see that P's precondition will fail to hold on any trace tt' that we can divide up into two ***RT***-healthy portions, the first of which is of duration not exceeding n time units, at the end of which P^f_f holds:

$$((trace(tt') \leq tock^n) \Rightarrow P^f_f) \mathbin{;} \mathbf{RT01}(\mathbf{true})$$

Obviously, we do not want this situation. The second case is where P's precondition held successfully over an interval of n time units, but then P's postcondition fails to establish the precondition for Q:

$$(P^t_f \wedge (trace(tt') = tock^n)) \mathbf{wp} \neg\, Q^f_f$$

where ***wp*** is the weakest precondition operator [14], see Section 3.5. The postcondition for the timeout process is very simple: it is the testing traces postcondition. All this is summarised in the following definition.

Definition 4.3.16 (Timeout)

$$P \overset{n}{\triangleright}_{\mathbf{RT}} Q = \mathbf{RT013} \begin{pmatrix} \neg\, (((trace(tt') \leq tock^n) \Rightarrow P^f_f) \mathbin{;} \mathbf{RT01}(\mathbf{true})) \\ \wedge\, ((P^t_f \wedge (trace(tt') = tock^n)) \mathbf{wp} \neg\, Q^f_f) \\ \vdash \\ P^t_f \overset{n}{\triangleright}_{\mathbf{TT}} Q^t_f \end{pmatrix}$$

4.3.10 Parallel Composition

We call two timed reactive designs *disjoint* if they share no programming variables; they are allowed, of course, to share observational variables. This rules out shared variable parallelism.

The precondition of the parallel composition of P and Q is the conjunction of the preconditions of P and Q. The postcondition merges the intermediate or final states of the two processes. It does this by running the two postconditions in parallel using the testing traces parallel operator. Since the program variables are partitioned, the equation $(v' = v)$ takes care of the appropriate merging of these programming variables, and we need worry only about merging the observational variables. The testing traces parallel operator has already taken care of the tt' trace, which then determines the value of rt'. The parallel composition reaches a stable state providing the two operands both reach a stable state, and this is taken care of by taking the conjunction of their individual results for their ok' variables. Similarly, the composition is in a waiting state if either of the processes end up in a waiting state. This is taken care of by taking the disjunction of their waiting states.

Definition 4.3.17 (Parallel composition) *for disjoint P and Q*

$$P \|_{A_{\textbf{\textit{RT}}}} Q = \textbf{\textit{RT013}} \left(\begin{array}{l} \neg\,(P \vee Q)^f_f \\ \vdash \\ \exists\, ok0', ok1', wait0', wait1' \bullet \\ \qquad (P^t_f[ok0', wait0'/ok', wait'] \|_{A_{\textbf{\textit{TT}}}} Q^t_f[ok1', wait1'/ok', wait']) \,; \\ \qquad (rt' = rt) \wedge (v' = v) \\ \qquad \wedge\, (ok' = ok0 \wedge ok1) \wedge (wait' = wait0 \vee wait1) \end{array} \right)$$

4.3.11 Hiding

There are two sources of divergence arising from hiding. First, a process $P \setminus A$ may diverge because P itself diverges. Second, it may be that hiding an unbounded sequence of events causes the hiding process to diverge. This is captured by the precondition $\neg\,(P^f_f \setminus_{\textbf{\textit{TT}}} A)$. The postcondition is formed from using the testing traces hiding operator.

Definition 4.3.18 (Hiding)

$$P \setminus_{\textbf{\textit{RT}}} A = \textbf{\textit{RT013}}(\neg\,(P^f_f \setminus_{\textbf{\textit{TT}}} A) \vdash P^t_f \setminus_{\textbf{\textit{TT}}} A)$$

4.3.12 Recursion

If F is ***RT***-healthy, then the least fixed point of F is just the ***RT***-healthy least fixed point of the ***TT***-healthy fixed-point of F.

Definition 4.3.19

$$(\mu\, X \bullet F(X)) = \textbf{\textit{RT}}(\mu\, X \bullet F(X))$$

Chapter 5

CML-UTP Operator Correspondences

In this chapter, we describe the relationship between the notation defined for *CML* and that used to express its denotational semantics. The treatment of types and values in the UTP semantics is explained, and tables of correspondences are presented to relate the other important constructs.

5.1 Introduction

The Compass Modelling Language (*CML*) is a heterogeneous language consisting of constructs from VDM and CSP with time extensions, which are additionally structured according to concepts drawn from object-oriented programming. The draft syntax is presented in [34]; the version of the language presented in this document is dubbed *CML1*.

The denotational semantics for *CML* unifies the component languages in the framework provided by UTP (Unifying Theories of Programming) [14]. It defines a core set of operators required to give meaning to the rest of the language. It is expressed in the style of UTP and based on notation associated with UTP. The rationale for this is to support proof, i.e., by direct application of the theory and algebraic laws of UTP, and to allow straightforward communication within heterogeneous languages research communities.

The two notations are quite different in style and genericity, and the meaning of many of the operators of *CML* is given by definition in terms of more fundamental operators. Thus the notations appear very different. The aim of this Chapter is to explain those differences and provide a basic guide for reconciling the dissimilarities. In essence this amounts to (i) explaining which aspects of the semantic framework are generic and describing how they will be instantiated; (ii) identifying where the languages only differ in syntax; and (iii) defining how non-primitive operators of *CML* are defined in terms of the primitives.

The concepts are scoped according to priority. This is a notational guide; it is not intended to give a complete definition of the language, as this will be presented elsewhere. However, we do aim to cover the most important elements of the language. In addition, we do not cover the object-oriented features of the language.

The Chapter is structured following the syntax of *CML* presented in [34]. Section 5.2 gives an overview of the UTP operators referred to during the discussion. Sections 5.3, 5.4, and 5.5 address expressions, specifications, and actions respectively. The section on actions is further decomposed to address flow of control, process operators, and time operators. Section 5.6 briefly discusses the global structure of *CML*, and a summary is given in Section 5.7.

5.2 UTP Notation

This section gives an overview of the UTP notation used in the remainder of the Chapter. It is structured in three parts: (i) common UTP notation; (ii) additional UTP notation defined and redefined by *CML*'s denotational semantics; and (iii) specification notation as used in this text.

5.2.1 Common UTP Operators

There are several UTP operators are that have the same definition in the majority of UTP semantic treatments. These are used directly in the denotational semantics with their usual meaning. They are summarised in table 5.1.

Operator Usage	Description
$P \; ; \; Q$	Sequential composition. Defined as relational composition of alphabetised relations.
$P \sqcap Q$	nondeterministic choice. Defined as disjunction of alphabetised relations.
$\bigsqcap_i P(i)$	Indexed nondeterministic choice. Defined as indexed disjunction of alphabetised relations.
var x	Variable scope begin
end x	Variable scope end
$P \lhd b \rhd Q$	If b then P else Q $((b \wedge P) \vee \neg\,(b \wedge Q))$

Table 5.1: Common UTP operators

5.2.2 Denotational Semantics UTP Operators

Certain operators appear frequently in UTP semantic treatments, but require redefinition for the specific program lattice and healthiness conditions in place for each. This is also the case for *CML*'s semantics. Table 5.2 summarises these operators.

Additionally, the denotational semantics defines other core operators of *CML*, which are used to define the meaning of operator variants. These are summarised in Table 5.3

Operator Usage	Description
$\bot$	The bottom of the program lattice (the least refined program)
$\top$	The top of the program lattice (the most refined program). $\top$ is unimplementable.
$I\!I$	Skip. Terminate immediately. (See Section 3.3.)
$x := e$	Assignment. Set x equal to e, leave all other variables unchanged and terminate immediately.
$\mu X \bullet P(X)$	Least fixed-point on the lattice of programs (recursion).

Table 5.2: Redefined UTP operators

Operator Usage	Description
$STOP$	The deadlocked process (may advance in time).
$CHAOS$	Synonym for $\bot$
$a \rightarrow P$	Communication Prefix
$P \,\square\, Q$	External Choice
$P \overset{n}{\triangleright} Q$	Timeout. Defer to Q if P has failed to produce an observable event within n.
$P \llbracket s_1 \mid cs \mid s_2 \rrbracket Q$	Generalised Parallel. P operates on state s_1, Q operates on state s_2. They synchronise on events in cs and interleave on all other events.
$P \setminus cs$	Hiding. The events in cs are hidden from the observer.

Table 5.3: Defined UTP operators

5.2.3 Design notation

In UTP the notation $P \vdash Q$ is used to express designs. A design corresponds to the familiar precondition and postcondition (total correctness) specification style of VDM, B, etc. It means that if a program is started in a state satisfying P it is guaranteed to terminate and result in a state satisfying Q. The meaning of $P \vdash Q$ is given in terms of the auxiliary variable ok, which represents termination.

In non-sequential programming paradigms, additional auxiliary variables are required, e.g., tr, $wait$, and ref, to capture the semantics of events and communication. This is the case in the *CML* semantics, which is based on rt and $wait$ as well as ok. For theories with additional variables (additional) healthiness conditions are required to express the constraints those variables must adhere to. The healthiness conditions are idempotent and are used dually in UTP as functions to impose healthiness conditions on alphabetised relations which are generally weaker than the healthy relations. This mechanism is used extensively for transforming a concept in a more primitive programming theory into an equivalent concept in a related theory.

In *CML* the healthiness function ***RT***(_) is applied to alphabetised relations to make them healthy. One of the ways this is particularly useful is to transform specifications in sequential programming languages (cf. $P \vdash Q$) into specifications in the *CML* timed-traces model. ***RT***$(P \vdash Q)$ represents the program which is started in a state satisfying P terminates and results in a state satisfying Q, otherwise it diverges. Thus it provides a way for interpreting sequential specifications within the *CML* semantic landscape.

In what follows we will use the notation $P \vdash_{\textbf{\textit{RT}}} Q$ as a synonym for $\textbf{\textit{RT}}(P \vdash Q)$. This is summarised in table 5.4.

Operator Usage	Description
$P \vdash Q$	Total correctness specification with precondition P and postcondition Q
$P \vdash_{\textbf{\textit{RT}}} Q$	Specification $P \vdash Q$ interpreted as an operation in the timed-traces model.

Table 5.4: Design Notation

5.3 Expressions

Expressions are the basic building blocks of the language and are covered in the *CML0* language definition [34, Section 17]. We distinguish between general (i.e., non-Boolean) and Boolean expressions in our discussion.

5.3.1 General

UTP is essentially generic with respect to expression notation. For the *CML* semantics, some expression operators are required to manipulate the auxiliary observable variables needed to capture the meaning of the the language. However, expressions tend to be limited to Boolean operations on Boolean variables, as well as operations on sequences (e.g., *head*, *append*) to manipulate traces. These are expressed using the notation preferred in the UTP book.

UTP focuses on giving meaning to the higher-level constructs of a language, such as program operators, treating expressions as a shallow embedding. The meaning of an expression $\mathtt{e}$ in its denotational setting is also written e. UTP distinguishes the two by a change of font. The expression form in UTP is essentially a place-holder for whatever expression notation, and corresponding meaning, one requires. The expression notation is used, for example, to define operations on the program variables through assignment, branching etc.

The *CML* semantics under development exploits this genericity. It does not need to refer explicitly to expression notation in order to give meaning to the higher level operators of *CML* (process composition, flow of control etc) that are based on it. However, in practice, the expression syntax can be thought of UTP augmented with that of *CML*, which in turn hails from VDM. Intuitively, the expressions of *CML* are interpreted according to the VDM semantics [18, 23], giving rise to the alphabetised relations (satisfying models) required by the *CML* semantics (see Section 5.3.2 also).

5.3.2 Boolean Valued Expressions

UTP employs propositional connectives and quantifiers extensively. The notation, as used in the *CML* semantics, uses the Boolean operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\iff$, which can be viewed as synonyms for *CML*'s **`not`**, **`and`**, **`or`**, => and <=>. The quantifiers $\forall\, v \bullet P$ and $\exists\, v \bullet P$ are also used, and are equivalent to the use of *CML*'s **`forall`** and **`exists`**. Additionally UTP uses square brackets $[\,P\,]$ to express universal quantification over the free variables of P. As with *CML* predicates can be used as expressions, e.g., to define the value of Boolean variables.

UTP is based on alphabetised relations, which are predicates along with an associated alphabet of variables. The predicate is *characteristic* of the relation it defines. The semantic link between *CML* (VDM) predicates, and alphabetised relations, can be viewed in terms of the VDM denotational semantic function for expressions. For each expression the semantic function yields another function from Environments to Values. The alphabetised relation corresponding to a predicate P is essentially the set of environments in which P is evaluated to *true*, restricted to the alphabet of interest.

UTP is again generic with respect to n-ary relations yielding Boolean values, employing only what is needed to define the programming paradigm in question. Correspondingly, the *CML* semantics is generic in this respect, using very few relations (e.g., $\leq$ for sequence prefix relation). Relations provide a semantic bridge between non-Boolean expressions and predicates, and those of *CML* can again be understood in terms of the VDM denotational function for expressions.

5.4 Specification

In UTP specifications are often expressed in terms of designs. A design represents the familiar precondition-postcondition pair within the total correctness framework. If a program starts within a state satisfying the precondition, it will terminate and do so in a state satisfying the postcondition. The usual notation for a design is $P \vdash Q$, in which P is the precondition and Q is the postcondition.

The notion of design needs to be reinterpreted within the timed-traces semantic framework via the healthiness function ***RT***, as described in Section 5.2.3 above. The assignment $x := e$ is based on the redefined assignment operator for timed traces. In practice this is based on the redefinition of design for timed traces ($\vdash_{\boldsymbol{RT}}$) introduced in Table 5.2. Table 5.5 provides the correspondence between specification constructs in *CML* and UTP, based on ***RT***.

5.5 Actions

The principal syntactic structure by which operations and processes are defined in *CML* is the process paragraph. The most important element is the *action* syntax, which defines the body of operations and processes. It includes flow-of-control operators, process operators and time operators. These are discussed in detail in the following subsections.

Syntax	Equivalent form	Notes	***CML0*** **Reference**
pre P **post** Q	$P \vdash_{RT} Q$	Specification Statement	Section 13 and 14 Section 15.4
post Q	***true*** $\vdash_{RT} Q$	As above, no precondition supplied	As above
x := e	$x := e$	Assignment (similarly for multiple assignment)	Section 15.2

Table 5.5: Specification and Assignment

5.5.1 Control Statements

Control statements [34, Table 7] comprise guarded commands, conditionals and loop statements. The following describes how each CML construct is interpreted in UTP.

Guarded Commands

The first set of operators considered are Dijkstra's guarded commands, which are covered in [34, Section 15.7]. The conditional statement diverges if no guard holds, else it behaves nondeterministically as one of the actions whose guard does hold. The repetitive statement repeatedly behaves as one of the actions whose guard holds until none of the guards hold, at which point it terminates. The relationship between syntactic and semantic notations, defining the meaning of these statements, is given in Table 5.6.

<table>
<tr><th>Syntax</th><th>Equivalent form</th><th>Notes</th></tr>
<tr><td>if e_1 -> a_1
...
[] e_n -> a_n
end</td><td>(e_1$^{\top}$; a_1)
|~| ...
|~| (e_n$^{\top}$; a_n)
<| e_1 or ... or e_n |> $\bot$</td><td>Conditional statement</td></tr>
<tr><td>do e_1 -> a_1
...
[] e_n -> a_n
end</td><td>mu X @
((((e_1$^{\top}$; a_1)
|~| ...
|~| (e_n$^{\top}$; a_n)) ; X)
<| e_1 or ... or e_n |> $I\!I$)</td><td>Repetitive statement</td></tr>
</table>

Table 5.6: Guarded Commands

Conditionals

Next are the conditional and cases statements, which feature in [34, Section 15.8]. The definitions in the semantic notation for each construct are given in Table 5.7. We provide only a basic example of the cases statement to avoid the semantic treatment of patterns and pattern lists.

Syntax	Semantic equivalent	Notes
if e -> a	$a \lhd e \rhd \mathit{II}$	conditional statement (no elseif/else)
if e -> a_1 **else** a_2	$a_1 \lhd e \rhd a_2$	conditional statement with else (no elseif)
if e_1 -> a_1 **elseif** e_2 -> a_2 **else** a_3	$a_1 \lhd e_1 \rhd$ $(a_2 \lhd e_2 \rhd a_3)$	conditional statement with else/elseif Similarly for multiple elseif
cases e_1: (e_2) -> a_1 (e_3) -> a_2 ... (e_n+1) -> a_n **others**-> a_n+1 end	$a_1 \lhd (e_1 = e_2) \rhd$ $(a_2 \lhd (e_1 = e_3) \rhd$ $(\ldots$ $(a_n \lhd e_1 = e_{n+1} \rhd a_{n+1}\ ) \cdots)\ )))$	Cases statement

Table 5.7: Conditionals

Loops

The final set of control statements concern loops, which are presented in [34, Section 15.9]. These comprise the sequence for-loop, the set for-loop, the index for-loop, and the while loop. Their definitions in the semantic notation are given in Table 5.8.

Syntax	Semantic equivalent	Notes
for x **in** s **do** a	***var*** $v := s;$ ***var*** $x;$ $\mu X \bullet (x, v := \mathit{head}(s), \mathit{tail}(s);\ a;\ X)$ $\lhd v \neq \langle\rangle \rhd \mathit{II}$ ***end*** v, x	Sequence for-loop v is a fresh variable
while e **do** a	$\mu X \bullet ((a;\ X) \lhd e \rhd \mathit{II})$	While loop

Table 5.8: Loops

5.5.2 Process Operators

The process operators in the action syntax hail primarily from CSP. Many of the operators are defined directly in the denotational semantics and are therefore identical in notation. Other operators are synonyms for semantic operators, (e.g., existing UTP operators). Several operators—principally the parallel operators—are defined by application of more primitive operators, which are expressed directly in the denotational semantics.

The following two subsections describe first the identical operators and synonyms and then the parallel operator. The majority of process operators are addressed, however several syntactic constructions are omitted from the discussion:

- Parametrised and instantiated actions—these express higher-order functions whose semantics are deferred until the introduction of object-oriented extensions (see 8).
- Replicated actions—these are indexed versions of the regular binary operators, and are generally expressed either in terms of UTP's own indexed operators or inductively. An example of a replicated action (external choice) is provided at the end of the section.
- Block statements—these introduce variables and their scopes, optionally constraining their initial value. Block statements are omitted since their syntax is not yet stable, and they have not been addressed explicitly in the current denotational semantics. However, UTP has variable block constructs ***var*** x and ***end*** x, which should provide an adequate basis for the block semantics.

Finally, timed operators are dealt with in Section 5.5.3.

Identities, synonyms and chaos

The operators shown in Table 5.9 are identical in the semantics notation. All but "`;`" are defined explicitly as part of the semantics; the ";" operator is UTP's usual sequential composition operator. All the operators are taken from the action syntax at the beginning of [34, Section 15], in the definition of ***CML0***. Several further operators are synonyms

Operator	Description	Operator	Description
`->`	Prefixing	`&`	Guarded Action
`[]`	External Choice	`|~|`	Internal Choice
`\`	Hiding	`;`	Sequential Composition

Table 5.9: Identical Operators

for semantic notation. These are given in Table 5.10. Finally, the definition of `Chaos` in terms of the semantic notation is: $Chaos = \textbf{\textit{true}} \vdash_{RT} \textbf{\textit{true}}$.

Parallel operators

The parallel operators are shown in [34, Table 4, Section 15]. There are eight variants in all. The denotational semantics, by contrast, only defines the semantics for a single

Operator (Syntactic)	Operator (Semantic)	Description	Operator (Syntactic)	Operator (Semantic)	Description
`Skip`	$I\!I$	Terminate	`Stop`	$STOP$	Deadlock
`Div`	$\bot$	Divergence	**`mu`**	μ	Recursion

Table 5.10: Synonymous Operators

general operator—this is since the meaning of each of the other seven variants can be expressed in terms of this one operator.

The general operator ($P \|_{A_{RT}} Q$) is defined semantically in Section 4.3.10. Its syntactic synonym is called generalised parallel and has the form: `A [| ns_1 | cs | ns_2 |] B`. It behaves as `A` and `B` executed in parallel and synchronising on the set of channels in `cs`. `A` (resp., `B`) can modify only the state components in `ns_1` (resp., `ns_2`).

If (the alphabet of A) $\alpha(A) = \{ns\_1, wait, ok, tt\}$ and $\alpha(B) = \{ns\_2, wait, ok, tt\}$, then `A [| ns_1 | cs | ns_2 |] B` is defined as $A \|_{cs_{RT}} B$.

Following this, the definitions of the first five derived operators are straightforward, and shown in Table 5.11. The definitions of the remaining two operators require a slightly

Syntax	Definition	Description
`A [| ns_1 |` cs `| ns_2 |] B`	A `[| ns_1 |` cs `| ns_2 |]` B	Generalised Parallel (**identical**)
`A [| cs |] B`	A `[|` $\{\}$ `|` cs `|` $\{\}$ `|]` B	Generalised Parallel (no state) Empty states
`A [|| ns_1	ns_2 ||]` B	A `[| ns_1 |` $\{\|\}$ `| ns_2 |]` B	Interleaving Empty channel set
`A` **`inter`** `B`	A `[|` $\{\}$ `|` $\{\|\}$ `|` $\{\}$ `|]` B	Interleaving (no state) Empty channel set and states
`A [| ns_1	ns_2 |] B`	A `[| ns_1 |` Σ `| ns_2 |]` B	Synchronous Parallel All channels (Σ)
`A || B`	A `[|` $\{\}$ `|` Σ `|` $\{\}$ `|]` B	Synchronous Parallel (no state) All channels, no empty states

Table 5.11: Variant Parallel Operators

more elaborate definition. We define the following construct to restrict the actions of a process to a set of actions: ***Res***$(A, ns1, ns2, X)$ = A `[|` $ns1$ `|` $\Sigma \backslash X$ `|` $ns2$ `|]` $STOP$. Σ in the definition represents the set of all channels. Given this definition the remaining two, alphabetised, parallel operators are presented in Table 5.12.

Syntax	Definition	Description
`A [| ns_1 |` X `||` Y `| ns_2 |] B`	$\mathbf{Res}(A, ns1, ns2, X)$ `[| ns_1 |` $X \cap Y$ `| ns_2 |]` $\mathbf{Res}(B, ns1, ns2, Y)$	Alphabetised Parallel
`A[`X `||` Y`] B`	$\mathbf{Res}(A, \{\}, \{\}, X)$ `[|` $\{\}$ `|` $X \cap Y$ `|` $\{\}$ `|]` $\mathbf{Res}(B, \{\}, \{\}, Y)$	Alphabetised Parallel (no state)

Table 5.12: Alphabetised Parallel Operators

Example of replicated action

The replicated actions generalise the binary process operators over sets (or sequences in the case of sequential composition) of processes. Each takes a declaration and instantiates the process supplied, over the operator of interest, for each parameter binding admitted by the declaration. They are described in Table 5 (section 15) of the *CML0* language definition.

Several of the replicated actions can be defined directly in terms of UTP's own indexed operators (e.g., $\sqcap_i$ and $\bigsqcup_i$). However, a general scheme for expressing the meaning of replicated actions is in terms of the operators' binary equivalents, via inductive definitions. As an example, the definition for replicated external choice is given in Table 5.13.

Syntax	Definition	
`[] e: e @ A(i)`	$STOP$	if $e = \{\}$
	$\sqcap_j (A(j)$ `[]` $($`[]` $i : s \setminus \{j\} \bullet A(i)))$	if $e \neq \{\}$, where $j \in e$

Table 5.13: Replicated Action Example—External Choice

5.5.3 Time Operators

As with the parallel operators, there is a single time operator—timeout—defined in the denotational semantics, and each variant is defined in terms of this one operator. Semantically, the timeout operator is represented by the notation $a1 \overset{n}{\rhd} a2$. Its syntactic synonym is `a1` $[$`n` $|>$ `a2`.

The time operators appear in Table 3, sect 15, in the *CML0* language definition. Table 5.14 defines the variant time operators.

Syntax	Definition	Description
`A [n] B`	$A \overset{n}{\rhd} B$	Timeout (**synonym**)
`A [> B`	$\sqcap_i A \overset{i}{\rhd} B$ for $i \in \mathcal{N}$	Untimed Timeout
`wait` `n`	$STOP \overset{n}{\rhd} I\!I$	Delay
`A` **`startby`** `n`	$A \overset{n}{\rhd} \top$	Start Deadline
`A /n\ B`	$(A[\|\| \ S \mid \{\} \ \|\|]\ \text{wait}\ n) \overset{n}{\rhd} B$	Timed Interrupt (on state S)

Table 5.14: Time Operators

5.6 Global Structure

The *CML0* language specification contains a lot more than just the operators of the language, it describes the overall structure of a *CML* specification in terms of the sequences of paragraphs that comprise it. One additional notational consideration is how this overall structure is interpreted within the ***CML*** semantic (UTP) framework.

UTP prescribes no rigid format for an overall structure of a theory. However it is certainly definition-based in the sense that constants, operators and laws are built on top of definitions of operators and constants provided earlier (UTP uses the $=_{df}$ notation for this). An effective model for a ***CML*** specification is a set of UTP definitions building up an environment (for channels, actions, processes etc), in which later definitions are generally constructed by reference to, and application of operators over, earlier definitions. One way to present these definitions is in a notation very close to ***CML*** itself – in the past, for example, the meaning of a ***Circus*** specification has been given as a set of Z paragraphs [30]. The use of similar notation at both syntactic and semantic level is possible due to UTP's genericity and the syntactic/semantic overloading of notation (e.g., expressions).

Several of the definitions given in the preceding sections assume the existence of such an environment of definitions – for example, the use of channel set as arguments to actions, which would often be specified by a preceding channel set paragraph.

5.7 Summary

This Chapter contains a guide to the relationships between the syntactic and semantic notations for *CML*, the latter being based on Unifying Theories of Programming. In particular, this comprises i) how some syntactic constructs (*cf.* expressions) are lifted into the semantic domain; ii) how certain other constructs (e.g., sequential composition) are identical in both settings; iii) how some syntactic notation (e.g., `Skip`) are synonymous with their semantic counterparts; and iv) how some syntactic constructs (e.g., **`wait`**) are defined in terms of more primitive operators of the language. To do this, an overview of the UTP notation is presented, along with a discussion of expressions and predicates. The operators of the action syntax are considered in some detail, together with a brief discussion of the global structure of *CML*.

The guide is not comprehensive, nor is it intended to be. However, it has aimed to provide characteristic examples to explain how the syntactic and semantic domains of *CML* relate to one another. Some portions of the *CML* syntax have been treated as out of scope, for example pattern bindings, because further investigation is required to finalise their semantics. A major proportion of the *CML* syntax is considered out of scope because it pertains to the object-oriented features of the language, or to constructs that require the semantic extensions of OO in order to be interpreted. Examples include method calls, parametrised and instantiated actions etc. For further information about the envisaged strategy for incorporating object-oriented features see Chapter 8.

Chapter 6

Undefinedness

6.1 Introduction

We consider the problem of potentially undefined expressions in *CML*, which arise from two language constructs: partial function application and definite description.

A simple example of the problem is in the expression $y = 1/0$. Here, the division operator is a partial function that is not defined for a zero divisor: it is being applied outside its domain of definition. So what should we make of the expression "$1/0$"? Does it denote a value? If so, then which value? If not, then what do we make of the containing predicate "$y = 1/0$"? Is this defined? Does it denote a truth value or not?

More generally, if we choose a specific treatment of undefined expressions, then is it possible to use verification tools with different treatments? For example, there are two different treatments of undefined expressions for VDM: Jones's VDM uses the Logic of Partial Functions (LPF), which has been implemented in Isabelle [2], whilst Larsen's VDM in Overture uses McCarthy Conditionals [17]. What is the relationship between these? How does the treatment in the VDM part interact with the other tools that we might use? For example, in the FDR implementation of CSPM [9], undefinedness is handled by a combination of arithmetic overflow and boolean short-circuit expressions. In the *Circus* tools, undefinedness is handled through the use of classical logic and arbitrary undefined values. Does any of this matter? And what if *CML* is used for a system of systems with heterogeneous constituents using different formalisms with different solutions to the undefined problem?

One possible solution to all these problems is to adopt a single treatment of undefinedness, such as the one used in UTP [14], where the basic relational calculus is classical: there is no undefinedness and every expression denotes a value. There is an outline of a more specific treatment of undefinedness in UTP, but this is explored briefly in the book by Hoare & He [14, Section 9.3]. But there are several other possibilities, and in this Chapter we describe some of them. We need to have a firm position on undefinedness in our metalanguage that can then be used to define the possible solutions that could be chosen for *CML*. To this end, we develop a unifying theory for monotonic partial logics (we explain this term fully below).

The work presented in this Chapter forms the basis of Victor Bandur's PhD work and is based on original ideas due to Mark Saaltink in his underpinnings for the Z/Eves theorem prover [27]. They have published joint papers with the authors at Marktoberdorf and ICECCS 2007 [33].

In Section 6.2, we augment UTP's alphabetised relational calculus with a basic treatment of three-valued logic with possibly undefined expressions and predicates. In Section 6.3, we give a treatment of first-order theories for monotonic partial logics and prove a theorem about construct monotonicity (Theorem 6.3.1). In Section 6.4, we formalise three theories of undefinedness: strict logic, McCarthy's left-to-right logic, and Kleene's three-valued logic. In Section 6.5, we describe a theory of guard systems for generating verification conditions for the definedness of expressions and predicates. We present our main theorem that allows us to trade theorems between different logics by proving facts about the guard in a stronger system and guaranteeing that the construct is defined in a weaker logic (Theorem 6.5.1). We also present a guard system for the definite McCarthy logic and state its soundness (Theorem 6.5.2). Finally in Section 6.6, we draw some conclusions and plan future work.

6.2 3-valued logic in UTP

In this section, we describe a restricted semi-classical three-valued logic in UTP. The logic has a semantic value for undefined expressions and predicates. Operators of the predicate calculus are strict but equality is classical, allowing a fine control of undefinedness.

6.2.1 Basic Sets and Constructors

The set of boolean values is $\mathbb{B} = \{true, false\}$. The universe of values, disjoint from $\mathbb{B}$, is $\mathbb{U}$. We introduce a specific semantic undefined value: $\bot$. Any set not already containing undefined can be lifted to include it: $X^\bot = X \cup \{\bot\}$. Notice that $\bot$ is neither a tuple nor a function, nor is it in $\mathbb{B}$ or $\mathbb{U}$.

For k, a natural number, X^k is the set of k-tuples over X, with X^0 having the single element: the 0-tuple (). X^* is the union of all X^ks.

As usual, we have two kinds of function space: $X \to Y$, the set of total functions, and $X \nrightarrow Y$, the set of partial functions.

We take inspiration from Rose's standard encoding of three-valued logic [26], which is reminiscent of Hoare & He's UTP designs [14, Chapter 3], in modelling three logical values using just a pair of predicates: (P, Q). The intuitive meaning is that P describes the region where (P, Q) is true and Q describes the region where (P, Q) is defined. Just like Hoare & He designs, we can combine the pair of predicates into a single predicate by introducing an observational variable, in this case *def*: the observation that the predicate is defined. This gives us a model for the pair.

Definition 6.2.1 (TVL predicate pair) *The observation def is true exactly when the pair is defined (Q) and, providing it is defined, then P determines whether it is true or not.*

$$(P, Q) \;\widehat{=}\; (def \Rightarrow P) \wedge (Q = def)$$

The next example demonstrates that this definition accounts for all three logical values.

Example 6.2.1 (TVL extreme points) *Consider the four extreme points for the pair:*

$$\begin{array}{rcccc} R = \mathit{true} & = & (\mathit{true}, \mathit{true}) & = & \mathit{def} \\ R = \mathit{false} & = & (\mathit{false}, \mathit{true}) & = & \mathit{false} \\ R = \bot & = & \left\{ \begin{array}{c} (\mathit{true}, \mathit{false}) \\ (\mathit{false}, \mathit{false}) \end{array} \right\} & = & \neg\, \mathit{def} \end{array}$$

□

Two lemmas follow immediately from Definition 6.2.1. The first shows how we can make use of the definedness condition in the pair.

Lemma 6.2.1 (Definedness trading) *The definedness condition can be traded back and forth in a TVL predicate pair:*

$$(P \land Q, Q) = (P, Q)$$

Proof 6.2.1

$$\begin{array}{l} (P \land Q, Q) \\ \{\, \textit{Definition 6.2.1} \,\} \\ = (\mathit{def} \Rightarrow P \land Q) \land (Q = \mathit{def}) \\ \{\, \textit{Propositional calculus} \,\} \\ = (\mathit{def} \Rightarrow P) \land (Q = \mathit{def}) \\ \{\, \textit{Definition 6.2.1} \,\} \\ = (P, Q) \end{array}$$

□

The second lemma shows that every three-valued predicate can be expressed as a TVL pair.

Lemma 6.2.2 (TVL-model-canonical-form) *Every three-valued predicate has a canonical form:*

$$R = (R^t, \neg\, R^f), \qquad \mathit{where}\ R^b = R[b/\mathit{def}]$$

Proof 6.2.2

$$\begin{array}{l} ((P, Q)^t, \neg\, (P, Q)^f) \\ \{\, \textit{Definition 6.2.1, twice} \,\} \\ = (((\mathit{def} \Rightarrow P) \land (Q = \mathit{def}))^t, \neg\, ((\mathit{def} \Rightarrow P) \land (Q = \mathit{def}))^f) \\ \{\, \textit{Definition of } R^b. \,\} \\ = ((\mathit{true} \Rightarrow P) \land (Q = \mathit{true}), \neg\, ((\mathit{false} \Rightarrow P) \land (Q = \mathit{false})) \end{array}$$

$$
\begin{aligned}
&\{\ \textit{Propositional calculus}\ \}\\
&= (P \land Q, \neg\,(\mathit{true} \land \neg\, Q))\\
&\{\ \textit{Propositional calculus}\ \}\\
&= (P \land Q, Q)
\end{aligned}
$$

$\square$

Example 6.2.2 (Definedness of a partial expression) *Consider the predicate* $(z = x/y)$ *interpreted as a three-valued predicate. It is defined exactly when* $(y \neq 0)$*, and when it is defined, it is true when* $(x = y * z)$*, where* $(\_ * \_)$ *is the total multiplication operator. So the three-valued predicate* $(z = x/y)$ *is modelled by the pair:*

$$((x = y * z), (y \neq 0))$$

We can consider three examples with specific values for x*,* y*, and* z*.*

$$
\begin{aligned}
&(3 = 6/2)\\
&= ((6 = 2 * 3), (2 \neq 0))\\
&= (\mathit{true}, \mathit{true})\\
&= \mathit{def}
\end{aligned}
\qquad
\begin{aligned}
&(2 = 6/2)\\
&= ((6 = 2 * 2), (2 \neq 0))\\
&= (\mathit{false}, \mathit{true})\\
&= \mathit{false}
\end{aligned}
\qquad
\begin{aligned}
&(2 = 6/0)\\
&= ((6 = 0 * 2), (0 \neq 0))\\
&= (\mathit{false}, \mathit{false})\\
&= \neg\,\mathit{def}
\end{aligned}
$$

$\square$

The model that we have chosen for three-valued predicates is not closed under any of the propositional operators, so we must choose particular definitions for them. There are plenty of choices: for two operands of three values, there are nine possible results, each of three values, making a total of: $3^9 = 19,683$ combinations. We choose strict interpretations of each operator.

6.2.2 Conjunction

The conjunction of two three-valued predicates is defined as follows.

Definition 6.2.2 (TVL conjunction) $T \land_T U$ *is defined exactly when both* T *and* U *are defined; it is true exactly when both* T *and* U *are true.*

$$(P, Q) \land_T (R, S) \ \widehat{=}\ (P \land R, Q \land S)$$

It is useful to see the truth table for conjunction:

$\land_T$	def	$\neg\,\mathit{def}$	false
def	def	$\neg\,\mathit{def}$	false
$\neg\,\mathit{def}$	$\neg\,\mathit{def}$	$\neg\,\mathit{def}$	$\neg\,\mathit{def}$
false	false	$\neg\,\mathit{def}$	false

This truth table looks a little better if we replace the values in the model by the three truth values themselves:

$\land_T$	true_T	$\bot_T$	false_T
true_T	true_T	$\bot_T$	false_T
$\bot_T$	$\bot_T$	$\bot_T$	$\bot_T$
false_T	false_T	$\bot_T$	false_T

1227 □

Example 6.2.3 (Partial conjunction) *Partial conjunction gives a meaning to the case where one operand is undefined.*

$$\begin{aligned}&(y = 3) \wedge_T (z = x/y)\\ &= ((y = 3), \mathit{true}) \wedge_T ((x = y * z), (y \neq 0))\\ &= ((y = 3) \wedge (x = y * z), \mathit{true} \wedge (y \neq 0))\\ &= ((y = 3) \wedge (x = 3 * z), (y \neq 0))\end{aligned}$$

1228 □

1229 6.2.3 Negation

Definition 6.2.3 (TVL negation) *The negation of a three-valued predicate R is defined exactly when R is defined, and is true exactly when R is false:*

$$\neg_T\ (P, Q) = (\neg\ P, Q)$$

The truth table is:

$\neg_T$	
def	*false*
$\neg$ *def*	$\neg$ *def*
false	*def*

$\neg_T$	
true_T	false_T
$\bot_T$	$\bot_T$
false_T	true_T

1230 □

Example 6.2.4 (Partial negation)

$$\begin{aligned}&\neg_T\ (z = x/y)\\ &= \neg_T\ ((x = y * z), (y \neq 0))\\ &= ((x \neq y * z), (y \neq 0))\end{aligned}$$

1231 □

1232 6.2.4 Disjunction

Definition 6.2.4 (TVL disjunction) *The disjunction of two three-valued predicates $T \vee_T U$ is defined exactly when both T and U are defined; it is true when either of them is true.*

$$(P, Q) \vee_T (R, S) \ \widehat{=}\ (P \vee Q, R \wedge S)$$

The truth tables are:

$\vee_T$	*def*	$\neg$ *def*	*false*
def	*def*	$\neg$ *def*	*def*
$\neg$ *def*	$\neg$ *def*	$\neg$ *def*	$\neg$ *def*
false	*def*	$\neg$ *def*	*false*

$\vee_T$	true_T	$\bot_T$	false_T
true_T	true_T	$\bot_T$	true_T
$\bot_T$	$\bot_T$	$\bot_T$	$\bot_T$
false_T	true_T	$\bot_T$	false_T

1233 □

Example 6.2.5 (Partial disjunction) *Define $P \Rightarrow_T Q$ as $\neg_T P \vee_T Q$. Now suppose that f is a partial function symbol, such that*

$$(y = f(x)) = ((y = f(x)), x \in \operatorname{dom} f)$$

Now consider the predicate $x \in \operatorname{dom} f \Rightarrow_T (y = f(x))$. What happens if we interpret this in three-valued logic?

$$\begin{aligned}
& x \in \operatorname{dom} f \Rightarrow_T (y = f(x)) \\
&= \neg_T (x \in \operatorname{dom} f) \vee_T (y = f(x)) \\
&= \neg_T (x \in \operatorname{dom} f, \mathit{true}) \vee_T (y = f(x)) \\
&= (x \notin \operatorname{dom} f, \mathit{true}) \vee_T (y = f(x)) \\
&= (x \notin \operatorname{dom} f, \mathit{true}) \vee_T ((y = f(x)), x \in \operatorname{dom} f) \\
&= (x \notin \operatorname{dom} f \vee (y = f(x)), \mathit{true} \wedge x \in \operatorname{dom} f) \\
&= (x \in \operatorname{dom} f \Rightarrow (y = f(x)), x \in \operatorname{dom} f) \\
&= ((y = f(x)), x \in \operatorname{dom} f)
\end{aligned}$$

It is defined exactly when $x \in \operatorname{dom} f$, and when it is defined, it is true exactly when $(y = f(x))$. □

6.2.5 Equality

There is nothing special about equality in our treatment of undefined values: it is just the existing classical equality in UTP. So, two three-valued predicates are equal exactly when their representation as pairs are equal. This is the symmetric closure of the following rules:

$$\begin{array}{rclcrcl}
(\mathit{def} =_T \neg\, \mathit{def}) & = & \mathit{false} & \qquad & (\mathit{true}_T {=_T} \bot_T) & = & \mathit{false} \\
(\mathit{def} =_T \mathit{false}) & = & \mathit{false} & & (\mathit{true}_T {=_T} \mathit{false}_T) & = & \mathit{false} \\
(\neg\, \mathit{def} =_T \mathit{false}) & = & \mathit{false} & & (\bot_T {=_T} \mathit{false}_T) & = & \mathit{false}
\end{array}$$

Example 6.2.6 (Partial equality) *One of the definitions that we use later is a conditional containing five equations between three-valued predicates:*

$$(f(x, y) = \bot) \lhd (x = \bot) \vee (y = \bot) \rhd (f(x, y) = (x = y))$$

Each equation is by definition either true or false: it cannot be undefined. In this way, UTP equality contains the use of three-valued logic. We also restrict our use of quantifiers to avoid undefinedness. □

A very simple lemma is a consequence of these definitions.

Lemma 6.2.3 (TVL) *When they are defined, the TVL propositional operators behave exactly like their classical counterparts.*

1. $Q \Rightarrow (\neg_T (P, Q) = \neg\, P)$
2. $R \wedge S \Rightarrow ((P, Q) \wedge_T (R, S) = P \wedge Q)$

3. $R \wedge S \Rightarrow ((P, Q) \vee_T (R, S) = P \vee Q)$

$\square$

This justifies UTP with three-valued logic. In addition, we will not use definite description or partial functions, so we cannot manufacture undefined values. But we can build logics that do have these features.

6.3 First-order Theories

6.3.1 Contexts for First-order Theories

We introduce a context theory ***CXT*** for our first-order theories, which will all be subtheories of ***CXT***. Its alphabet contains two observational variables:

$$\begin{aligned} PShape &: \mathbb{P}((\mathbb{U}^{\perp})^* \nrightarrow \mathbb{B}^{\perp}) \\ FShape &: \mathbb{P}((\mathbb{U}^{\perp})^* \nrightarrow \mathbb{U}^{\perp}) \end{aligned}$$

and its signature is:

$$\begin{aligned} =_T &: \mathbb{U}^{\perp} \times \mathbb{U}^{\perp} \to \mathbb{B}^{\perp} \\ \neg_T &: \mathbb{B}^{\perp} \to \mathbb{B}^{\perp} \\ \vee_T &: \mathbb{B}^{\perp} \times \mathbb{B}^{\perp} \to \mathbb{B}^{\perp} \\ \forall_T &: (\mathbb{U} \nrightarrow \mathbb{B}^{\perp}) \to \mathbb{B}^{\perp} \\ \iota_T &: (\mathbb{U} \nrightarrow \mathbb{B}^{\perp}) \to \mathbb{U}^{\perp} \end{aligned}$$

PShape describes all the possible denotations for the predicate symbols of this theory. Every denotation is a partial function from some number of parameters, each of which could be drawn from $\mathbb{U}$ or could be undefined, to a boolean result, which could also be undefined. The purpose of *PShape* is to constrain all the theory's predicate symbols in a uniform way. *FShape* does the same job as *PShape*, except that it describes the possible denotations of function symbols. The operators $=_T$, $\neg_T$, and $\vee_T$ give the syntax for equality, negation, and disjunction, respectively.

The $\forall_T$ function takes as its argument a function $\mathbb{U} \nrightarrow \mathbb{B}^{\perp}$ that describes a binding for the universal quantifier that characterises the predicate that must be universally true. The function considers each element of its domain in turn and assigns to it one of the three logical values. The $\forall_T$ function takes this binding function and decides whether the universally quantified predicate is true, false, or undefined. Notice that the binding function ranges only over defined values. This means that we are excluding logics where bound variables may be undefined, as is the case in LCF [10].

The ι_T function also takes a binding function as its argument. It decides whether this binding is a definite description of a value in $\mathbb{U}$ or is undefined. Once more, the bound variable must be everywhere defined.

We add a single healthiness condition to constrain the definite description function:

$$\begin{aligned} &\textbf{\textit{CTX}}(P) = \\ &\qquad P \wedge (\forall f : \mathbb{U} \nrightarrow \mathbb{B}^{\perp} \bullet f \neq \emptyset \Rightarrow \iota_T(f) \in \operatorname{dom} f^{\perp}) \end{aligned}$$

This requires that the definite description of a non-empty binding function returns either an undefined value or an element from the domain of the binding. We require this result in Lemma 6.3.3, where we prove that theories are closed under constructs over their signature.

Example 6.3.1 (Context) *Consider a context with no predicate symbols and only monadic and dyadic function symbols.*

$$\mathbf{X1}(P) = \\ P \wedge (PShape = \emptyset) \wedge (FShape = (\mathbb{U}^{\perp} \cup (\mathbb{U}^{\perp})^2 \nrightarrow \mathbb{U}^{\perp}))$$

PShape and FShape are used to add type information: we use them to restrict how predicate and function symbols behave, particularly, as we shall see later, with respect to undefinedness. $\square$

6.3.2 First-order Theory

A first-order theory is an enrichment of a particular context and acts as its model. We add to the context six alphabetical variables and three healthiness conditions. The set of names A is partitioned into three sets: variables, predicate symbols, and function symbols.

$$A \text{ partition } \langle Var, Pred, Fun \rangle$$

The set $Dom : \mathbb{P}\,\mathbb{U}$ describes the domain of values for the first-order theory. Finally, the rank function $\rho : Pred \cup Fun \to \mathbb{N}$ describes the number of parameters that each predicate and function symbol can take.

The first healthiness condition requires that every variable is defined and has a value drawn from Dom:

$$\mathbf{DV}(P) = P \wedge (\forall\, v : Var \bullet v \in Dom)$$

The second and third healthiness conditions require that every predicate and function symbol ranges over arguments taken from $Dom^{\perp}$ and produces results in $\mathbb{B}^{\perp}$ and $\mathbb{U}^{\perp}$, respectively:

$$\mathbf{DP}(P) = P \wedge (\forall\, p : Pred \bullet p \in ((Dom^{\perp})^{\rho(p)} \to \mathbb{B}^{\perp}) \cap PShape)$$

$$\mathbf{DF}(P) = P \wedge (\forall\, f : Fun \bullet f \in ((Dom^{\perp})^{\rho(f)} \to Dom^{\perp}) \cap FShape)$$

Example 6.3.2 (First-order theory) *Consider a theory* $\mathbf{T1}$ *with context* $\mathbf{X1}$ *that has just a single function symbol for integer division:*

$$\begin{aligned} \mathbf{T1}(P) = {} & \\ & \mathbf{X1}(P) \\ & \wedge\ Var = \emptyset \\ & \wedge\ Pred = \emptyset \\ & \wedge\ Fun = \{\_/\_\} \\ & \wedge\ Dom = \mathbb{N} \\ & \wedge\ \rho = \{\_/\_ \mapsto 2\} \\ & \wedge\ \_/\_ \in (\mathbb{N}^{\perp} \times \mathbb{N}^{\perp} \to \mathbb{N}^{\perp}) \cap FShape \end{aligned}$$

$\square$

6.3.3 Information-theoretic Ordering

Our whole approach to unifying the treatment of undefinedness in different logics is built on a rather flat information-theoretic ordering. This says that the undefined value is worse than every other value; these other values are incomparable with each other.

Definition 6.3.1 (Information-theoretic ordering) *Elements: for any set X with $a, b \in X$*

$$a \sqsubseteq b \;\widehat{=}\; (a \neq \bot) \Rightarrow (a = b)$$

Pointwise extension to tuples: for $x, y \in X^k$

$$x \sqsubseteq y \;\widehat{=}\; \forall\, i : 1\,..\,k \bullet x_i \sqsubseteq y_i$$

Pointwise extension to functions: for $f, g \in X \to Y$

$$f \sqsubseteq g \;\widehat{=}\; (\mathrm{dom}\, f = \mathrm{dom}\, g) \wedge (\forall\, x : \mathrm{dom}\, f \bullet f(x) \sqsubseteq g(x))$$

Comparing sets of functions: for $A, B : \mathbb{P}\, X$, the Hoare preorder is defined:

$$A \sqsubseteq_H B \;\widehat{=}\; \forall\, a : A \bullet \exists\, b : B \bullet a \sqsubseteq b$$

□

These definitions are illustrated in the following set of examples.

Example 6.3.3 (Ordering)

1. *Elements:*

$$\begin{array}{rcl} \bot & \sqsubseteq & 1 \\ 1 & \sqsubseteq & 1 \\ \neg\,(1 & \sqsubseteq & 2) \end{array}$$

2. *Tuples:*

$$\begin{array}{rcl} (0, \bot, 2) & \sqsubseteq & (0, 1, 2) \\ () & \sqsubseteq & () \\ (1, 2) & \sqsubseteq & (1, 2) \\ \neg\,((1, 2) & \sqsubseteq & (2, 2)) \end{array}$$

3. *Functions:*

$$\begin{array}{l} (\lambda\, x, y : \mathbb{N} \bullet \bot \lhd (y = 0) \rhd x/y) \\ \qquad \sqsubseteq (\lambda\, x, y : \mathbb{N} \bullet 0 \lhd (y = 0) \rhd x/y) \\ (\lambda\, n : \mathbb{N} \bullet \bot \lhd (n \bmod 2 = 0) \rhd n) \sqsubseteq (\lambda\, n : \mathbb{N} \bullet n) \end{array}$$

4. *Sets of functions:*

$$\begin{array}{l}\{(\lambda\, x, y : \mathbb{N} \bullet \bot \lhd (y = 0) \rhd x/y),\\ \ (\lambda\, n : \mathbb{N} \bullet \bot \lhd (n \mathsf{\ mod\ } 2 = 0) \rhd n),\\ \ (\lambda\, n : \mathbb{N} \bullet n)\}\\ \sqsubseteq_H\\ \{(\lambda\, x, y : \mathbb{N} \bullet 0 \lhd (y = 0) \rhd x/y),\\ \ (\lambda\, n : \mathbb{N} \bullet n)\}\end{array}$$

$\square$

We further generalise the ordering by lifting it to contexts.

Definition 6.3.2 (Ordering on contexts)

$$\boldsymbol{S} \sqsubseteq_H \boldsymbol{T} = \forall\, P : \boldsymbol{S};\ Q : \boldsymbol{T} \bullet P \sqsubseteq_H Q$$

where

$$\begin{array}{l} P \sqsubseteq_H Q = \\ \quad PShape_{\boldsymbol{S}} \sqsubseteq_H PShape_{\boldsymbol{T}} \\ \quad \land\ FShape_{\boldsymbol{S}} \sqsubseteq_H FShape_{\boldsymbol{T}} \\ \quad \land\ (=_{\boldsymbol{S}}) \sqsubseteq (=_{\boldsymbol{T}}) \\ \quad \land\ (\neg_{\boldsymbol{S}}) \sqsubseteq (\neg_{\boldsymbol{T}}) \\ \quad \land\ (\lor_{\boldsymbol{S}}) \sqsubseteq (\lor_{\boldsymbol{T}}) \\ \quad \land\ (\forall_{\boldsymbol{S}}) \sqsubseteq (\forall_{\boldsymbol{T}}) \\ \quad \land\ (\iota_{\boldsymbol{S}}) \sqsubseteq (\iota_{\boldsymbol{T}}) \end{array}$$

Example 6.3.4 (Subtheory) *Consider* ***X2****, a subtheory of* ***X1****, where the following holds:*

$$\forall\, f : FShape_{\boldsymbol{X1}} \bullet zero \circ f \in FShape_{\boldsymbol{X2}}$$

and where the total function zero is defined:

$$zero(x) \ \widehat{=}\ (0 \lhd (x = \bot) \rhd x)$$

All other components remain unchanged. Then $P_{\boldsymbol{X1}} \sqsubseteq_H P_{\boldsymbol{X2}}$*, since*

$$\begin{array}{l} f \sqsubseteq zero \circ f \\ = (\mathrm{dom}\, f = \mathrm{dom}(zero \circ f)) \land \forall\, x : \mathrm{dom}\, f \bullet f(x) \sqsubseteq zero \circ f(x) \end{array}$$

and so we have $FShape_{X1} \sqsubseteq_H FShape_{X2}$. $\square$

In the following sections, we introduce the three important notions of strictness, definiteness, and monotonicity.

6.3.4 Strictness

The notion of strictness is a familiar one from the definition of programming languages. A function f is strict if $f(\bot) = \bot$, and it is usually used to denote that a function loops forever or performs an illegal operation, such as division by zero. We can interpret a strict function operationally as one that always evaluates all of its arguments. A restricted notion considers functions that are strict in one or more arguments.

Definition 6.3.3 (Strictness) *Function $f : (X^\perp)^{\rho(f)} \to Y^\perp$ is strict if, whenever at least one of its arguments is undefined, then the result is undefined:*

$$\textbf{\textit{strict}}(f) = \forall\, x : (X^\perp)^{\rho(f)} \bullet (\exists\, i : 1 \mathinner{..} \rho(f) \bullet (x_i = \bot)) \Rightarrow (f(x) = \bot)$$

□

Example 6.3.5 (Strict function) *Suppose that $\_*\_$ is the standard multiplication operator on natural numbers: $\_*\_ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Now define a strict version of the operator:*

$$\begin{aligned}&\_*\_ : \mathbb{N}^\perp \times \mathbb{N}^\perp \to \mathbb{N}^\perp\\ &x * y = \bot \lhd (x = \bot) \vee (y = \bot) \rhd x * y\end{aligned}$$

□

We can extend the notion of strictness to a context, where every predicate has only strict denotations for its predicate and function symbols. We find it useful to define a healthiness function ***strict***() that it applied to a context (which of course is a set of predicates).

Definition 6.3.4 (Strict contexts) *We make a context **T** strict:*

$$\begin{aligned}&\textbf{\textit{strict}}(\textbf{\textit{T}}) = \{\, P : \textbf{\textit{T}} \bullet \textbf{\textit{strict}}(P) \,\}\\ &\textbf{\textit{where strict}}(P) = \exists\, PShape_0, FShape_0 \bullet\\ &\qquad PShape = \{\, p : PShape_0 \mid \textbf{\textit{strict}}(p) \,\}\\ &\qquad \wedge\ FShape = \{\, f : FShape_0 \mid \textbf{\textit{strict}}(f) \,\}\\ &\qquad \wedge\ P[PShape_0, FShape_0 / PShape, FShape]\end{aligned}$$

□

6.3.5 Definiteness

Definiteness is, in a sense, a dual notion to strictness. If a function is definite, then it cannot manufacture undefinedness. That is, if the function produces an undefined result, then it must have had an undefined argument.

Definition 6.3.5 (Definite) *Function $f : (X^\perp)^{\rho(f)} \to Y^\perp$ is definite:*

$$\textbf{\textit{definite}}(f) = \forall\, x : (X^\perp)^{\rho(f)} \bullet (f(x) = \bot) \Rightarrow (\exists\, i : 1 \mathinner{..} \rho(f) \bullet (x_i = \bot))$$

□

Example 6.3.6 (Definite function) *$\_*\_$ is definite.* □

As for strictness, we define a healthiness function for contexts.

Definition 6.3.6 (Definite Contexts) *Making a context definite:*

$$\begin{aligned}&\textbf{\textit{definite}}(\textbf{\textit{T}}) = \{\, P : \textbf{\textit{T}} \bullet \textbf{\textit{definite}}(P) \,\}\\ &\textbf{\textit{where definite}}(P) =\\ &\qquad \exists\, PShape_0, FShape_0 \bullet\\ &\qquad\quad PShape = \{\, p : PShape_0 \mid \textbf{\textit{definite}}(p) \,\}\\ &\qquad\quad \wedge\ FShape = \{\, f : FShape_0 \mid \textbf{\textit{definite}}(f) \,\}\\ &\qquad\quad \wedge\ P[PShape_0, FShape_0 / PShape, FShape]\end{aligned}$$

□

6.3.6 Monotonicity

A monotonic function on ordered sets is one that preserves that order. In our unifying theory, we are interested in defined-monotonic functions, that is, one that preserves the definedness ordering.

Definition 6.3.7 (Monotonicity) *Function $f : (X^{\perp})^{\rho(f)} \to Y^{\perp}$ is monotonic:*

$$\textbf{\textsf{monotonic}}(f) = \forall\, x_1, x_2 : (X^{\perp})^{\rho(f)} \bullet x_1 \sqsubseteq x_2 \Rightarrow f(x_1) \sqsubseteq f(x_2)$$

□

Example 6.3.7 (Monotonicity) $\neg_T$ *is monotonic*

$\neg_T$	
true	*false*
$\perp$	$\perp$
false	*true*

□

This time, it is convenient to define a predicate that is true if a context is monotonic.

Definition 6.3.8 (Monotonic Contexts) $\boldsymbol{T}$ *is a monotonic context:*

$$
\begin{array}{l}
\textbf{\textsf{monotonic}}(\boldsymbol{T}) = \forall\, P : \boldsymbol{T} \bullet \textbf{\textsf{monotonic}}(P) \\
\textbf{\textsf{where monotonic}}(P) = \\
\qquad\qquad (\forall\, p : Pred_{\boldsymbol{T}} \bullet \textbf{\textsf{monotonic}}(p)) \\
\qquad\qquad \wedge\, (\forall\, f : Fun_{\boldsymbol{T}} \bullet \textbf{\textsf{monotonic}}(f)) \\
\qquad\qquad \wedge\, \textbf{\textsf{monotonic}}(=_{\boldsymbol{T}}) \\
\qquad\qquad \wedge\, \textbf{\textsf{monotonic}}(\neg_{\boldsymbol{T}}) \\
\qquad\qquad \wedge\, \textbf{\textsf{monotonic}}(\vee_{\boldsymbol{T}}) \\
\qquad\qquad \wedge\, \textbf{\textsf{monotonic}}(\forall_{\boldsymbol{T}}) \\
\qquad\qquad \wedge\, \textbf{\textsf{monotonic}}(\iota_{\boldsymbol{T}})
\end{array}
$$

□

The following simple lemma is useful.

Lemma 6.3.1 (Strict monotonic) *Every strict function is monotonic.*

6.3.7 Comparing FOTs

In Definition 6.3.2, we lifted our information-theoretic ordering up to contexts; now we lift it to first-order theories. This makes sense only if the two FOTs in question have the same domain of values.

Definition 6.3.9 (Comparing FOTs) *Comparing FOTs* $\boldsymbol{U}$ *and* $\boldsymbol{V}$*: for* $P : \boldsymbol{U}$ *and* $Q : \boldsymbol{V}$

$$P \sqsubseteq_H Q = Dom_{\boldsymbol{U}} = Dom_{\boldsymbol{V}} \wedge Pred_{\boldsymbol{U}} \sqsubseteq_H Pred_{\boldsymbol{V}} \wedge Fun_{\boldsymbol{U}} \sqsubseteq_H Fun_{\boldsymbol{V}}$$

□

Using this definition, we can state an important lemma. If $\boldsymbol{S}$ and $\boldsymbol{T}$ are two contexts, such that $\boldsymbol{S}$ is less defined than (or equal to) $\boldsymbol{T}$, and we have a FOT that models $\boldsymbol{S}$, then there will also be a FOT that models $\boldsymbol{T}$.

Lemma 6.3.2 (Models) *Suppose that we have two CXTs $\boldsymbol{S}$ and $\boldsymbol{T}$, where $\boldsymbol{S} \sqsubseteq_H \boldsymbol{T}$. Suppose further that $\boldsymbol{U}$ is a FOT extending $\boldsymbol{S}$. Then there is a FOT $\boldsymbol{V}$ extending $\boldsymbol{T}$ such that $\boldsymbol{U} \sqsubseteq \boldsymbol{V}$.* □

The proof of this lemma is quite straightforward. The relationship between $\boldsymbol{S}$ and $\boldsymbol{T}$ shows where undefined values in the former have been replaced by defined values in the latter. This is used as a guide to construct an appropriate model.

Example 6.3.8 (Application of Model Lemma) *Suppose that we have two contexts $\boldsymbol{S}$ and $\boldsymbol{T}$. Suppose further that $\boldsymbol{S}$ has only a single monadic function symbol inc : $\mathbb{U}^{\perp} \nrightarrow \mathbb{U}^{\perp}$. Define a simple model $\boldsymbol{U}$ for $\boldsymbol{S}$ that instantiates inc as a rather trivial increment operation on binary digits. This operation is easy to define on the argument 0, it returns the result 1. It is undefined otherwise. The context $\boldsymbol{T}$, on the other hand produces only defined results inc : $\mathbb{U}^{\perp} \nrightarrow \mathbb{U}$. There must be a model $\boldsymbol{V}$ for $\boldsymbol{T}$, such that $\boldsymbol{U} \sqsubseteq \boldsymbol{V}$. This is easy to construct. The domain of values has to be the same as for $\boldsymbol{U}$. The inc can return an arbitrary value for any argument that returns $\perp$. Note that this makes it non-strict: it must produce a defined value for the argument $\perp$. All this is summarised in the following table:*

	$\boldsymbol{S}$	$\boldsymbol{T}$
$PShape$	$\emptyset$	$\emptyset$
$FShape$	$\textbf{strict}(\mathbb{U}^{\perp} \nrightarrow \mathbb{U}^{\perp})$	$\mathbb{U}^{\perp} \nrightarrow \mathbb{U}$
	$\boldsymbol{U}$	$\boldsymbol{V}$
Dom	$\{0,1\}$	$\{0,1\}$
ρ	$\{inc \mapsto 1\}$	$\{inc \mapsto 1\}$
A	$inc(\perp) = \perp$	$inc(\perp) = 0$
	$inc(0) = 1$	$inc(0) = 1$
	$inc(1) = \perp$	$inc(1) = 1$

□

We state another important lemma about the closure of a FOT under the syntax of expressions.

Lemma 6.3.3 (Expression Consistency) *Suppose that e is an expression over a FOT $\boldsymbol{U}$, then every $\boldsymbol{U}$-healthy predicate P ensures:*

$$P \Rightarrow e \in Dom_{\boldsymbol{U}}^{\perp}$$

□

This lemma is proved by syntactic induction.

A third important result is the following theorem that states that constructs (expressions or predicates) are monotonic.

Theorem 6.3.1 (Construct Monotonicity) *Suppose $\boldsymbol{S} \sqsubseteq_H \boldsymbol{T}$, that $\boldsymbol{U}$ extends $\boldsymbol{S}$, $\boldsymbol{V}$ extends $\boldsymbol{T}$, and that either $\boldsymbol{S}$ or $\boldsymbol{T}$ is monotonic. Then, for any construct c, we have*

$$c_{\boldsymbol{U}} \sqsubseteq c_{\boldsymbol{V}}$$

Proof 6.3.1 (Construct monotonicity) *The proof of the theorem is by induction on the syntax of the construct c. To illustrate the proof, we consider only the second induction case: application of a function symbol to actual parameters. This is enough to demonstrate the role of monotonicity in one of the two contexts.*

The induction hypothesis is that $x_{\mathbf{S}} \sqsubseteq x_{\mathbf{T}}$.

Case 2.1: ***S*** *is monotonic*

$$
\begin{aligned}
& (f(x))_{\mathbf{U}} && \{\ \text{interpretation}\ \}\\
&= f_{\mathbf{U}}(x_{\mathbf{U}}) && \{\ \text{hypothesis } x_{\mathbf{U}} \sqsubseteq x_{\mathbf{V}} + \mathbf{S} \text{ monotonic, and so } f_{\mathbf{U}} \text{ is monotonic}\ \}\\
&\sqsubseteq f_{\mathbf{U}}(x_{\mathbf{V}}) && \{\ \text{assumption: } P_{\mathbf{U}} \sqsubseteq Q_{\mathbf{V}}\text{, and so } Fun_{\mathbf{U}} \sqsubseteq Fun_{\mathbf{V}} \text{ and so } f_{\mathbf{U}} \sqsubseteq f_{\mathbf{V}}\ \}\\
&\sqsubseteq f_{\mathbf{V}}(x_{\mathbf{V}}) && \{\ \text{interpretation}\ \}\\
&= (f(x))_{\mathbf{V}}
\end{aligned}
$$

Case 2.2: ***T*** *is monotonic*

$$
\begin{aligned}
& (f(x))_{\mathbf{U}} && \{\ \text{interpretation}\ \}\\
&= f_{\mathbf{U}}(x_{\mathbf{U}}) && \{\ \text{assumption: } P_{\mathbf{S}} \sqsubseteq P_{\mathbf{T}}\ \}\\
&\sqsubseteq f_{\mathbf{V}}(x_{\mathbf{U}}) && \{\ \text{hypothesis} + \mathbf{V} \text{ monotonic}\ \}\\
&\sqsubseteq f_{\mathbf{V}}(x_{\mathbf{V}}) && \{\ \text{interpretation}\ \}\\
&= (f(x))_{\mathbf{V}}
\end{aligned}
$$

□

6.4 Specific First-order Theories

In this section we consider three different theories of undefinedness: strict logic, McCarthy's logic, Kleene's logic. In our definitions, we demonstrate the differences between these three; in our theorems, we demonstrate the similarities.

6.4.1 Strict Logic

Strict logic treats undefinedness as extremely contagious: whenever an undefined value appears in an expression or predicate, the overall construct collapses to become undefined. As we saw in Definition 6.3.3, this is strictness. First of all, every predicate in this theory is strict (see Definition 6.3.4). This means that *PShape* and *FShape* both contain only strict denotations.

$$\mathbf{S1}(P) = \mathbf{strict}(P)$$

Next, equality is strict:

$$(=_{\mathbf{s}}(x, y) = \bot) \lhd (x = \bot) \vee (y = \bot) \rhd (=_{\mathbf{s}}(x, y) = (x = y))$$

Recall Example 6.2.6 for an explanation of the definedness of this definition. If either argument is undefined, then the equality is undefined: otherwise, strict equality depends on the underlying UTP equality.

Definite description is strict:

$$(\iota_s(f) = x) \lhd \bot \notin \operatorname{ran} f \wedge (\operatorname{dom}(f \rhd \{\mathit{true}\}) = \{x\}) \rhd (\iota_s(f) = \bot)$$

The argument to ι_s is a function f that binds elements of its domain to one of three truth values. If this binding is everywhere defined and there is only one element of f's domain that satisfies f's characteristic predicate, then the definite description is exactly this element. Otherwise, it is undefined.

The universal quantifier is strict. Once more, the argument to $\forall_s$ is a binding. If this binding is anywhere undefined, then the universal quantifier is itself undefined. Otherwise, it depends on whether every element evaluates to true or not.

$$(\forall_s(f) = \bot) \lhd \bot \in \operatorname{ran} f \rhd (\forall_s(f) = (\operatorname{ran} f = \{\mathit{true}\}))$$

Negation is strict and is modelled by the underlying strict UTP operator:

$$\neg_s(P) = \neg\, P$$

Similarly, disjunction is strict and is modelled by the underlying UTP strict operator:

$$\vee_s\,(P, Q) = P \vee Q$$

The last two definitions are perhaps more appealing as truth tables.

$\neg_s$	
true	*false*
$\bot$	$\bot$
false	*true*

$\vee_s$	*true*	$\bot$	*false*
true	*true*	$\bot$	*true*
$\bot$	$\bot$	$\bot$	$\bot$
false	*true*	$\bot$	*false*

6.4.2 Kleene System

Kleene's system makes the logical connectives as defined as possible, whilst still being monotonic. So, every function is monotonic:

$$\mathbf{K1}(P) = P \wedge (\forall f : \mathit{PShape}_k \cup \mathit{FShape}_k \bullet \mathbf{monotonic}(f))$$

Equality and definite are both strict:

$$(=_k) = (=_s)$$
$$(\iota_k) = (\iota_s)$$

If the binding function f for the universal quantifier evaluates anywhere to *false*, then this is enough information to constitute a counterexample, and so $\forall_k(f)$ is also *false*. Otherwise, if it evaluates everywhere to *true*, then clearly it is universally satisfied. Otherwise, it is undefined.

$$((\forall_k(f) = \mathit{false}) \lhd \mathit{false} \in \operatorname{ran} f \rhd \\ ((\forall_k(f) = \mathit{true}) \lhd (\operatorname{ran} f = \{\mathit{true}\}) \rhd (\forall_k(f) = \bot)))$$

Negation is strict:

$$\neg_{\textbf{\textit{k}}} = \neg_{\textbf{\textit{s}}}$$

If either operand is *true*, then the disjunction is also *true*, regardless of whether the other operand is defined or not. If both are false, then so is the disjunction. Otherwise the disjunction is undefined.

$$\begin{array}{l} ((\vee_{\textbf{\textit{k}}}(P, Q) = \mathit{true}) \lhd (P = \mathit{true}) \vee (Q = \mathit{true}) \rhd \\ \quad ((\vee_{\textbf{\textit{k}}}(P, Q) = \mathit{false}) \lhd (P = \mathit{false}) \wedge (Q = \mathit{false}) \rhd \\ \quad\quad (\vee_{\textbf{\textit{k}}}(P, Q) = \perp))) \end{array}$$

$\vee_{\textbf{\textit{k}}}$	*true*	$\perp$	*false*
true	*true*	*true*	*true*
$\perp$	*true*	$\perp$	$\perp$
false	*true*	$\perp$	*false*

1369 ### 6.4.3 McCarthy System

McCarthy's system is very operational in flavour: it is assumed that there is an interpreter working through the text of logical constructs from left to right. The left-hand operand is evaluated first. The right-hand operand is evaluated only if it is needed. Function and predicate symbols are monotonic, just like Kleene's system.

$$\textbf{\textit{M1}} = \textbf{\textit{K1}}$$

Equality and definite description are both strict.

$$(=_{\textbf{\textit{m}}}) = (=_{\textbf{\textit{k}}})$$
$$\iota_{\textbf{\textit{m}}} = \iota_{\textbf{\textit{k}}}$$

In general, universal quantification in McCarthy's system is just the same as in the Kleene's system. However, Overture [17] uses a variant of McCarthy where the binding function is executed from left to right, which distinguishes it from a Kleene.

$$\forall_{\textbf{\textit{m}}} = \forall_{\textbf{\textit{k}}}$$

Negation is the same as Kleene.

$$\neg_{\textbf{\textit{m}}} = \neg_{\textbf{\textit{k}}}$$

Finally, disjunction has a short-circuit semantics making the left-to-right evaluation:

$$\begin{array}{l} ((\vee_{\textbf{\textit{m}}}\ (P, Q) = \mathit{true}) \lhd (P = \mathit{true}) \vee ((P = \mathit{false}) \wedge (Q = \mathit{true})) \rhd \\ \quad ((\vee_{\textbf{\textit{m}}}\ (P, Q) = \perp) \lhd (P = \perp) \vee (Q = \perp) \rhd (\vee_{\textbf{\textit{m}}}\ (P, Q) = \mathit{false}))) \end{array}$$

$\vee_{\textbf{\textit{m}}}$	*true*	$\perp$	*false*
true	*true*	*true*	*true*
$\perp$	$\perp$	$\perp$	$\perp$
false	*true*	$\perp$	*false*

1370 All three systems are monotonic.

Lemma 6.4.1 (strict-Kleene-McCarthy monotonicity)

1. *strict system is monotonic*
2. *Kleene system is monotonic*
3. *McCarthy system is monotonic*

□

There is an interesting definedness order between the three systems:

Lemma 6.4.2 (Strict-McCarthy-Kleene ordering) *for* $\rho_{\boldsymbol{s}} = \rho_{\boldsymbol{m}} = \rho_{\boldsymbol{k}}$ *and* $Dom_{\boldsymbol{s}} = Dom_{\boldsymbol{m}} = Dom_{\boldsymbol{k}}$ *q*

$$\boldsymbol{FOT}_{\boldsymbol{s}} \sqsubseteq \boldsymbol{FOT}_{\boldsymbol{m}} \sqsubseteq \boldsymbol{FOT}_{\boldsymbol{k}}$$

□

This lemma allows us relate theorems proved in the different systems. Suppose that P is a theorem in the strict system; then it would also be true in the McCarthy and Kleene systems. More concretely, if we prove a theorem in VDM in Overture, then it would still be a theorem if we interpreted it in LPF, since the former is a McCarthy system and the latter is a Kleene system.

6.5 Guard Systems

We turn our attention now to the proof obligations that different systems can use to demonstrate the definedness of constructs.

6.5.1 Validity

Suppose $\boldsymbol{T}$ is a $\boldsymbol{CXT}$ and P is a predicate. Then define P is valid in $\boldsymbol{T}$:

$$\boldsymbol{T} \models P \ \ \widehat{=} \ \ \text{for all } \boldsymbol{U}, \boldsymbol{T} \sqsubseteq_H \boldsymbol{U} \text{ implies } P_{\boldsymbol{U}} = \mathit{true}$$

6.5.2 Guards

Suppose that c is a construct. Then predicate G is a guard for c in $\boldsymbol{CXT}_{\boldsymbol{T}}$ (denoted by $G \rightsquigarrow_{\boldsymbol{T}} P$) iff for every $\boldsymbol{FOT}_{\boldsymbol{V}}$ that extends $\boldsymbol{CXT}_{\boldsymbol{T}}$ we have

1. $(G_{\boldsymbol{V}} \neq \bot)$
2. $(G_{\boldsymbol{V}} = \mathit{true}) \Rightarrow (c_{\boldsymbol{V}} \neq \bot)$

G is a tight guard if we also have

3. $(G_{\boldsymbol{V}} = \mathit{false}) \Rightarrow (c_{\boldsymbol{V}} = \bot)$

Now we are ready to state and prove our main result, which is due originally to Saaltink.

Theorem 6.5.1 (Main Theorem (Saaltink)) *Suppose that* $\mathbf{CXT_S} \sqsubseteq \mathbf{CXT_T}$*, that either one is monotonic, and that* G *is a guard for* P *in* $\mathbf{CXT_S}$*. Then, if* $(\mathbf{T} \models G)$ *and* $(\mathbf{T} \models P)$*, we have that* $(\mathbf{S} \models P)$*.* □

The significance of this result is in trading theorems between provers, as shown in the next example.

Example 6.5.1 (Trading theorems) *Suppose that we want a proof of* P *in Larsen's VDM, as implemented in the Overture toolset [17], but the only theorem prover we have is for Jones's VDM. Overture uses a form of McCarthy's logic, whilst Jones's VDM uses LPF, a form of Kleene's logic. By Lemma 6.4.2, we have Overture* $\sqsubseteq$ *LPF. We could find a guard* G *for* P *in Overture (McCarthy logic), and then can carry out the proof of both* G *and* P *in Jones's logic (Kleene). Our Main Theorem then tells us that* P *is a theorem in Overture. All proofs are carried out in the stronger logic, but hold in weaker one. Perhaps more interestingly, a similar theorem holds for using classical logic instead of Kleene's logic. In this way, classical logic could be used to prove results in Overture.*
□

Proof 6.5.1 (Main Theorem)

1. *From the Models Lemma 6.3.2, since* $\mathbf{CXT_S} \sqsubseteq \mathbf{CXT_T}$ *and* $\mathbf{FOT_U}$ *extends* $\mathbf{CXT_S}$*, then there exists* $\mathbf{FOT_V}$ *that extends* $\mathbf{CXT_T}$ *and for which we have* $\mathbf{FOT_U} \sqsubseteq \mathbf{FOT_V}$*.*

2. *Since* $G \rightsquigarrow_{\mathbf{S}} P$*, know that* $(G_{\mathbf{U}} \neq \bot) \wedge ((G_{\mathbf{U}} = true) \Rightarrow (P_{\mathbf{U}} \neq \bot))$ *from the definition of a guard.*

3. *Now, from construct monotonicity (since* $\mathbf{S}$ *is monotonic) we have that* $G_{\mathbf{U}} \sqsubseteq G_{\mathbf{V}}$*. But because* $(G_{\mathbf{U}} \neq \bot)$*, it must be that* $(G_{\mathbf{U}} = G_{\mathbf{V}})$*. We are assuming that* G *is valid in* T $(\mathbf{T} \models G)$*, so we have that* $(G_{\mathbf{V}} = true)$ *and so* $(G_{\mathbf{U}} = true)$*. Now, from the definition of a guard, we must have that* $(P_{\mathbf{U}} \neq \bot)$

4. *We now repeat this argument for* P*. By construct monotonicity, (*$\mathbf{S}$ *monotonic), we have* $P_{\mathbf{U}} \sqsubseteq P_{\mathbf{V}}$*, therefore* $(P_{\mathbf{U}} = P_{\mathbf{V}})$*. But* $\mathbf{T} \models P$*, so* $(P_{\mathbf{V}} = true)$ *and therefore* $(P_{\mathbf{U}} = true)$*.*

6.5.3 Definedness Guards

Suppose that e is an expression. We use the notation $\mathcal{D}e$ to define the circumstances under which e is defined.

Example 6.5.2 (Definedness guard)

$$\mathcal{D}((x+y)/z) = z \neq 0$$

□

The definedness guards that we are interested in are all first order; that is, the guards themselves are always defined.

Definition 6.5.1 (First-order definedness) *The definedness function is first order:*

$$\mathbf{D1}(\mathcal{D}\Phi) \;\widehat{=}\; \mathcal{D}\Phi \wedge \mathcal{D}(\mathcal{D}\Phi)$$

If we define a system of guards for every construct in our language, then we can use this system inductively to generate verification conditions for the definedness of all constructs. In the next section we demonstrate this for the case of the definite McCarthy system.

6.5.4 Guards for Definite McCarthy System

$$
\begin{array}{rcl}
\mathcal{D}_{m} x & = & true \\
\mathcal{D}_{m}(p(e)) & = & \forall\, i : 1 \,..\, \rho(P) \bullet \mathcal{D}_{m} e_i \\
\mathcal{D}_{m}(f(e)) & = & \forall\, i : 1 \,..\, \rho(f) \bullet \mathcal{D}_{m} e_i \\
\mathcal{D}_{m}(e_1 = e_2) & = & \mathcal{D}_{m} e_1 \wedge \mathcal{D}_{m} e_2 \\
\mathcal{D}_{m}(\neg\, P) & = & \mathcal{D}_{m} P \\
\mathcal{D}_{m}(P \vee Q) & = & \mathcal{D}_{m} P \wedge (P \vee \mathcal{D}_{m} Q) \\
\mathcal{D}_{m}(\forall\, x \bullet P) & = & \forall\, x \bullet \mathcal{D}_{m} P \\
\mathcal{D}_{m}(\iota x \bullet P) & = & (\forall\, x \bullet \mathcal{D}_{m} P) \wedge (\exists_1\, x \bullet P)
\end{array}
$$

Theorem 6.5.2 (McCarthy guards) *If c is a construct, then $\mathcal{D}_{m}(c)$ is a guard for c in* ***definite*(*T*)*, and a tight guard for c in* ***strict*(*definite*(*T*))** □

6.6 Summary

6.6.1 Contribution

We have presented a unifying theory for monotonic partial logics with undefined expressions, based closely on Saaltink's original work, but cast in Hoare & He's Unifying Theories of Programming. We have demonstrated this work for three logical systems (strict, McCarthy, and Kleene). These results can now be used to give semantics for the treatment of undefined constructs in *CML*.

6.6.2 Future work

This is only part of the story, since *CML* is not restricted to definite constructs: precondition predicates are needed for handling indefinite expressions and predicates. The next step will be to extend the work in this way, so that a comprehensive treatment of undefined expressions in *CML* can be given.

There are more general questions to be answered. Can every treatment of undefinedness be included in the unifying theory? What about the following? (i) the Alloy paradigm, where there is no function application; (ii) the logic of LCF, where quantifiers also range over undefined values; (iii) second-order undefinedness; (iv) logics with more than three values.

Chapter 7

Operational Semantics for ***CML1***

7.1 Introduction

In this chapter an initial draft of the core operational semantics for *CML* is presented for discussion. *CML* contains abstract operators to express high-level specifications, and concrete operators for expressing low-level specifications in a form close to their implementation. This initial version of the operational semantics addresses all the kernel features in the action language: the fundamental CSP operators and the basic manipulation of state. The latter includes the scoping of variables, assignment, and pre- and postcondition specifications. Many of the imperative and reactive language constructs that are yet to be added can be derived from the ones treated in this operational semantics, and many will be described in the table of correspondences included with deliverable D23.2. The next version of the operational semantics will include rules to deal with encapsulation of the action language in *CML* processes.

The semantics deals with a *CML* program text and its current state, which is an assignment to all the program variables in scope. This state is structured into global and local variables. So for example, when the program text is the parallel composition of two actions, each may have its own local state as well as there being a global state that persists beyond both their lifetimes. Program texts and states are treated uniformly as syntactic objects, so as well as providing transition rules for programs, normalisation rules for states are also provided.

The Chapter is structured as follows: Section 7.2 presents the syntax of the actions covered by the operational semantics, including reactive and imperative actions, as well as several pseudo-actions used in the operational rules to manage local state. Section 7.3 describes the representation of state information and normalisation rules. Section 7.4 introduces the semantic domain for the operational semantics. Finally, Section 7.5 presents the annotated transition rules.

CSP Operators	
`A := a -> A`	event prefix
`a.e -> A`	Synchronisation
`a!e -> A`	output
`a?x:T -> A`	input
`A |~| A`	internal choice
`A [] A`	external choice
`p & A`	guard
`A <| p |> A`	conditional
`A [| x1	cs	x2 |] A`	parallel composition
`A \ cs`	hiding
`A ; A`	sequential composition
`mu` `X @ A`	recursion
`X`	recursive reference
`A`	unparametrised action reference
`A(e)`	parametrised action reference
`var` `x:T @ A`	variable block
`SKIP`	termination
`STOP`	deadlock
Persistent State Actions	
`x := e`	assignment
`x: [P, Q]`	specification statement
Pseudo Operators	
`loc` `x @ A`	local state
`A [+] A`	external choice manager
`A [+| x1	cs	x2 |+] A`	parallel composition manager
`end` `x`	end scope
`ret`	return to scope

Table 7.1: Syntactic Overview

7.2 Syntax

The operational semantics is based on a portion of the syntax of *CML* extended with five additional pseudo-operators. It consists of three levels:

- The operators from CSP that apply to actions with non-persistent state.
- Additional operators that apply to actions with persistent state (heterogeneous actions involving components of VDM and CSP).
- Five pseudo-operators, which are not available to the specifier, but are used in the operational semantics to give meaning to the operators that are available to the specifier.

An summary of the syntax for the three levels of operators is given in Table 7.1. Further explanation of the CSP syntax and persistent state actions can be found in [34, Section 15]. The pseudo operators follow the same syntactic style.

7.3 State

7.3.1 Representation

State is encoded as a UTP relation with an empty input alphabet. These UTP relations are represented syntactically, so that they may readily be embedded into action definitions (see, e.g., the **loc** operator, below). They adhere to the following syntax:

$$\mathit{GroundAssignment} \quad ::= \quad \mathtt{v} \ \mathtt{:=} \ \mathtt{k}$$

where $\mathtt{v}$ is a possibly empty, comma-separated list of $n \geq 0$ variable names drawn from $\mathcal{V}$ (the set of variables), with the name $\mathtt{v\_i} \neq \mathtt{v\_j}$ for all $i \neq j$; $\mathtt{k}$ is a comma-separated list of n constants drawn from $\mathcal{K}$ (the set of ground terms). The length of these lists is explained when it is important.

$$\begin{array}{lll} \mathit{Assignment} ::= \mathtt{v} \ \mathtt{:=} \ \mathtt{e} & & \text{(Assignment)} \\ \qquad\qquad\quad \mid \ \mathbf{var} \ \mathtt{v} \ \mathtt{:=} \ \mathtt{e} & & \text{(Assignment to new variables)} \end{array}$$

where e is a comma-separated list of expressions drawn from $\mathcal{E}$. Note, all ground terms k are also expressions ($\mathcal{K} \subset \mathcal{E}$).

$$\begin{array}{lll} \mathit{Relation} ::= & \mathit{GroundAssignment} & \\ \mid & \mathit{Relation} \ \mathtt{;} \ \mathit{Assignment} & \text{(Sequential composition)} \\ \mid & \mathit{Relation} \ \mathtt{;} \ \mathbf{var} \ \mathtt{v} & \text{(Seq. comp. with variable introduction)} \\ \mid & \mathit{Relation} \ \mathtt{;} \ \mathbf{end} \ \mathtt{v} & \text{(End of Scope)} \\ \mid & \mathit{Relation} \restriction \mathtt{v} & \text{(Frame Restriction)} \\ \mid & \mathit{Relation} \ \mathtt{||} \ \mathit{Relation} & \text{(Frame Merge)} \end{array}$$

The following notation is used. Variables in program texts are denoted thus: $\mathtt{x}$. The corresponding semantic variable is denoted by changing the font, thus: x. These variables may actually be lists of variables, so the assignment $\mathtt{x} \ \mathtt{:=} \ \mathtt{e}$ is the multiple pairwise assignment of the elements of the vector of expressions $\mathtt{e}$ to the elements of the vector of variables $\mathtt{x}$. If $\mathtt{s}$ is an assignment, then the assumption is that it is ordered alphabetically.

7.3.2 Normalisation

In the ground semantics, state relations have a normal form corresponding to an ordered ground assignment. A set of reduction rules define a non-labelled transition system for reducing each state expression into normal form, and this is then used to define the main labelled transition system for the operational semantics. Normalisation is denoted by $\rightsquigarrow$, and corresponds to a homogeneous function on *Relation*.

End of Scope

$$\frac{\mathtt{w} = \mathtt{v} \setminus \{\mathtt{y}\} \qquad \forall i \bullet \mathtt{w\_i} = \mathtt{v\_i} \Rightarrow \mathtt{l\_i} = \mathtt{k\_i}}{\mathtt{v} \ \mathtt{:=} \ \mathtt{k} \ \mathtt{;} \ \mathbf{end} \ \mathtt{y} \ \rightsquigarrow \ \mathtt{w} \ \mathtt{:=} \ \mathtt{l}}$$

The end of scope rule removes a single variable from the state. The antecedents define the set of variables of the new assignment as the existing set minus the variable to be removed, define an ordered version of this set, and match the ground terms to the ordered variables. The consequent states that this new assignment is the normalised form of the end of scope operation.

Sequential composition with variable introduction

$$\frac{}{\texttt{v := k ; }\mathbf{var}\texttt{ w} \rightsquigarrow \texttt{v, w := k, k\_bot}}$$

This rule extends the state representation by adding fresh variables. The new variables are assigned to an arbitrary but fixed ground term `k_bot`. The operational rules never introduce variables without assigning to them (according to expression on variables in the existing state) within the same transition, so the `k_bot` values merely serve as temporary placeholders for those values. The antecedent defines the normalised extended state as `t` where `k_bot`, …, `k_bot` represents a sequence of m `k_bot`s.

Sequential composition with assignment to new variables

$$\frac{}{\texttt{s; }\mathbf{var}\texttt{ w := e} \rightsquigarrow \texttt{s; }\mathbf{var}\texttt{ w; w := e}}$$

The above rule defines variable introduction with assignment in terms of variable introduction followed by assignment. `s` corresponds to a *GroundAssignment* (i.e. normalised *Relation*).

Sequential Composition For a relation sequentially composed with an assignment the following rule applies:

$$\frac{\begin{array}{c}\forall\, \texttt{i} \bullet \texttt{k\_e\_i} = \texttt{e\_i}[\texttt{k\_1}, \ldots, \texttt{k\_n}/\texttt{v\_1}, \ldots, \texttt{v\_n}] \\ \forall\, \texttt{i} \bullet (\texttt{v\_i} = \texttt{w\_i} \Rightarrow \texttt{l\_i} = \texttt{k\_e\_i}) \wedge (\texttt{v\_i} \neq \texttt{w\_i} \Rightarrow \texttt{l\_i} = \texttt{k\_i})\end{array}}{\texttt{v := k ; w := e} \rightsquigarrow \texttt{v := l}}$$

The rule has two antecedents. The first antecedent evaluates the expressions in the vector `e`. The second overrides the assignment `v := k` appropriately.

Frame restriction

$$\frac{\forall\, \texttt{i} \bullet \texttt{w\_i} = \texttt{v\_i} \Rightarrow \texttt{l\_i} = \texttt{k\_i}}{\texttt{(v := k)} \upharpoonright \texttt{w} \rightsquigarrow \texttt{w := l}}$$

Frame restriction removes the variables not present in the second argument while keeping the ground terms associated with the remaining variables the same.

Frame merge

$$\frac{\texttt{v} \cap \texttt{w} = \emptyset}{\texttt{v := k || w := l} \rightsquigarrow \texttt{v, w := k, l}}$$

Frame merge takes two disjoint state relations and constructs a corresponding relation over the variables of both.

7.4 Operational Semantics

7.4.1 Role of Normalisation

In the following rules, the state component is normalised whenever appropriate. Normalisation is not itself a transition.

7.4.2 Semantic Domain

The semantic domain is constructed from:

- $\mathcal{A}$, the set of events available to actions.
- $\mathcal{V}$, the set of variables.
- $\mathcal{K}$, the set of ground terms (values denoted by variables).
- *GroundAssignment*, the syntax of ground assignments (cf. Section 7.3).
- *Action*, the syntax of actions (cf. Table 7.1).

The set of atomic events $\mathcal{A}$ is extended to $\mathcal{A}^{\mathcal{K}}$ by the inclusion of parametrised events. $\mathcal{A}^{\mathcal{K}} = \mathcal{A} \cup (\mathcal{A} \times \mathcal{K})$, where $a.k$ denotes the pair (a, k). This is then extended to $\mathcal{A}^{\mathcal{K}}_{\tau}$ by the inclusion of the silent τ event ($\tau \notin \mathcal{A}$). $\mathcal{A}^{\mathcal{K}}_{\tau} = \mathcal{A}^{\mathcal{K}} \cup \{\tau\}$.

Node Space The nodes of the LTS are triples $(w, \mathtt{s}, \mathtt{A}) \in \Sigma$ where:

- $w \subseteq \mathcal{V}$, the set of persistent variables.
- $\mathtt{s} \in \mathit{GroundAssignment}$, the data state (referred to in the commentary as the state component).
- $\mathtt{A} \in \mathit{Action}$, the action the node relates to.

Labelled Transition System The semantics of the language is given by a labelled transition system (LTS). The LTS is a triple $(\sigma, \mathcal{T}, \sigma_0)$, where:

- $\sigma \subseteq \Sigma$ is the set of states.
- $\mathcal{T} \subseteq (\Sigma \times \mathcal{A}^{\mathcal{K}}_{\tau} \times \Sigma)$ are the transitions.
- $\sigma_0 \subseteq \Sigma$ is the set of potential initial states.

We write $(w1, \mathtt{s1}, \mathtt{A1}) \xrightarrow{\mathtt{l}} (w2, \mathtt{s2}, \mathtt{A2})$for $((w1, \mathtt{s1}, \mathtt{A1}), \mathtt{l}, (w2, \mathtt{s2}, \mathtt{A2})) \in \mathcal{T}$.

7.5 Semantic Rules

7.5.1 Operations

Assignment

$$\frac{}{(w, \texttt{s}, \texttt{x := e}) \xrightarrow{\tau} (w, \texttt{s ; x := e}, \texttt{SKIP})}$$

The assignment action evolves via a τ event into `SKIP`. The state component is updated by sequentially composing the assignment (and normalising).

Specification Non-diverge

$$\frac{\alpha_{out}(s) = \{x, y\} \qquad \vdash P \land x' = e \land y' = y \Rightarrow Q}{(w, \texttt{s}, \texttt{x:[P, Q]}) \xrightarrow{\tau} (w, \texttt{s ; x := e}, \texttt{SKIP})}$$

The specification statement action evolves via a τ event into `SKIP`. The state component is updated by selecting an assignment that refines the specification statement and composing it with the source state (and normalising). The first antecedent introduces y, the difference between the program variable alphabet and the frame. The second antecedent establishes that the assignment $x := e$ is indeed a valid refinement of the specification statement.

Specification Diverge

$$\frac{\vdash s \Rightarrow \neg\, P}{(w, \texttt{s}, \texttt{x:[P, Q]}) \xrightarrow{\tau} (w, \texttt{s}, \texttt{x:[P, Q]})}$$

Specification statement actions can also diverge when their precondition does not hold in the source state component (the antecedent of the above rule). The transition below the line is a reflexive τ transition, characterising divergence.

7.5.2 Synchronisation and Communication

Simple Prefix

$$\frac{}{(w, \texttt{s}, \texttt{d -> A}) \xrightarrow{\texttt{d}} (w, \texttt{s}, \texttt{A})}$$

The above rule defines how an action prefixed with a simple event evolves, via that event, into the action it prefixes. The state component is unchanged.

Synchronisation

$$\frac{\alpha_{out}(\mathtt{s}) = v \quad \vdash s \Rightarrow (k = e[v'/v])}{(w, \mathtt{s}, \mathtt{d.\ e\ \text{->}\ A}) \xrightarrow{\mathtt{d.k}} (w, \mathtt{s}, \mathtt{A})}$$

The event's expression parameter is evaluated in the source state component of the transition. The first antecedent establishes the variables of the state component; the second evaluates the expression as a ground term. The consequent states that the action evolves via the ground-term parametrised event into the prefixed action. The state remains unchanged.

Output

$$\frac{(w, \mathtt{s}, \mathtt{d.\ e\ \text{->}\ A}) \xrightarrow{\mathtt{d.k}} (w, \mathtt{s}, \mathtt{A})}{(w, \mathtt{s}, \mathtt{d!e\ \text{->}\ A}) \xrightarrow{\mathtt{d.k}} (w, \mathtt{s}, \mathtt{A})}$$

The output event prefix is a synonym for the parametrised event prefix.

Input

$$\frac{k \in T}{(w, \mathtt{s}, \mathtt{d?\ x:\ T\ \text{->}\ A}) \xrightarrow{\mathtt{d.k}} (w, \mathtt{s}\ \mathtt{;}\ \mathbf{var}\ \mathtt{x\ :=\ k, A}\ \mathtt{;}\ \mathbf{end}\ \mathtt{x})}$$

An input prefix `d? x: T -> A` may make any transition `d. k`, where $k \in T$, the type of the channel.

7.5.3 Internal Choice

$$\frac{}{(w, \mathtt{s}, \mathtt{A1\ |\sim|\ A2}) \xrightarrow{\tau} (w, \mathtt{s}, \mathtt{A1})}$$

$$\frac{}{(w, \mathtt{s}, \mathtt{A1\ |\sim|\ A2}) \xrightarrow{\tau} (w, \mathtt{s}, \mathtt{A2})}$$

An internal choice between two actions can evolve via a τ event into either of them.

7.5.4 External Choice

The operational semantics of the external choice `A1 [] A2` runs both actions in parallel until there is an externally observable way of distinguishing between them, by observing one terminating or by observing one of them engaging in an event. The rules use a special syntax to denote the fact that a choice is pending `A1 [+] A2` (extra choice), and that `A1` and `A2` are running in parallel.

External Choice Begin

$$\frac{}{(w, \texttt{s}, \texttt{A1 [] A2}) \xrightarrow{\tau} (w, \texttt{s}, \texttt{(}\mathbf{loc}\ \texttt{s @ A1) [+] (}\mathbf{loc}\ \texttt{s @ A2))})}$$

An external choice between two actions evolves via a τ event into the `[+]` operator, which characterises the behaviour of the external choice on local states until the choice is resolved. Each action initially carries a local copy of the source state component `s` as represented by the **loc** `s` prefix.

External Choice Silent

$$\frac{(w, \texttt{s}, \texttt{A1}) \xrightarrow{\tau} (w, \texttt{s}, \texttt{A3})}{(w, \texttt{s}, \texttt{A1 [+] A2}) \xrightarrow{\tau} (w, \texttt{s}, \texttt{A3 [+] A2})}$$

$$\frac{(w, \texttt{s}, \texttt{A2}) \xrightarrow{\tau} (w, \texttt{s}, \texttt{A3})}{(w, \texttt{s}, \texttt{A1 [+] A2}) \xrightarrow{\tau} (w, \texttt{s}, \texttt{A1 [+] A3})}$$

The above two rules represent the two scenarios in which a `[+]` can carry out a τ event. There are two rules because external choice (and therefore `[+]`) commutes. The silent τ event is available when each action (including its implicit local state) can carry out a τ event, for example to resolve its own internal choice. The resulting τ transition on the composite allows the action carrying out the τ to evolve while the other remains as it is.

External Choice SKIP

$$\frac{}{(w, \texttt{s}, \texttt{(}\mathbf{loc}\ \texttt{s1 @ SKIP) [+] A}) \xrightarrow{\tau} (w, \texttt{s1}, \texttt{SKIP})}$$

$$\frac{}{(w, \texttt{s}, \texttt{A [+] (}\mathbf{loc}\ \texttt{s1 @ SKIP))}) \xrightarrow{\tau} (w, \texttt{s1}, \texttt{SKIP})}$$

The above two rules represent the cases where one of the actions in the external choice terminates. If one action has evolved into `SKIP` (with its accompanying local state **loc** `s1`), then the choice as a whole can evolve via a τ event into `SKIP`. The target state component is `s1`, the local state of the action that terminated.

External Choice End

$$\frac{(w, \texttt{s1}, \texttt{A1}) \xrightarrow{\texttt{l}} (w, \texttt{s3}, \texttt{A3}) \quad \texttt{l} \neq \tau}{(w, \texttt{s}, \texttt{(}\mathbf{loc}\ \texttt{s1 @ A1) [+] A2}) \xrightarrow{\texttt{l}} (w, \texttt{s3}, \texttt{A3})}$$

$$\frac{(w, \texttt{s2}, \texttt{A2}) \xrightarrow{\texttt{l}} (w, \texttt{s3}, \texttt{A3}) \quad \texttt{l} \neq \tau}{(w, \texttt{s}, \texttt{A1 [+] (}\mathbf{loc}\ \texttt{s2 @ A2))}) \xrightarrow{l} (w, \texttt{s3}, \texttt{A3})}$$

The above two rules represent the cases where an external choice is resolved by one of the actions carrying out an observable event. If one action can carry out an l and $l \neq \tau$ (i.e., observable) to an action `A3` then the composition as a whole can evolve via l into

A3. If the action updates its local state to s3 whilst taking the event l, then the target state component of resolved external choice is s3.

7.5.5 State-based Choice

Guard

$$\frac{\alpha_{out}(\mathtt{s}) = v \quad \vdash s \Rightarrow p[v'/v]}{(w, \mathtt{s}, \mathtt{p\ \&\ A}) \xrightarrow{\tau} (w, \mathtt{s}, \mathtt{A})}$$

Guarded actions are stuck, unless the guard is true.

Conditional

$$\frac{\alpha_{out}(\mathtt{s}) = v \quad \vdash s \Rightarrow p[v'/v]}{(w, \mathtt{s}, \mathtt{A1 <| p |> A2}) \xrightarrow{\tau} (w, \mathtt{s1}, \mathtt{A1})}$$

$$\frac{\alpha_{out}(\mathtt{s}) = v \quad \vdash s \Rightarrow \neg\, p[v'/v]}{(w, \mathtt{s}, \mathtt{A1 <| p |> A2}) \xrightarrow{l} (w, \mathtt{s}, \mathtt{A2})}$$

The rules for executing a conditional rely on evaluating the condition: if it is true, then the first action is selected; otherwise, the second is selected.

7.5.6 Sequential Composition

(i) Seq-comp-progress

$$\frac{(w, \mathtt{s1}, \mathtt{A1}) \xrightarrow{\mathtt{l}} (w, \mathtt{s3}, \mathtt{A3})}{(w, \mathtt{s1}, \mathtt{A1\ ;\ A2}) \xrightarrow{\mathtt{l}} (w, \mathtt{s3}, \mathtt{A3\ ;\ A2})}$$

The seq-com-progress rule describes the case where the first argument of the sequential composition has not terminated. The antecedent says the first action is capable of a transition and update in state component. The consequent says the sequential composition is capable of the same transition and update, but the second component remains sequentially composed and unchanged.

(ii) Seq-comp-skip

$$\frac{}{(w, \mathtt{s}, \mathtt{SKIP\ ;\ A}) \xrightarrow{\tau} (w, \mathtt{s}, \mathtt{A})}$$

This rule deals with the case where the first component of a sequential composition has terminated, i.e., has evolved into SKIP. It says that a SKIP composed with A evolves silently into A, whilst leaving the state component unchanged.

7.5.7 Parallel Composition

In the parallel composition `A1 [| x1 | cs | x2 |] A2`), the actions `A1` and `A2` are run in parallel. The program state is partitioned between the two actions, so that `A1` has exclusive access to the variables `x1` and `A2` has exclusive access to the variables `x2`. The two actions have to synchronise on all events in the channel set `cs`. A new operator (extra parallel) is used to keep track of the evolution of the partitioned state: `(`**`loc`**` s1 `**`@`**` SKIP) [+| x1 | cs | x2 |+] (`**`loc`**` s2 `**`@`**` SKIP)`.

Parallel Begin

$$\frac{\langle \mathtt{x1}, \mathtt{x2}\rangle \text{ partition } \alpha_{out}(\mathtt{s})}{\begin{array}{l}(w, \mathtt{s}, \mathtt{A1\ [|\ x1\ |\ cs\ |\ x2\ |]\ A2}) \\ \quad \xrightarrow{\tau} \\ \qquad (w, \mathtt{s}, \mathtt{(}\,\mathbf{loc}\,(\mathtt{s} \upharpoonright \mathtt{x1})\ \mathbf{@}\ \mathtt{A1)\ [+|\ x1\ |\ cs\ |\ x2\ |+]\ (}\,\mathbf{loc}\,(\mathtt{s} \upharpoonright \mathtt{x2})\ \mathbf{@}\ \mathtt{A2))}\end{array}}$$

The parallel composition of two actions is dealt with initially by the parallel-begin rule. The rule takes the operator of the action syntax (`[| ... |]`), and evolves silently into the operator (`[+| ... |+]`) of the extended action syntax, which manages the essential behaviour of parallel composition. The silent transition from one operator to the other introduces a local state for each parallel action, which is the source state component restricted to `x1` (respectively `x2`).)

Parallel Non-synchronisation

$$\frac{(w, \mathtt{s}, \mathtt{A1}) \xrightarrow{\mathtt{l}} (w, \mathtt{s}, \mathtt{A3}) \quad l \notin cs}{(w, \mathtt{s}, \mathtt{A1\ [+|\ x1\ |\ cs\ |\ x2\ |+]\ A2}) \xrightarrow{\mathtt{l}} (w, \mathtt{s}, \mathtt{A3\ [+|\ x1\ |\ cs\ |\ x2\ |+]\ A2})}$$

$$\frac{(w, \mathtt{s}, \mathtt{A2}) \xrightarrow{\mathtt{l}} (w, \mathtt{s}, \mathtt{A3}) \quad l \notin cs}{(w, \mathtt{s}, \mathtt{A1\ [+|\ x1\ |\ cs\ |\ x2\ |+]\ A2}) \xrightarrow{\mathtt{l}} (w, \mathtt{s}, \mathtt{A2\ [+|\ x1\ |\ cs\ |\ x2\ |+]\ A3})}$$

The above two rules represent the cases where one of the parallel actions can proceed independently of the other. For this to happen, the event carried out by the action must not be in the synchronising set, the second argument `cs` accompanying the operator. The antecedents state that the left-hand (respectively right-hand) action is capable of such an event. The consequent allows the transition to proceed independently. Implicitly the actions `A1`, `A2`, and `A3` have local state (cf. **`loc`**) as guaranteed by the parallel-begin rule.

Parallel Synchronisation

$$\frac{(w, \mathtt{s}, \mathtt{A1}) \xrightarrow{\mathtt{l}} (w, \mathtt{s}, \mathtt{A3}) \quad (w, \mathtt{s}, \mathtt{A2}) \xrightarrow{\mathtt{l}} (w, \mathtt{s}, \mathtt{A4}) \quad l \in \mathtt{cs}}{(w, \mathtt{s}, \mathtt{A1\ [+|\ x1\ |\ cs\ |\ x2\ |+]\ A2}) \xrightarrow{l} (w, \mathtt{s}, \mathtt{A3\ [+|\ x1\ |\ cs\ |\ x2\ |+]\ A4})}$$

If both actions in a parallel composition are capable of the same event within the synchronising set `cs`, then they can synchronise.

Parallel End

$$\frac{}{\begin{array}{l}(w, \mathtt{s}, (\mathbf{loc}\ \mathtt{s1}\ \mathtt{@}\ \mathtt{SKIP})\ \mathtt{[+|\ x1\ |\ cs\ |\ x2\ |+]}\ (\mathbf{loc}\ \mathtt{s2}\ \mathtt{@}\ \mathtt{SKIP}))\\ \quad\xrightarrow{\tau}\\ \qquad (w, \mathtt{s1\ ||\ s2}, \mathtt{SKIP})\end{array}}$$

The parallel-end rule deals with the case where both parallel actions have terminated, i.e., evolved into `SKIP` on their own local state spaces. The consequent describes the transition which constructs the resulting target state component from the actions' local states. This is defined as the merge of the left hand action (respectively the right hand action) restricted to `x1` (respectively `x2`).

7.5.8 Hiding

Hiding Silent

$$\frac{(w, \mathtt{s1}, \mathtt{A1}) \xrightarrow{\tau} (w, \mathtt{s2}, \mathtt{A2})}{(w, \mathtt{s1}, \mathtt{A1} \setminus \mathtt{cs}) \xrightarrow{\tau} (w, \mathtt{s2}, \mathtt{A2} \setminus \mathtt{cs})}$$

An action with hidden events can make its own internal progress.

$$\frac{(w, \mathtt{s1}, \mathtt{A1}) \xrightarrow{\mathtt{l}} (w, \mathtt{s2}, \mathtt{A2}) \quad l \in cs}{(w, \mathtt{s1}, \mathtt{A1} \setminus \mathtt{cs}) \xrightarrow{\tau} (w, \mathtt{s2}, \mathtt{A2} \setminus \mathtt{cs})}$$

An action with hidden events will make internal progress if the underlying action could engage in any of the hidden events.

Hiding Non-silent

$$\frac{(w, \mathtt{s1}, \mathtt{A1}) \xrightarrow{\mathtt{l}} (w, \mathtt{s2}, \mathtt{A2}) \quad l \notin cs}{(w, \mathtt{s1}, \mathtt{A1} \setminus \mathtt{cs}) \xrightarrow{\mathtt{l}} (w, \mathtt{s2}, \mathtt{A2} \setminus \mathtt{cs})}$$

An action with hidden events will make visible progress if it can engage in a event that is not being hidden.

Hiding End

$$\frac{}{(w, \mathtt{s1}, \mathtt{SKIP} \setminus \mathtt{cs}) \xrightarrow{\tau} (w, \mathtt{s2}, \mathtt{SKIP})}$$

The hiding operator can be disposed of once its operand action terminates.

7.5.9 Recursion

$$\frac{}{(w, \mathtt{s}, \mathbf{mu}\,\mathtt{X}\ \mathbf{@}\ \mathtt{P}) \xrightarrow{\tau} (w, \mathtt{s}, \mathtt{P}\,[\,\mathbf{mu}\,\mathtt{X}\ \mathbf{@}\ \mathtt{P}\ /\ \mathtt{X}\,])}$$

The recursion rule involves substitution in the syntax of the action. Whenever **mu** X **@** P is encountered, the Xs in P are replaced by **mu** X **@** P. This replacement is achieved via a silent τ transition.

7.5.10 Action Reference

Action Reference

$$\frac{\mathtt{P} = \mathtt{A}}{(w, \mathtt{s}, \mathtt{P}) \xrightarrow{\tau} (w, \mathtt{s}, (\,\mathbf{loc}\ (\mathtt{s} \upharpoonright w)\ \mathbf{@}\ \mathtt{A}\ \mathbf{;}\ \mathbf{ret}\,))}$$

Action reference applies when an action refers to an action definition. The first case is when the rule carries no parameters and is described by the above rule. The antecedent says that the defining equation P = A exists in the action environment. The consequent says the reference P can evolve silently into its definition A; however, there is some additional complexity introduced by variable scoping. The action A is given its own local scope, which disregards the non-persistent variables present in the source state component. In addition A is sequentially composed with **ret**, which restores the variable scope of the referring action when A terminates.

Action Reference Parametrised

$$\frac{y = x \setminus \alpha_{out}(s) \qquad z = \{w, x\} \qquad \mathtt{P\,(x)\ = A}}{(w, \mathtt{s}, \mathtt{P\,(e)}) \xrightarrow{\tau} (w, \mathtt{s}, (\,\mathbf{loc}\ ((\mathtt{s}\ \mathbf{;}\ \mathbf{var}\ \mathtt{y}\ \mathbf{;}\ \mathtt{x} := \mathtt{e}) \upharpoonright \mathtt{z})\ \mathbf{@}\ \mathtt{A}\ \mathbf{;}\ \mathbf{ret}\,))}$$

The second rule handles the case where the action reference is parametrised by a list of expressions e. The antecedent states that the reference must have a defining equation with x as parameter. The consequent says that the reference evolves silently into its definition; however, as with the simple action reference, there are additional complexities introduced by variable scoping. The definition A operates on its own local state, and is composed with **ret** in order to destroy the local scope on A's termination. The initial local state is constructed by the following logic: first the variables in x not already in s are introduced, next the variables of x are assigned the value e, and finally the resulting state is restricted to the variables in the persistent state w and x.

7.5.11 Variable Scoping

Local Progress

$$\frac{(w, \mathtt{s1}, \mathtt{P1}) \xrightarrow{\mathtt{l}} (w, \mathtt{s2}, \mathtt{P2})}{(w, \mathtt{s}, (\,\mathbf{loc}\ \mathtt{s1}\ \mathbf{@}\ \mathtt{P1}\,)) \xrightarrow{\mathtt{l}} (w, \mathtt{s}, (\,\mathbf{loc}\ \mathtt{s2}\ \mathbf{@}\ \mathtt{P2}\,))}$$

This rule governs progress on local state (cf. the **loc** operator). If an action can perform an event from source state component to target state component, then the action with local state can perform the equivalent transition operating on its local state.

Local Return

$$\frac{}{(w, \mathtt{s1}, (\,\mathbf{loc}\ \mathtt{s2}\ \mathbf{@}\ \mathbf{ret})) \xrightarrow{\tau} (w, \mathtt{s1}, \mathtt{SKIP})}$$

This rule deals with the case when a local state action (i.e., an action reference) has evolved into **ret**. Such an action evolves silently into SKIP destroying the local state.

End Scope

$$\frac{}{(w, \mathtt{s}, \mathbf{end}\ \mathtt{x}) \xrightarrow{\tau} (w, \mathtt{s}\ \mathbf{;}\ \mathbf{end}\ \mathtt{x}, \mathtt{SKIP})}$$

This rule handles the **end** x action, which is introduced by input events and variable blocks. It evolves silently into SKIP and applies the **end** x relation to the state component, removing it from scope.

7.5.12 Variable Block

$$\frac{k \in T}{(w, \mathtt{s}, \mathbf{var}\ \mathtt{x}\mathbf{:}\ \mathtt{T}\ \mathbf{@}\ \mathtt{P}\ \mathbf{end}\ ) \xrightarrow{\tau} (w, \mathtt{s}\ \mathbf{;}\ \mathbf{var}\ \mathtt{x}\ \mathbf{:=}\ \mathtt{k}, \mathtt{P}\ \mathbf{;}\ \mathbf{end}\ \mathtt{x})}$$

For variable blocks the action evolves silently into the action surrounded by the block, with the state component is enriched by the new variable. The antecedent and updated state stipulate that the new variable takes any value in its declared type. In order to remove the variable from scope on termination, the action is sequentially composed with **end** x.

Chapter 8

Overview of OO Copy Semantics

This Chapter describes a strategy for incorporating the object-oriented features of the Compass Modelling Language (*CML*) into the semantic framework currently under development. The discussion focuses on the "copy" semantics, in which object state is accessed and communicated by value rather than reference.

8.1 Introduction

The purpose of this discussion is to describe a strategy for incorporating the object-oriented (OO) features of the Compass Modelling language (*CML*) into the semantic framework under development. The OO concepts are currently defined syntactically [34], and include forms for standard notions such as encapsulation, protected access, inheritance, aggregation, etc. The emerging semantic framework has developed a core semantic theory for *CML* based on Unifying Theories of Programming, integrating state-based specification in VDM and timed CSP processes, based on Lowe and Ouaknine's model of time processes[19]. Tables of correspondence in Chapter 5 describe how a subset of *CML* syntax can be expressed equivalently in terms of the primitives for which the semantics is defined. The table does not address the macro-structure of *CML* specifications, nor the way OO concepts are mapped into the semantic foundations.

Full exploitation of the facilities provided by OO requires a further set of considerations, which are summarised below. Key to this exploitation is the identification and streamlining of specific patterns of refinement that only become evident given OO's rich structure.

The general COMPASS strategy for addressing the OO issue is in two stages: the first objective is to provide a copy (value-based) semantics, the second is to extend this to a reference-based semantics. The difference is that the latter allows access and communication of objects by reference (name), providing greater genericity with respect to target implementation languages. The remainder of the Chapter concentrates on the former.

The discussion is structured as follows: Section 8.1 describes the background to the problem, introducing *OhCircus* [5], which is the most important source of prior experience in this area; Section 8.3 explains its semantic approach in more detail. Section 8.4 considers the strategy for incorporating OO in *CML* in two parts. The first is based principally on the concepts and techniques of *OhCircus*, whilst the second looks at time, a feature of *CML* requiring additional consideration. Finally Section 8.5 summarises the discussion and concludes.

8.2 Background

The investigators working on the COMPASS language have prior experience of interpreting and exploiting OO concepts within a *CML*-related heterogeneous language. *OhCircus* is an object-oriented extension of *Circus*. Both are based on Z [29, 32], CSP [13], and the specification statements of the refinement calculus [21], and permit specification and refinement of system behaviour in terms of operations on data and inter-process communication. The former is further augmented with the means to encapsulate state and associated methods, along the recognised data-type constructors of OO, such as inheritance. Its meaning is given by a copy semantics.

In considering the proximity and applicability of the *OhCircus* approach to what is required for *CML*, it is first necessary to compare the languages of *OhCircus* and *CML*. The general similarities are evident, and have already been mentioned above. Additionally, the two languages do not differ significantly in the OO constructs they support. For example:

- *Class Definition:* attributes, methods, etc [34, Section 4].
- *Qualifiers:* `public`, `private`, `protected`, `logical` [34, Section 11].
- *Inheritance:* `extends` clause [34, Section 4].
- *Call/Access mechanism:* e.g.,

  ```
  object designator . name ( [ expression list ])
  ```

  [34, Section 15.3].
- *Constructors:* `initial` clause, the `new` statement [34, Section 4/15.5].

The differences between the languages can be summarised as follows:

- *Syntactic:* *CML* is based on the VDM language, whilst *OhCircus* is based on Z. However, whilst Z and VDM differ in style, they both offer the same essential facilities and can be accommodated consistently within UTP semantics. The syntactic differences are largely superficial.
- *Semantic:* VDM and Z (and by extension *CML* and *OhCircus*) have some subtle semantic differences concerning the treatment of type and the logic of undefined expressions and predicates. Whilst these are relevant, they are being treated orthogonally within separate tasks. These differences do not affect the OO strategy significantly.
- *Syntactic and semantic:* Semantically, *CML* incorporates a model of *time*, and correspondingly the syntax offers various operators expressing temporal constructs,

e.g., timeout [34, Section 15]. This represents a significant difference from *OhCircus*, which does not incorporate temporal operators; however, the UTP is good at combining such diverse paradigms.

Given the proximity of the languages, and despite the differences outlined above, our view is that *OhCircus* provides a sound basis for the treatment of the OO constructs of *CML*. We proceed by describing *OhCircus*'s semantic approach in more detail.

8.3 Overview of *OhCircus*'s semantic approach

Semantically, there are four main elements to the *OhCircus* approach: semantic foundations, interpretation of call and access mechanisms, treatment of OO structural relationships, exploitation of structure in refinement. These are covered in more detail in the following.

8.3.1 Semantic foundations

The semantic foundations of *OhCircus* are similar to those of *Circus*. UTP's alphabetised relations are employed to characterise the behaviour of a specification. The alphabet includes the state variables of the system along with the auxiliary observation variables tr (to record process traces), ref (to record refusal sets), $wait$ (to indicate that a process is waiting for interaction with its environment) and ok (to indicate whether a process has reached a stable state).

OhCircus differs from *Circus* in that it needs to characterise class methods in order to give meaning to the behaviour of object instances. This is achieved by the use of higher-order UTP specifications (see [14, Chapter 9]). In higher-order UTP, variables may themselves take the value of programs or processes, and this is the means by which class methods are incorporated into the model.

Classes are represented by record-typed variables comprising of fields for: (i) the constructor function of the class; (ii) each method of the class. The method fields range over types based on alphabetised relations (higher-order values). These are defined on an alphabet derived from the attributes of the class.

A variable taking a class type in *OhCircus* is also represented semantically as a record type consisting of the object's attribute fields, i.e., state variables. State variables may themselves take object types, producing a hierarchical state structure.

The formulation preserves the standard operator interpretations from UTP, including refinement as reverse implication.

8.3.2 Call and Access mechanism

The attributes and methods of component objects are crucial in defining the behaviour of the object of which they are a constituent. To do this, the syntax and semantics of *OhCircus* needs to allow access mechanisms to inspect the internal state of component objects, as well as call mechanisms to update the internal state of component objects according to their method specifications. The call mechanisms must also provide a means of supplying input parameters and receiving outputs into the outer context.

Attribute access, e.g., $object\_o.x\_attribute$, where access is permitted, amounts to straightforward binding selection, i.e., reference to a variable (cf. $x\_attribute$) within the attributes field of the object in question (cf. $object\_o$). However, method calls require special consideration.

Each method in *OhCircus*, as it is represented in the semantic structure, is given generically as a function (lambda expression). Each function takes *values* representing the before state of an object instance, plus any possible input parameter, and variable *names* representing the after state of the object instance and any possible output parameter. The function yields a specific alphabetised relation corresponding to the method's specification, applied to the values supplied, updating the names supplied.

Component methods are used in specifying the behaviour of the object in which the components appear as attributes. Syntactically, this is based on the method invocation statement familiar in object-oriented programming, e.g., $object\_o.method\_m(params)$. To give meaning to such expressions, *OhCircus* accesses the function corresponding to the method (cf. $method\_m$) in the component object class's (cf. $object\_o$) methods field. Applying the function to the arguments in the statement (cf. o and $params$) instantiates the predicate for use in the outer (calling) context. Parameter fields (cf. $params$) may also include the names of any output variables defined by the method. For purposes of symbolic proof, each lambda expression simplifies via β-reduction into the instance required.

OhCircus additionally allows methods to be used as expressions (rather than predicates) using a modified form of the above mechanism. In order to use a method as an expression the method must be deterministic. In practice, proof obligations are required to govern the use of methods as expressions.

In addition to regular method calls, *OhCircus* also allows the use of the *new* operator (cf. $newobject\_o(i)$), to invoke an object's constructor method. The *new* operator behaves as an expression, and as such needs to be defined deterministically.

8.3.3 Structural relationships

A specification's structural relationships, such as class inheritance, are described by the denotational semantics. It defines, for example, how the semantic structure produced by a super class gives rise to the semantic structure corresponding to one of its subclasses. At the same time access control, such as where qualifiers (e.g., `private`) feature, act as domain restrictions on the denotational semantics, and give rise to conditions capable of being analysed statically.

OhCircus extends the primitives outlined above to deal with the additional considerations introduced by structure. For example, it permits access to the attributes and methods of an object's super-type (keyword `super`) as well as class type-casting. Both are familiar concepts in OO.

8.3.4 Refinement

One of the key contributions of the *OhCircus* approach is its treatment of refinement. The theory and laws of refinement extend those already established in *Circus*. *Circus* embraces a compositional philosophy, in which the operators and structural relationships are designed to be monotonic with respect to refinement. This streamlines verification effort by allowing component-wise refinement of operations, actions, and processes without the need to reverify the system as a whole.

OhCircus specifications have a rich structure based on OO, and this can be exploited for the purposes of refinement. Refinement and refactoring patterns can be identified based on structural relationships and formulated as laws. Each law expresses the conditions that need to be met for each refinement structurally consistent with the pattern to be valid. Such pattern-specific conditions in general reduce the burden of proof.

OhCircus identifies additional types of transformation in its extended refinement strategy:

- Class simulation, in addition to process simulation.
- Class refinement, in addition to process refinement.

Concrete examples of these strategies (from [5]) include:

- The refinement of a super-class specification by a sub-class specification that extends it. The example shows how refinement capability adds a complementary dimension to the standard OO notion of inheritance, and serves as a powerful tool in systems development.
- The partitioning of an active object specification (with associated behaviour) into two concurrent specifications, one of which is a new active object defined by a new class and behaviour. This transformation is related to the class extraction refactoring technique seen within object-oriented development. However, it applies more generally, at the specification level, permitting behaviour to be refined as well as refactored.

8.4 Strategy for OO in *CML*

A two-part strategy is proposed for extending the current semantics of *CML* to incorporate the object-oriented elements of the language. The first part looks at the features *OhCircus* and *CML* have in common and considers how the concepts of *OhCircus* transfer into *CML*. The second considers time. Time, and timing operators, are features of *CML* which are not present in *OhCircus*, and therefore they need to be considered independently.

8.4.1 Application of concepts from *OhCircus*

The semantics of *CML* is based on the timed testing model of Lowe and Ouaknine [19]. In *CML*, behaviour is represented by trace information comprising actions, refusals, and the passage of time (the *tock*) event. This contrasts with the approach of *OhCircus*, which has no explicit representation of time, and which is based on the standard failures/divergences model of CSP. The difference is reflected in the UTP auxiliary variables present in each model, with the former being based on tr and ok, and the latter additionally including $wait$ and ref. The semantic models of *CML* and *OhCircus* both incorporate state variables, (i.e., the data being manipulated at component/system level). The current *CML* semantics is abstract with respect to the types state variables can take, although in practice this will be instantiated based on the type system of VDM. *OhCircus*, in contrast, has a more concrete notion of type. This is partly necessitated by the underlying model-based specification language (cf. Z), and partly due to the need to incorporate the semantics of the OO constructs. The types necessary for OO are based on binding types to characterise object instances, and high-order (predicate) functions to capture classes and their methods. Finally, *CML* defines the primitive operators necessary to express the denotational semantic function (the mapping from syntax into semantic model). However the structural elements of this function, for example how paragraphs give rise to environments, and how environments guide the application of operators, are as yet undefined. In *OhCircus* the denotational semantic function is embellished—compared to that of circus—with the additional apparatus required to interpret and enforce consistency of the structures of OO.

Comparing the current semantic models of *CML* and *OhCircus* it can be seen that they are on the whole complementary. Considering the semantic elements, e.g., type and denotational function, which are heavily influenced in *OhCircus* by the complexities introduced by OO, *CML* remains largely uncommitted at this stage. The two issues key to the successful transfer of approach are: (i) how the differences in behavioural model between *OhCircus* and *CML* impact on the handling of OO; and (ii) whether the approach of *OhCircus* requires anything specifically of *CML*'s semantic model that may be difficult to accommodate.

The behavioural model issue is investigated further in the next section; however, it is important to note that the complications introduced by OO for the most part concern data state and operators on that state. This is evidenced by the semantic mechanisms introduced by necessity in *OhCircus* to capture the meaning of classes via their methods and objects via their attribute hierarchy. Processes can be influenced by the OO structure (cf. inheritance), but the effects on processes are not new in themselves, they just need adapting to the timed behavioural model of *CML*.

In addressing the additional demands OO places on *CML*'s semantic model it is necessary to consider the *CML* semantics as instantiated for the VDM type and value system. In particular, it is interesting to consider how compatible *CML* is with the mechanisms introduced in *OhCircus* and whether any recasting of these mechanisms will be necessary. Again there are two issues: (i) whether the *CML* type and value system has the capability to define and manipulate the record types needed to represent attribute hierarchy; (ii) whether the higher-order functions used to capture class semantics present any specific problems within *CML*.

Record Types Examining *CML0* shows that there is already provision for composite types [34, p.28] comprising fields, field selection [34, p.55], and record expressions [34, p.55] which are used to construct records. To conclude, the use of record types presents no problems.

High Order Functions The purpose of each higher-order function is to instantiate a predicate representing the class's method application for use in context it is required (i.e., object, input, output). Each function yields a predicate operating on a record type (schema type) constraining the after-state attributes of the class. To achieve this, the method specification, which is defined over variables representing the individual attributes of the class, is projected onto the required record type—e.g., a specification on variables a and b is projected to a specification between records containing the fields a and b. One issue is whether this projection is expressible in high-order UTP based on ***CML***/VDM's type and value system. We would argue it is, based on the record expression construct, which is in essence close to Z's θS. The only difference is that field values in record expressions need to be listed explicitly and ordered, rather than by reference to a schema. However, this can be considered only a minor complication for ***CML***'s denotational semantic function.

Another issue concerns the explicit nature of preconditions in VDM, and therefore ***CML***. Method specifications in ***OhCircus*** are given by Z operation schemas, whose precondition is understood implicitly. In ***CML***, as in VDM, an operation is defined by a precondition/postcondition pair. In practice, proof obligations will ensure that such pairs $[P, Q]$ respect the *law of the excluded miracle*, and therefore that they convey the same information as the Z schema with the constraint $P \wedge Q$. Semantically, there is therefore good reason to suppose the explicit pre-post style can be accommodated faithfully within a semantic framework modelled on ***OhCircus***.

However, it is prudent to investigate the issues surrounding the use of explicit preconditions in an OO context further. For example, should reference to attribute method specifications be allowed within pre-conditions or ruled out statically? What is the precise meaning of such uses? What impact does the use of calls to method specifications within post-conditions have on the law of the excluded miracle? Does ***CML*** require high-order functions to yield the preconditions of class methods in order to address any issues that arise?

In considering the general application of ***OhCircus*** concepts to ***CML*** the approach is eminently viable. Moreover, basing the OO semantics of ***CML*** on ***OhCircus*** will permit reuse of the refinement concepts, laws and proofs carried out as part of the ***OhCircus*** project.

8.4.2 Consideration of temporal aspects

Whilst the process models of ***OhCircus*** and ***CML*** differ due to the introduction of time in the latter, its impact within OO is limited to one consideration, that of process inheritance. In ***OhCircus***, classes can be imbued with process specifications describing their active behaviour. When one class inherits a process specification from another class, as well as supplying its own process specification, the meaning is given by their parallel composition. This definition is based on the notion of substitutability (following Fischer and Wehrheim [8]). In other words, the inherited process constrains the actions of the inheriting process over their common alphabet.

The issue for ***CML*** is whether the same definition of process inheritance acts as a sufficiently sound basis for processes with time. There are strong indications that it does.

However it would be prudent to investigate this further, for example supplying a rigorous argument/proof for substitutability.

In addition, as proposed by the originators of *OhCircus*, it be would be of potential benefit to extend the theory of process inheritance. This might allow, for instance, a more direct expression of behaviour that does not require processes to be composed in parallel, based on the validity of simple side-conditions.

Finally, there is a strong foundation for investigating temporal issues further, based on other work within the circus family of languages. *CircusTime* [28] is an adaptation of *Circus* with time, albeit a slightly different model to *CML*. Speculative work as been conducted investigating the links between *CircusTime*, *Circus*, and *OhCircus*, based on composable Galois connections. Further focus in this area will reveal more about the structure of the relationships between the languages, which will in turn guide investigations into the points raised. In addition, these relationships ought to inform the process of identifying further refinement patterns and laws applicable due to the inclusion of time.

8.5 Summary

In summing up, it is clear that *OhCircus* provides a sound basis for the development of OO semantic extensions to *CML*. In addition, *OhCircus* provides a further valuable source of transferable concepts, especially in refinement patterns, but also in the laws and proofs that justify them. In practice, the *CML* semantic framework can be adapted for purpose by instantiation, i.e., by allowing variables to range over record types to record attribute hierarchies, and higher-order functions to represent class methods. This does not present technical challenges that are incompatible with the envisaged non-OO semantics.

Some further investigation is desirable, particularly in the areas of explicit pre-/postcondition handling and process inheritance. However, we do not believe that these issues constitute fundamental barriers to the adoption of the strategy.

Bibliography

[1] J.-R. Abrial. *The B-Book: Assigning Progams to Meanings*. Cambridge University Press, 1996.

[2] Sten Agerholm and Jacob Frost. An Isabelle-based theorem prover for VDM-SL. In Elsa L. Gunter and Amy P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, USA, August 19–22, 1997, Proceedings*, volume 1275 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1997.

[3] R. J. R. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[4] Andrew Butterfield, Pawel Gancarski, and Jim Woodcock. State visibility and communication in unifying theories of programming. In Wei-Ngan Chin and Shengchao Qin, editors, *TASE*, pages 47–54. IEEE Computer Society, 2009.

[5] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Unifying classes and processes. *Software and System Modeling*, 4(3):277–296, 2005.

[6] Ana Cavalcanti and Jim Woodcock. A tutorial introduction to CSP in Unifying Theories of Programming. In Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *Refinement Techniques in Software Engineering, First Pernambuco Summer School on Software Engineering, PSSE 2004, Recife, Brazil, November 23–December 5, 2004, Revised Lectures*, volume 3167 of *Lecture Notes in Computer Science*, pages 220–268. Springer, 2006.

[7] COMPASS consortium. Description of Work: Comprehensive Modelling for Advanced Systems of Systems. EU project 287829, 2011.

[8] Clemens Fischer and Heike Wehrheim. Behavioural subtyping relations for object-oriented formalisms. In Teodor Rus, editor, *Algebraic Methodology and Software Technology. 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20–27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 469–483. Springer, 2000.

[9] Michael Goldsmith. FDR2 user's manual. Technical Report Version 2.82, Formal Systems (Europe) Ltd, Jun. 2005.

[10] Michael J. C. Gordon, Robin Milner, and Christophe P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[11] Will Harwood, Ana Cavalcanti, and Jim Woodcock. A theory of pointers for the utp. In John S. Fitzgerald, Anne Elisabeth Haxthausen, and Hüsnü Yenigün, edi-

tors, *Theoretical Aspects of Computing - ICTAC 2008, 5th International Colloquium, Istanbul, Turkey, September 1-3, 2008. Proceedings*, volume 5160 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2008.

[12] Eric C. R. Hehner. Retrospective and prospective for Unifying Theories of Programming. In Steve Dunne and Bill Stoddart, editors, *Unifying Theories of Programming, First International Symposium, UTP 2006, Walworth Castle, February 5–7, 2006, Revised Selected Papers*, volume 4010 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2006.

[13] C. A. R. Hoare. *Communicating Sequential Processes.* Intl. Series in Computer Science. Prentice Hall, 1985.

[14] C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming.* Series in Computer Science. Prentice Hall, 1998.

[15] *Circus* homepage. `http://www.cs.york.ac.uk/circus/`, accessed September 28, 2012.

[16] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 1986.

[17] Peter Gorm Larsen, Nick Battle, Miguel Alexandre Ferreira, John S. Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture initiative: integrating tools for VDM. *ACM SIGSOFT Software Engineering Notes*, 35(1):1–6, 2010.

[18] Peter Gorm Larsen and Wieslaw Pawlowski. The formal semantics of iso vdm-sl, 1995.

[19] Gavin Lowe and Joël Ouaknine. On timed models and full abstraction. *Electr. Notes Theor. Comput. Sci.*, 155:497–519, 2006.

[20] Vienna Development Method. `en.wikipedia.org/wiki/Vienna_Development_Method`, accessed September 28, 2012.

[21] C. Morgan. *Programming from Specifications.* Prentice-Hall, 2nd edition, 1994.

[22] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[23] Information Standards Organisation. Information technology – programming languages, their environments and system software interfaces – vienna development method – specification language – part 1: Base language. ISO/Iec 13817-1:1996.

[24] Juan Perna and Jim Woodcock. Utp semantics for handel-c. In Andrew Butterfield, editor, *Unifying Theories of Programming*, volume 5713 of *Lecture Notes in Computer Science*, pages 142–160. Springer Berlin / Heidelberg, 2010.

[25] Communicating Sequential Processes. `en.wikipedia.org/wiki/Communicating_sequential_processes`, accessed September 28, 2012.

[26] Alan Rose. A lattice-theoretic characterisation of three-valued logic. *Journal of the London Mathematical Society*, 25:255–259, 1950.

[27] Mark Saaltink. The Z/EVES system. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer, 1997.

[28] Adnan Sherif and Jifeng He. Towards a time model for *Circus*. In Chris George and Huaikou Miao, editors, *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21–25, 2002, Proceedings*, volume 2495 of *Lecture Notes in Computer Science*, pages 613–624. Springer, 2002.

[29] J. Michael Spivey. *The Z Notation: A Reference Manual.* Series in Computer Science. Prentice Hall International, 2nd edition, 1992.

[30] Jim Woodcock and Ana Cavalcanti. The semantics of *circus*. In Didier Bert, Jonathan Bowen, Martin Henson, and Ken Robinson, editors, *ZB 2002:Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer Berlin / Heidelberg, 2002.

[31] Jim Woodcock and Ana Cavalcanti. A tutorial introduction to Designs in Unifying Theories of Programming. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4–7, 2004, Proceedings*, volume 2999 of *Lecture Notes in Computer Science*, pages 40–66. Springer, 2004.

[32] Jim Woodcock and Jim Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.

[33] Jim Woodcock and Leo Freitas. Linking VDM and Z. In Mike Hinchey, editor, *ICECCS*, pages 143–152. IEEE Computer Society, 2008.

[34] Jim Woodcock and Alvaro Miyazawa. CML Definition 0. Public Document. Deliverable Number: D23.1, Version: 1.0, COMPASS Project, University of York, June 2012.

[35] Naijun Zhan, Eun-Young Kang, and Zhiming Liu. Component publications and compositions. In Andrew Butterfield, editor, *UTP*, volume 5713 of *Lecture Notes in Computer Science*, pages 238–257. Springer, 2008.

[36] Huibiao Zhu, Fan Yang, and Jifeng He. Generating denotational semantics from algebraic semantics for event-driven system-level language. In Shengchao Qin, editor, *Unifying Theories of Programming*, volume 6445 of *Lecture Notes in Computer Science*, pages 286–308. Springer Berlin / Heidelberg, 2010.

Appendix A

Proofs

A.1 Well Foundedness

Theorem 4.2.1 (Well foundedness) *Every* CML *operator preserves* ***T1****-healthiness.*

Proof A.1.1 *By induction on syntax.*

1. ***Deadlock***

$$
\begin{array}{l}
STOP[\langle\rangle/tt'] \\
= \{\, STOP \,\} \\
(\textbf{\textit{T0}}(trace(tt') \in tock^*))[\langle\rangle/tt'] \\
= \{\, \textbf{\textit{T0}}\text{, substitution} \,\} \\
trace(\langle\rangle) \in tock^* \land \langle\rangle \in timedTrace \\
= \{\, \langle\rangle \in timedTrace \,\} \\
trace(\langle\rangle) \in tock^* \\
= \{\, trace(\langle\rangle) = \langle\rangle \,\} \\
\langle\rangle \in tock^* \\
= \{\, \text{Kleene closure} \,\} \\
\textbf{\textit{true}}
\end{array}
$$

2. ***Prefix***

$$
\begin{array}{l}
(a \to P)[\langle\rangle/tt'] \\
= \{\,\textit{prefix}\,\} \\
\textbf{\textit{T0}}\left(\begin{array}{l} a \notin refusals(tt')) \\ \quad \lhd\ trace(tt') \in tock^* \rhd \\ \qquad a = head(trace(idlesuffix(tt'))) \land \\ \qquad a \notin refusals(idleprefix(tt')) \land \\ \qquad P[tail(idlesuffix(tt'))/tt'] \end{array}\right)[\langle\rangle/tt'] \\
= \{\,\textit{definition conditional}\,\} \\
\textbf{\textit{T0}}(a \notin refusals(tt') \land trace(tt') \in tock^* \\
\quad \lor\ a = head(trace(idlesuffix(tt'))) \\
\qquad \land\ a \notin refusals(idleprefix(tt')) \\
\qquad \land\ P[tail(idlesuffix(tt'))/tt'])[\langle\rangle/tt'] \\
\Leftarrow \{\,\textit{propositional calculus}\,\} \\
\textbf{\textit{T0}}(a \notin refusals(tt') \land trace(tt') \in tock^*)[\langle\rangle/tt'] \\
= \{\,\textit{substitution}\,\} \\
\textbf{\textit{T0}}(a \notin refusals(\langle\rangle) \land \langle\rangle \in tock^*) \\
= \{\,\textbf{\textit{T0}}\,\} \\
a \notin refusals(\langle\rangle) \land trace(\langle\rangle) \in tock^* \land \langle\rangle \in TimedTrace \\
= \{\,\langle\rangle \in timedTrace\,\} \\
trace(\langle\rangle) \in tock^* \land a \notin refusals(\langle\rangle)) \\
\Leftarrow \{\,\textit{propositional calculus}\,\} \\
trace(\langle\rangle) \in tock^* \land a \notin refusals(\langle\rangle) \\
= \{\,trace(\langle\rangle) = \langle\rangle\,\} \\
\langle\rangle \in tock^* \land a \notin refusals(\langle\rangle) \\
= \{\,refusals(\langle\rangle) = \emptyset\,\} \\
\langle\rangle \in tock^* \land a \notin \emptyset \\
= \{\,\textit{empty set}\,\} \\
\langle\rangle \in tock^* \land \textbf{\textit{true}} \\
= \{\,\textit{propositional calculus}\,\} \\
\langle\rangle \in tock^* \\
= \{\,\textit{Kleene closure}\,\} \\
\textbf{\textit{true}}
\end{array}
$$

3. ***Internal choice*** *Suppose* $P[\langle\rangle/tt']$

$$
\begin{array}{l}
(P \sqcap Q)[\langle\rangle/tt'] \\
= \{\,\textit{nondeterministic choice}\,\} \\
(P \lor Q)[\langle\rangle/tt'] \\
= \{\,\textit{substitution}\,\} \\
P[\langle\rangle/tt'] \lor Q[\langle\rangle/tt'] \\
= \{\,\textit{assumption: } P[\langle\rangle/tt']\,\} \\
\textbf{\textit{true}}
\end{array}
$$

4. ***External choice*** *Suppose both* $P[\langle\rangle/tt']$ *and* $Q[\langle\rangle/tt']$

$$(P \,\Box\, Q)[\langle\rangle/tt']$$

$= \{$ *external choice* $\}$
$((P \land Q)[idleprefix(tt')/tt'] \land (P \lor Q))[\langle\rangle/tt']$
$= \{ idleprefix(\langle\rangle) = \langle\rangle \}$
$(P \land Q)[\langle\rangle/tt'] \land (P \lor Q)[\langle\rangle/tt']$
$= \{$ *propositional calculus* $\}$
$P[\langle\rangle/tt'] \land Q[\langle\rangle/tt'] \land (P[\langle\rangle/tt'] \lor Q[\langle\rangle/tt'])$
$= \{$ *propositional calculus* $\}$
$P[\langle\rangle/tt'] \land Q[\langle\rangle/tt']$
$= \{$ *assumptions:* $P[\langle\rangle/tt']$ *and* $Q[\langle\rangle/tt']$ $\}$
true

5. ***Parallel composition***

$(P \parallel_A Q)[\langle\rangle/tt']$
$= \{$ *parallel composition* $\}$
$(\exists t, u \bullet P[t/tt'] \land Q[u/tt'] \land tt' \in (t \parallel_A u))[\langle\rangle/tt']$
$= \{$ *substitution* $\}$
$\exists t, u \bullet P[t/tt'] \land Q[u/tt'] \land \langle\rangle \in (t \parallel_A u)$
$\Longleftarrow \{$ *predicate calculus* $\}$
$P[\langle\rangle/tt'] \land Q[\langle\rangle/tt'] \land \langle\rangle \in (\langle\rangle \parallel_A \langle\rangle)$
$= \{$ *parallel traces:* $\langle\rangle \in (\langle\rangle \parallel_A \langle\rangle)$ $\}$
$P[\langle\rangle/tt'] \land Q[\langle\rangle/tt']$
$= \{$ *assumptions:* $P[\langle\rangle/tt']$ *and* $Q[\langle\rangle/tt']$ $\}$
true

6. ***Hiding***

$(P \setminus A)[\langle\rangle/tt']$
$= \{$ *hiding* $\}$
$(\exists t \bullet P[t/tt'] \land A \text{ urgent } t \land (tt' = (t \setminus A)))[\langle\rangle/tt']$
$= \{$ *substitution* $\}$
$\exists t \bullet P[t/tt'] \land A \text{ urgent } t \land (\langle\rangle = (t \setminus A))$
$\Longleftarrow \{$ *predicate calculus* $\}$
$P[\langle\rangle/tt'] \land A \text{ urgent } \langle\rangle \land (\langle\rangle = (\langle\rangle \setminus A))$
$= \{$ *assumption:* $P[\langle\rangle/tt']$ $\}$
$A \text{ urgent } \langle\rangle \land (\langle\rangle = (\langle\rangle \setminus A))$
$= \{$ *urgency:* A urgent $\langle\rangle$ $\}$
$(\langle\rangle = (\langle\rangle \setminus A))$
$= \{$ *trace hiding:* $(\langle\rangle \setminus A) = \langle\rangle$ $\}$
$(\langle\rangle = \langle\rangle)$

$= \{\ \textit{equality}\ \}$

$\textbf{true}$

7. ***Timing*** $W.T.P.\ [\, P[\langle\rangle/tt'] \wedge Q[\langle\rangle/tt'] \Rightarrow (P \overset{n}{\rhd} Q)[\langle\rangle/tt']\,]$

$$
\begin{array}{l}
(P \overset{n}{\rhd} Q)[\langle\rangle/tt'] \\
= \{\ \textit{timeout semantics}\ \} \\
\left(\begin{array}{l}
(\exists\, u \bullet u \preceq \langle\rangle \wedge (trace(u) = tock^n) \wedge P[u/tt'] \wedge Q[\langle\rangle - u/tt']) \\
\quad \lhd\, tock^n \leq trace(\langle\rangle) \,\rhd \\
\qquad P[\langle\rangle/tt']
\end{array}\right) \\
= \{\ \textit{assumption: } P[\langle\rangle/tt']\ \} \\
\left(\begin{array}{l}
(\exists\, u \bullet u \preceq \langle\rangle \wedge (trace(u) = tock^n) \wedge P[u/tt'] \wedge Q[\langle\rangle - u/tt']) \\
\quad \lhd\, tock^n \leq trace(\langle\rangle) \,\rhd \\
\qquad \textbf{true}
\end{array}\right) \\
= \{\ \textit{conditional: } (P \lhd b \rhd \textbf{true}) = (b \Rightarrow P)\ \} \\
tock^n \leq trace(\langle\rangle) \Rightarrow \\
(\exists\, u \bullet u \preceq \langle\rangle \wedge (trace(u) = tock^n) \wedge P[u/tt'] \wedge Q[\langle\rangle - u/tt']) \\
= \{\ \textit{trace extraction: } trace(\langle\rangle) = \langle\rangle\ \} \\
tock^n \leq \langle\rangle \Rightarrow (\exists\, u \bullet u \preceq \langle\rangle \wedge (trace(u) = tock^n) \wedge P[u/tt'] \wedge Q[\langle\rangle - u/tt']) \\
= \{\ \textit{event repetition: } tock^n \leq \langle\rangle = (n = 0)\ \} \\
n = 0 \Rightarrow (\exists\, u \bullet u \preceq \langle\rangle \wedge (trace(u) = tock^n) \wedge P[u/tt'] \wedge Q[\langle\rangle - u/tt']) \\
= \{\ \textit{precedence: } u \preceq \langle\rangle = (u = \langle\rangle)\ \} \\
n = 0 \Rightarrow (\exists\, u \bullet (u = \langle\rangle) \wedge (trace(u) = tock^n) \wedge P[u/tt'] \wedge Q[\langle\rangle - u/tt']) \\
= \{\ \textit{predicate calculus: one-point rule}\ \} \\
n = 0 \Rightarrow (trace(\langle\rangle) = tock^n) \wedge P[\langle\rangle/tt'] \wedge Q(\langle\rangle - \langle\rangle) \\
= \{\ \textit{assumption: } P[\langle\rangle/tt']\ \} \\
n = 0 \Rightarrow (trace(\langle\rangle) = tock^n) \wedge Q(\langle\rangle - \langle\rangle) \\
= \{\ \textit{sequence difference: } s - \langle\rangle = s\ \} \\
n = 0 \Rightarrow (trace(\langle\rangle) = tock^n) \wedge Q[\langle\rangle/tt'] \\
= \{\ \textit{assumption: } Q[\langle\rangle/tt']\ \} \\
n = 0 \Rightarrow (trace(\langle\rangle) = tock^n) \\
= \{\ \textit{trace extraction: } trace(\langle\rangle) = \langle\rangle\ \} \\
n = 0 \Rightarrow (\langle\rangle = tock^n) \\
= \{\ \textit{Leibniz}\ \} \\
n = 0 \Rightarrow (\langle\rangle = tock^0) \\
= \{\ \textit{event repetition: } a^0 = \langle\rangle\ \} \\
n = 0 \Rightarrow (\langle\rangle = \langle\rangle) \\
= \{\ \textit{equality}\ \} \\
n = 0 \Rightarrow \textbf{true} \\
= \{\ \textit{propositional calculus}\ \}
\end{array}
$$

$\textbf{true}$

8. ***Recursion*** ***T1*** *can be written as the conjunctive idempotent* $\textbf{T1}(P) = P \wedge P[\langle\rangle/tt']$. *Recursion therefore satisfies* ***T1****, as demonstrated in [11].*

A.1.1 Prefix Closure

Theorem 4.2.2 (Prefix closure) *Every* ***CML*** *operator preserves* ***T2****-healthiness.*

Proof A.1.2 *By induction on program syntax.*

1. ***Deadlock***

   Assume $STOP \wedge t \preceq tt'$

   $t \preceq tt' \Rightarrow trace(t) \leq trace(tt')$

$$
\begin{array}{l}
STOP[t/tt'] \\
= \{\, STOP \,\} \\
(\textbf{T0}(trace(tt') \in tock^*))[t/tt'] \\
= \{\, \text{substitution} \,\} \\
trace(t) \in tock^* \wedge t \in timedTrace \\
= \{\, \text{assumption: } t \preceq tt'; (s \leq t) \wedge t \in timedTrace \Rightarrow s \in timedTrace \,\} \\
trace(t) \in tock^* \wedge tt' \in timedTrace \\
\Longleftarrow \ \{\, \text{assumption: } t \preceq tt'; \text{regular expression: } (s \leq t) \wedge t \in e^* \Rightarrow s \in e^* \,\} \\
trace(tt') \in tock^* \wedge tt' \in timedTrace \\
= \{\, STOP \,\} \\
STOP
\end{array}
$$

2. ***Prefix*** *W.T.P.*

$$(a \rightarrow P) \wedge t \preceq tt' \Rightarrow (a \rightarrow P)[t/tt']$$

   given

$$[\, P \wedge t \preceq tt' \Rightarrow P[t/tt'] \,]$$

$$
\begin{array}{l}
(a \rightarrow P)[t/tt'] \\
= \{\, \text{prefix} \,\} \\
\textbf{T0}\left(\begin{array}{l}
a \notin refusals(tt') \\
\qquad \lhd\, trace(tt') \in tock^* \rhd \\
\qquad\qquad a = head(trace(idlesuffix(tt'))) \wedge \\
\qquad\qquad a \notin refusals(idleprefix(tt')) \wedge \\
\qquad\qquad P[tail(idlesuffix(tt'))/tt']
\end{array}\right)[t/tt'] \\
= \{\, \textbf{T0} \,\}
\end{array}
$$

$$\left(\begin{array}{l} a \notin refusals(tt') \land tt' \in TimedTrace \\ \quad \lhd trace(tt') \in tock^* \rhd \\ \qquad a = head(trace(idlesuffix(tt'))) \land \\ \qquad a \notin refusals(idleprefix(tt')) \land \\ \qquad P[tail(idlesuffix(tt'))/tt'] \land \\ \qquad tt' \in TimedTrace \end{array}\right)[t/tt']$$

$= \{$ substitution $\}$

$$\left(\begin{array}{l} a \notin refusals(t) \land t \in TimedTrace \\ \quad \lhd trace(t) \in tock^* \rhd \\ \qquad a = head(trace(idlesuffix(t))) \land \\ \qquad a \notin refusals(idleprefix(t)) \land \\ \qquad P[tail(idlesuffix(t))/tt'] \land \\ \qquad t \in TimedTrace \end{array}\right)$$

$= \{$ conditional $\}$

$trace(t) \in tock^* \land a \notin refusals(t) \land t \in TimedTrace$

$\lor$

$\neg (trace(t) \in tock^*) \land a = head(trace(idlesuffix(t)))$
$\quad \land a \notin refusals(idleprefix(t)) \land P[tail(idlesuffix(t))/tt']$
$\quad \land t \in TimedTrace$

$= \{$ ***T1***(P) $\}$

$trace(t) \in tock^* \land a \notin refusals(t) \land t \in TimedTrace \land P[\langle\rangle/tt']$

$\lor$

$\neg (trace(t) \in tock^*) \land a = head(trace(idlesuffix(t)))$
$\quad \land a \notin refusals(idleprefix(t)) \land$
$\quad P[tail(idlesuffix(t))/tt'] \land t \in TimedTrace$

$= \{ a = head(trace(idlesuffix(t))) \Rightarrow \neg (trace(t) \in tock^*) \}$

$trace(t) \in tock^* \land a \notin refusals(t) \land t \in TimedTrace \land P[\langle\rangle/tt']$

$\lor$

$a = head(trace(idlesuffix(t))) \land a \notin refusals(idleprefix(t)) \land$
$\quad P[tail(idlesuffix(t))/tt'] \land t \in TimedTrace$

$= \{ a = head(trace(idlesuffix(t))) \Rightarrow \neg (trace(tt') \in tock^*) \}$

$trace(t) \in tock^* \land a \notin refusals(t) \land t \in TimedTrace \land P[\langle\rangle/tt']$

$\lor$

$\neg (trace(tt') \in tock^*) \land a = head(trace(idlesuffix(t)))$
$\quad \land a \notin refusals(idleprefix(t)) \land P[tail(idlesuffix(t))/tt'] \land t \in TimedTrace$

$= \{$ propositional calculus $\}$

$trace(t) \in tock^* \land a \notin refusals(t) \land t \in TimedTrace \land P[\langle\rangle/tt']$

$\lor$

$\neg (trace(tt') \in tock^*)$

$\lor$

$$\neg\,(trace(tt') \in tock^*) \land a = head(trace(idlesuffix(t))) \land a \notin refusals(idleprefix(t)) \land$$
$$\quad P[tail(idlesuffix(t))/tt'] \land t \in TimedTrace$$
$$= \{\ trace(t) \in tock^* \land a \notin refusals(t) \Rightarrow \}$$
$$\{\ a \notin refusals(idleprefix(t)) \land tail(idlesuffix(t)) = \langle\rangle\ \}$$
$$trace(t) \in tock^* \land a \notin refusals(t) \land t \in TimedTrace$$
$$\lor$$
$$\neg\,(trace(tt') \in tock^*) \land trace(t) \in tock^* \land a \notin refusals(idleprefix(t)) \land$$
$$\quad P[tail(idlesuffix(t))/tt'] \land t \in TimedTrace$$
$$\lor$$
$$\neg\,(trace(tt') \in tock^*) \land a = head(trace(idlesuffix(t)))$$
$$\quad \land\ a \notin refusals(idleprefix(t)) \land P[tail(idlesuffix(t))/tt']$$
$$\quad \land\ t \in TimedTrace$$
$$= \{\ \textit{rearranging}\ \}$$
$$trace(t) \in tock^* \land a \notin refusals(t) \land t \in TimedTrace$$
$$\lor$$
$$\neg\,(trace(tt') \in tock^*) \land (trace(t) \in tock^* \lor a = head(trace(idlesuffix(t)))) \land$$
$$\quad a \notin refusals(idleprefix(t)) \land P[tail(idlesuffix(t))/tt'] \land t \in TimedTrace$$
$$\Longleftarrow\ \{\ t \preceq tt' \land a = head(trace(idlesuffix(tt'))) \Rightarrow \}$$
$$\quad \{\ trace(t) \in tock^* \lor a = head(trace(idlesuffix(t)))\ \}$$
$$trace(t) \in tock^* \land a \notin refusals(t) \land t \in TimedTrace$$
$$\lor$$
$$\neg\,(trace(tt') \in tock^*) \land a = head(trace(idlesuffix(tt')))$$
$$\quad \land\ a \notin refusals(idleprefix(t)) \land P[tail(idlesuffix(t))/tt']$$
$$\quad \land\ t \in TimedTrace$$
$$\{\ \textbf{T2}(P);\ tail(idlesuffix(t)) \preceq tail(idlesuffix(tt'))\ \}$$
$$trace(t) \in tock^* \land a \notin refusals(t) \land t \in TimedTrace$$
$$\lor$$
$$\neg\,(trace(tt') \in tock^*) \land a = head(trace(idlesuffix(tt')))$$
$$\quad \land\ a \notin refusals(idleprefix(t)) \land P[tail(idlesuffix(tt'))/tt']$$
$$\quad \land\ t \in TimedTrace$$
$$\{\ t \preceq tt \land a \notin refusals(idleprefix(tt') \Rightarrow a \notin refusals(idleprefix(t)\ \}$$
$$trace(t) \in tock^* \land a \notin refusals(t) \land t \in TimedTrace$$
$$\lor$$
$$\neg\,(trace(tt') \in tock^*) \land a = head(trace(idlesuffix(tt')))$$
$$\quad \land\ a \notin refusals(idleprefix(tt')) \land P[tail(idlesuffix(tt'))/tt']$$
$$\quad \land\ t \in TimedTrace$$
$$\{\ t \preceq tt \land tt' \in TimedTrace \Rightarrow t \in timedTrace\ \}$$
$$trace(t) \in tock^* \land a \notin refusals(t) \land tt' \in TimedTrace$$

$$\vee$$

$$\neg\,(trace(tt') \in tock^*) \wedge a = head(trace(idlesuffix(tt')))$$
$$\wedge\; a \notin refusals(idleprefix(tt')) \wedge P[tail(idlesuffix(tt'))/tt']$$
$$\wedge\; tt' \in TimedTrace$$

$$= \{\, t \preceq tt \wedge a \notin refusals(tt') \Rightarrow a \notin refusals(t) \,\}$$

$$trace(t) \in tock^* \wedge a \notin refusals(tt') \wedge tt' \in TimedTrace$$

$$\vee$$

$$\neg\,(trace(tt') \in tock^*) \wedge a = head(trace(idlesuffix(tt')))$$
$$refusals(idleprefix(tt')) \wedge P[tail(idlesuffix(tt'))/tt']$$
$$\wedge\; tt' \in TimedTrace$$

$$= \{\, t \preceq tt \wedge trace(tt') \in tock^* \Rightarrow trace(t) \in tock^* \,\}$$

$$trace(tt') \in tock^* \wedge a \notin refusals(tt') \wedge tt' \in TimedTrace$$

$$\vee$$

$$\neg\,(trace(tt') \in tock^*) \wedge a = head(trace(idlesuffix(tt')))$$
$$\wedge\; a \notin refusals(idleprefix(tt')) \wedge P[tail(idlesuffix(tt'))/tt']$$
$$\wedge\; tt' \in TimedTrace$$

$$= \{\, \textit{conditional} \,\}$$

$$\begin{array}{l} a \notin refusals(tt') \wedge tt' \in TimedTrace \\ \quad \lhd\, trace(tt') \in tock^* \,\rhd \\ \qquad a = head(trace(idlesuffix(tt'))) \wedge \\ \qquad a \notin refusals(idleprefix(tt')) \wedge \\ \qquad P[tail(idlesuffix(tt'))/tt'] \wedge \\ \qquad tt' \in TimedTrace \end{array}$$

$$= \{\, \textbf{\textit{T0}} \,\}$$

$$\textbf{\textit{T0}} \left(\begin{array}{l} a \notin refusals(tt') \\ \quad \lhd\, trace(tt') \in tock^* \,\rhd \\ \qquad a = head(trace(idlesuffix(tt'))) \wedge \\ \qquad a \notin refusals(idleprefix(tt')) \wedge \\ \qquad P[tail(idlesuffix(tt'))/tt'] \end{array} \right)$$

$$= \{\, \textit{prefix} \,\}$$

$$a \rightarrow P$$

$$= \{\, \textit{assumption: } a \rightarrow P \,\}$$

true

3. ***Nondeterminism*** *Assume*

$$t \preceq tt' \wedge P \Rightarrow P[tr \frown t/tt']$$
$$t \preceq tt' \wedge Q \Rightarrow Q[tr \frown t/tt']$$

W.T.P.

$$t \preceq tt' \wedge P \sqcap Q \Rightarrow (P \sqcap Q)[tr \frown t/tt']$$

Follows from properties of substitution.

4. ***External choice***

$$P \Box Q = (P \wedge Q)[idleprefix(tt')/tt'] \wedge (P \vee Q)$$

Assume

$$t \preceq tt' \wedge P \Rightarrow P[t/tt']$$
$$t \preceq tt' \wedge Q \Rightarrow Q[t/tt']$$

W.T.P.

$$t \preceq tt' \wedge (P \Box Q) \Rightarrow (P \Box Q)[t/tt']$$

$(P \Box Q)[t/tt']$
$= \{$ *external choice* $\}$
$((P \wedge Q)[idleprefix(tt')/tt'] \wedge (P \vee Q))[t/tt']$
$= \{$ *substitution* $\}$
$(P[idleprefix(tt')/tt'] \wedge Q[idleprefix(tt')/tt'] \wedge (P \vee Q))[t/tt']$
$= \{$ *substitution* $\}$
$(P[idleprefix(t)/tt'] \wedge Q[idleprefix(t)/tt'] \wedge (P[t/tt'] \vee Q[t/tt']))$
$\Longleftarrow \{$ *assumptions: (i)* $t \preceq tt'$*; (ii)* $t \preceq tt' \wedge P \Rightarrow P[t/tt'];\}$
$\{ \ t \preceq tt' \wedge Q \Rightarrow Q[t/tt'] \}$
$(P[idleprefix(t)/tt'] \wedge Q[idleprefix(t)/tt'] \wedge (P \vee Q))$
$\{ t \preceq tt' \Rightarrow idleprefix(t) \preceq idleprefix(tt') \}$
$\Longleftarrow \{ idleprefix(t) \preceq idleprefix(tt') \wedge P[idleprefix(tt')/idleprefix(tt')] \Rightarrow \}$
$\{ P[idleprefix(t)/idleprefix(tt')] \}$
$\{ idleprefix(t) \preceq idleprefix(tt') \wedge Q[idleprefix(tt')/idleprefix(tt') \Rightarrow \}$
$\{ Q[idleprefix(t)/idleprefix(tt')] \}$
$(P[idleprefix(tt')/tt'] \wedge Q[idleprefix(tt')/tt'] \wedge (P \vee Q))$
$= \{$ *propositional calculus* $\}$
$((P \wedge Q)[idleprefix(tt')/tt'] \wedge (P \vee Q))$
$= \{$ *definition external choice* $\}$
$P \Box Q$

5. ***parallel*** *Assume* $P \parallel_A Q$ *and* $t \preceq tt'$. *W.T.P.* $(P \parallel_A Q)[t/tt']$

$$
\begin{array}{l}
(P \parallel_A Q)[t/tt'] \\
= \{\, \text{definition of } \parallel_A \,\} \\
(\exists\, s, u \bullet P[s/tt'] \wedge Q[u/tt'] \wedge tt' \in s \parallel_A u)[t/tt'] \\
= \{\, \text{substitution} \,\} \\
\exists\, s, u \bullet P[s/tt'] \wedge Q[u/tt'] \wedge t \in s \parallel_A u \\
\Longleftarrow \{\, \text{P and Q are T2-healthy} \,\} \\
\exists\, s, u \bullet P[s_1/tt'] \wedge s \preceq s_1 \wedge Q[u_1/tt'] \wedge u \preceq u_1 \wedge t \in s \parallel_A u \\
\Longleftarrow \{\, \text{lemma: parallel-precedence} \,\} \\
P[s_1/tt'] \wedge Q[u_1/tt'] \wedge tt' \in s_1 \parallel_A u_1 \wedge t \preceq tt' \\
= \{\, \text{assumption: } t \preceq tt' \,\} \\
P[s_1/tt'] \wedge Q[u_1/tt'] \wedge tt' \in s_1 \parallel_A u_1 \\
= \{\, \text{definition of } \parallel_A \,\} \\
= P \parallel_A Q
\end{array}
$$

For proof of the parallel precedence lemma, see appendix.

6. ***Hiding*** *Assume*

$$P \wedge t \preceq tt' \Rightarrow P[t/tt']$$

Would like to prove

$$P \setminus A \wedge t \preceq tt' \Rightarrow (P \setminus A)[t/tt']$$

$$
\begin{array}{l}
P \setminus A \\
= \{\, \text{definition of hiding} \,\} \\
\exists\, u \bullet P[u/tt'] \wedge A \text{ urgent } u \wedge tt' = u \setminus A \\
= \{\, t \preceq tt' \wedge tt' = u \setminus A \Rightarrow \exists\, v \bullet v \leq u \wedge t = v \setminus A \,\} \\
\exists\, u, v \bullet P[u/tt'] \wedge A \text{ urgent } u \wedge tt' = u \setminus A \wedge v \leq u \wedge t = v \setminus A \\
= \{\, P \text{ is } T2 \,\} \\
\exists\, u, v \bullet P[v/tt'] \wedge A \text{ urgent } u \wedge tt' = u \setminus A \wedge v \leq u \wedge t = v \setminus A \\
= \{\, A \text{ urgent } u \wedge v \leq u \Rightarrow A \text{ urgent } v \,\} \\
\exists\, u, v \bullet P[v/tt'] \wedge A \text{ urgent } v \wedge tt' = u \setminus A \wedge v \leq u \wedge t = v \setminus A \\
\Rightarrow \{\, \text{predicate calculus} \,\} \\
\exists\, v \bullet P[v/tt'] \wedge A \text{ urgent } v \wedge t = v \setminus A \\
= \{\, \text{substitution} \,\} \\
(\exists\, v \bullet P[v/tt'] \wedge A \text{ urgent } v \wedge tt' = v \setminus A)[t/tt'] \\
= \{\, \text{definition hiding} \,\} \\
(P \setminus A)[t/tt']
\end{array}
$$

7. ***Timing***

$$[\, t \preceq tt' \wedge (P \Rightarrow P[t/tt']) \wedge (Q \Rightarrow Q[t/tt']) \wedge (P \overset{n}{\triangleright} Q) \Rightarrow (P \overset{n}{\triangleright} Q)[t/tt'] \,]$$

Case 1: $tock^n \leq trace(tt')$

$$P \overset{n}{\triangleright} Q$$

$$
\begin{array}{l}
= \{\, \textit{timeout} \,\} \\
(\exists\, u \bullet u \preceq tt' \land trace(u) = tock^n \land P[u/tt'] \land Q[tt' - u/tt']) \\
\quad \lhd\, tock^n \leq trace(tt') \,\rhd \\
\quad\quad P \\
= \{\, \textit{assumption: } tock^n \leq trace(tt') \,\} \\
\exists\, u \bullet u \preceq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt'] \\
\Rightarrow \{\, \textit{predicate calculus: for arbitrary } u \,\} \\
u \preceq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt'] \\
= \{\, \textit{precedence: } u \preceq tt' \land t \preceq tt' \Rightarrow t \prec u \lor u \preceq t \,\} \\
u \preceq tt' \land (t \prec u \lor u \preceq t) \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt'] \\
= \{\, \textit{propositional calculus} \,\} \\
(u \preceq tt' \land t \prec u \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt']) \\
\lor\, (u \preceq tt' \land u \prec t \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt']) \\
\Rightarrow \{\, \textit{precedence: } t \prec u \land (trace(u) = tock^n) \Rightarrow \neg\, tock^n \leq trace(t) \,\} \\
(\neg\, tock^n \leq trace(t) \land u \preceq tt' \land t \prec u \land (trace(u) = tock^n) \\
\quad \land P[u/tt'] \land Q[tt' - u/tt']) \\
\lor\, (u \preceq tt' \land u \preceq t \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt']) \\
\Rightarrow \{\, \textit{assumption: } t \preceq u \land P[u/tt'] \Rightarrow P[t/tt'] \,\} \\
(\neg\, tock^n \leq trace(t) \land u \preceq tt' \land t \prec u \land (trace(u) = tock^n) \\
\quad \land P[t/tt'] \land Q[tt' - u/tt']) \\
\lor\, (u \preceq tt' \land u \preceq t \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt']) \\
\Rightarrow \{\, \textit{propositional calculus} \,\} \\
(\neg\, tock^n \leq trace(t) \land P[t/tt']) \\
\lor\, (u \preceq tt' \land u \preceq t \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt']) \\
\Rightarrow \{\, \textit{precedence: } u \preceq t \preceq tt' \Rightarrow (t - u) \preceq (tt' - u) \,\} \\
(\neg\, tock^n \leq trace(t) \land P[t/tt']) \\
\lor\, (u \preceq tt' \land u \preceq t \land (t - u) \preceq (tt' - u) \land (trace(u) = tock^n) \\
\land P[u/tt'] \land Q[tt' - u/tt']) \\
\Rightarrow \{\, \textit{assumption: } (t - u) \preceq (tt' - u) \land Q[tt' - u/tt'] \Rightarrow Q[t - u/tt'] \,\} \\
(\neg\, tock^n \leq trace(t) \land P[t/tt']) \\
\lor\, (u \preceq tt' \land u \preceq t \land (trace(u) = tock^n) \land P[u/tt'] \land Q[t - u/tt']) \\
\Rightarrow \{\, \textit{propositional calculus} \,\} \\
(\neg\, tock^n \leq trace(t) \land P[t/tt']) \\
\lor\, (u \preceq t \land (trace(u) = tock^n) \land P[u/tt'] \land Q[t - u/tt']) \\
\Rightarrow \{\, \textit{precedence: } u \preceq t \Rightarrow trace(u) \leq trace(t) \,\} \\
(\neg\, tock^n \leq trace(t) \land P[t/tt']) \\
\lor\, (trace(u) \leq trace(t) \land u \preceq t \land (trace(u) = tock^n) \\
\quad \land P[u/tt'] \land Q[t - u/tt']) \\
= \{\, \textit{Leibniz} \,\}
\end{array}
$$

$$
\begin{aligned}
&(\neg\ tock^n \le trace(t) \land P[t/tt'])\\
&\lor (tock^n \le trace(t) \land u \preceq t \land (trace(u) = tock^n) \land P[u/tt'] \land Q[t-u/tt'])\\
&\Rightarrow \{\ \textit{predicate calculus}\ \}\\
&(\neg\ tock^n \le trace(t) \land P[t/tt'])\\
&\lor (tock^n \le trace(t) \land (\exists\, u \bullet u \preceq t \land (trace(u) = tock^n)\\
&\qquad \land P[u/tt'] \land Q[t-u/tt']))\\
&= \{\ \textit{conditional}\ \}\\
&(\exists\, u \bullet u \preceq t \land (trace(u) = tock^n) \land P[u/tt'] \land Q[t-u/tt'])\\
&\qquad \lhd\, tock^n \le trace(t) \rhd\\
&\qquad\qquad P[t/tt']\\
&= \{\ \textit{timeout}\ \}\\
&(P \overset{n}{\rhd} Q)[t/tt']
\end{aligned}
$$

Case 2: $\neg\ tock^n \le trace(tt')$*. Note that sequence prefix is transitive; in particular,*

$$
\begin{aligned}
&tock^n \le trace(t) \land trace(t) \le trace(tt') \Rightarrow tock^n \le trace(tt')\\
&= \{\ \textit{propositional calculus}\ \}\\
&\neg\ tock^n \le trace(tt') \land trace(t) \le trace(tt') \Rightarrow \neg\ tock^n \le trace(t)
\end{aligned}
$$

$$
\begin{aligned}
&P \overset{n}{\rhd} Q\\
&= \{\ \textit{timeout}\ \}\\
&(\exists\, u \bullet u \preceq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt'-u/tt'])\\
&\qquad \lhd\, tock^n \le trace(tt') \rhd\\
&\qquad\qquad P\\
&= \{\ \textit{assumption: } \neg\ tock^n \le trace(tt')\ \}\\
&P\\
&\Rightarrow \{\ \textit{assumption: } P \Rightarrow P[t/tt']\ \}\\
&P[t/tt']\\
&= \{\ \textit{assumption: } \neg\ tock^n \le trace(tt')\ \}\\
&\neg\ tock^n \le trace(tt') \land P[t/tt']\\
&= \{\ \textit{assumption: } trace(t) \le trace(tt')\ \}\\
&\neg\ tock^n \le trace(tt') \land trace(t) \le trace(tt') \land P[t/tt']\\
&\Rightarrow \{\ \textit{transitivity}\ \}\\
&\neg\ tock^n \le trace(t) \land P[t/tt']\\
&\Rightarrow \{\ \textit{conditional}\ \}\\
&(\exists\, u \bullet u \preceq t \land (trace(u) = tock^n) \land P[u/tt'] \land Q[t-u/tt'])\\
&\qquad \lhd\, tock^n \le trace(t) \rhd\\
&\qquad\qquad P[t/tt']\\
&= \{\ \textit{timeout}\ \}\\
&(P \overset{n}{\rhd} Q)[t/tt']
\end{aligned}
$$

8. ***Recursion T2*** *can be written as the conjunctive idempotent*

$$\mathbf{T2}(P) = P \wedge (P[t/tt'] \vee \neg\, t \preceq tt')$$

Recursion therefore satisfies ***T2****, by [11].*

A.1.2 Zeno Freedom

Theorem 4.2.5 (Prefix closure) *Suppose that P is a time-guarded process, then for every k there is an n, such that P is* ***T5****-healthy.*

Proof A.1.3 *Every program operator satisfies $T5$*

1. ***STOP***

$$\begin{aligned}
& T5(STOP) \\
& = \{\text{ definition of } T5 \,\} \\
& STOP \wedge \exists\, n \bullet \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(tt')) \leq n \\
& \Longleftarrow \{\text{ contract scope, remove superfluous conjunct} \,\} \\
& \exists\, n \bullet STOP \wedge \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(tt')) \leq n \wedge k = n \\
& = \{\text{ one point rule} \,\} \\
& STOP \wedge \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(tt')) \leq k \\
& \Longleftarrow \{\text{ arithmetic} \,\} \\
& trace(tt') \in tock^* \wedge \#(tt' \upharpoonright tock) = \#(trace(tt')) \\
& = \{\, trace(t) \in tock^* \Rightarrow \#(trace(t)) = \#(t \upharpoonright tock) \,\} \\
& trace(tt') \in tock^* \\
& = \{\text{ definition } STOP \,\} \\
& STOP
\end{aligned}$$

2. ***Prefix*** *Assume (P), $a \to P$. W.T.P.:* ***T5***$(a \to P)$

$$\begin{aligned}
& \mathbf{T5}(a \to P) \\
& = \{\text{ definition } a \to P;\, \mathbf{T5}(P) \,\} \\
& \left(\begin{array}{l}
a \notin refusals(tt')) \\
\quad \lhd\, trace(tt') \in tock^* \,\rhd \\
\qquad a = head(trace(idlesuffix(tt'))) \wedge \\
\qquad a \notin refusals(idleprefix(tt')) \wedge \\
\qquad P[tail(idlesuffix(tt'))/tt']
\end{array}\right) \\
& \quad \wedge \exists\, n \bullet \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(tt')) \leq n \\
& = \{\, \#(trace(idleprefix(t))) + \#(trace((idlesuffix(t)) = \#(trace(t)) \,\} \\
& \left(\begin{array}{l}
a \notin refusals(tt') \\
\quad \lhd\, trace(tt') \in tock^* \,\rhd \\
\qquad a = head(trace(idlesuffix(tt'))) \wedge \\
\qquad a \notin refusals(idleprefix(tt')) \wedge \\
\qquad P[tail(idlesuffix(tt'))/tt']
\end{array}\right)
\end{aligned}$$

$$\land \exists n \bullet \#(tt' \upharpoonright tock) \leq k \Rightarrow$$
$$\quad \#(trace(idleprefix(tt')) \frown \#(trace(idlesuffix(tt')))) \leq n$$
$$= \{ \#(t \upharpoonright tock) = \#(idleprefix(t \upharpoonright tock)) + \#(idlesuffix(t \upharpoonright tock)) \}$$
$$\left(\begin{array}{l} a \notin refusals(tt') \\ \quad \lhd trace(tt') \in tock^* \rhd \\ \qquad a = head(trace(idlesuffix(tt'))) \land \\ \qquad a \notin refusals(idleprefix(tt')) \land \\ \qquad P[tail(idlesuffix(tt'))/tt'] \end{array} \right)$$
$$\land \exists n \bullet \#(idleprefix(tt' \upharpoonright tock)) + \#(idlesuffix(tt' \upharpoonright tock)) \leq k \Rightarrow$$
$$\quad \#(trace(idleprefix(tt')) + \#(trace(idlesuffix(tt')))) \leq n$$
$$= \{ \#(idleprefix(t \upharpoonright tock)) = \#(trace(idleprefix(t)) \}$$
$$\left(\begin{array}{l} a \notin refusals(tt') \\ \quad \lhd trace(tt') \in tock^* \rhd \\ \qquad a = head(trace(idlesuffix(tt'))) \land \\ \qquad a \notin refusals(idleprefix(tt')) \land \\ \qquad P[tail(idlesuffix(tt'))/tt'] \end{array} \right)$$
$$\land \exists n \bullet \#(trace(idleprefix(tt')) + \#(idlesuffix(tt' \upharpoonright tock)) \leq k \Rightarrow$$
$$\quad \#(trace(idleprefix(tt')) + \#(trace(idlesuffix(tt')))) \leq n$$
$$= \{ \textit{renaming} \}$$
$$\left(\begin{array}{l} a \notin refusals(tt') \\ \quad \lhd trace(tt') \in tock^* \rhd \\ \qquad a = head(trace(idlesuffix(tt'))) \land \\ \qquad a \notin refusals(idleprefix(tt')) \land \\ \qquad P[tail(idlesuffix(tt'))/tt'] \end{array} \right)$$
$$\land \exists n, n_i \bullet n_i + \#(idlesuffix(tt' \upharpoonright tock)) \leq k \Rightarrow$$
$$\quad n_i + \#(trace(idlesuffix(tt')))) \leq n$$
$$= \{ \textit{arithmetic} \}$$
$$\left(\begin{array}{l} a \notin refusals(tt') \\ \quad \lhd trace(tt') \in tock^* \rhd \\ \qquad a = head(trace(idlesuffix(tt'))) \land \\ \qquad a \notin refusals(idleprefix(tt')) \land \\ \qquad P[tail(idlesuffix(tt'))/tt'] \end{array} \right)$$
$$\land \exists n, n_i \bullet \#(idlesuffix(tt' \upharpoonright tock)) \leq k - n_i \Rightarrow$$
$$\quad \#(trace(idlesuffix(tt')))) \leq n - n_i$$
$$= \{ \#(t) = \#(tl(t)) + 1 \}$$
$$\left(\begin{array}{l} a \notin refusals(tt') \\ \quad \lhd trace(tt') \in tock^* \rhd \\ \qquad a = head(trace(idlesuffix(tt'))) \land \\ \qquad a \notin refusals(idleprefix(tt')) \land \\ \qquad P[tail(idlesuffix(tt'))/tt'] \end{array} \right)$$
$$\land \quad \exists n, n_i \bullet \#(tl(idlesuffix(tt' \upharpoonright tock))) + 1 \leq k - n_i \Rightarrow$$
$$\quad \#(tl(trace(idlesuffix(tt'))))) + 1 \leq n - n_i$$

$$= \{ \textit{arithmetic} \}$$

$$\left(\begin{array}{l} a \notin refusals(tt') \\ \quad \lhd trace(tt') \in tock^* \rhd \\ \quad\quad a = head(trace(idlesuffix(tt'))) \land \\ \quad\quad a \notin refusals(idleprefix(tt')) \land \\ \quad\quad P[tail(idlesuffix(tt'))/tt'] \end{array} \right)$$

$$\land \exists n, n_i \bullet \#(tl(idlesuffix(tt' \restriction tock))) \leq k - n_i - 1 \Rightarrow$$

$$\#(tl(trace(idlesuffix(tt')))) \leq n - n_i - 1$$

$$= \{ \textit{renaming} \}$$

$$\left(\begin{array}{l} a \notin refusals(tt') \\ \quad \lhd trace(tt') \in tock^* \rhd \\ \quad\quad a = head(trace(idlesuffix(tt'))) \land \\ \quad\quad a \notin refusals(idleprefix(tt')) \land \\ \quad\quad P[tail(idlesuffix(tt'))/tt'] \end{array} \right)$$

$$\land \exists n_t \bullet \#(tl(idlesuffix(tt' \restriction tock))) \leq l \Rightarrow$$

$$\#(tl(trace(idlesuffix(tt')))) \leq n_t$$

$$= \{ \textit{\textbf{T5}(P)} \}$$

$$\begin{array}{l} a \notin refusals(tt')) \\ \quad \lhd trace(tt') \in tock^* \rhd \\ \quad\quad a = head(trace(idlesuffix(tt'))) \land \\ \quad\quad a \notin refusals(idleprefix(tt')) \land \\ \quad\quad P[tail(idlesuffix(tt'))/tt'] \end{array}$$

$$= \{ \textit{definition } a \rightarrow P \}$$

$$a \rightarrow P$$

3. ***internal choice.*** *Assume* **T5**(P), **T5**$(Q), P \sqcap Q$. *W.T.P.* **T5**$(P \sqcap Q)$

$$\textbf{T5}(P \sqcap Q)$$

$$= \{ \textit{definition } \sqcap \}$$

$$= \textbf{T5}(P \lor Q)$$

$$= \{ \textit{definition } \textbf{T5} \}$$

$$(P \lor Q) \land \exists n \bullet \#(tt' \restriction tock) \leq k \Rightarrow \#(trace(tt')) \leq n$$

$$= \{ \textit{de Morgans} \}$$

$$(P \land \exists n \bullet \#(tt' \restriction tock) \leq k \Rightarrow \#(trace(tt')) \leq n) \lor$$

$$(Q \land \exists n \bullet \#(tt' \restriction tock) \leq k \Rightarrow \#(trace(tt')) \leq n)$$

$$= \{ \textbf{T5}(P), \textbf{T5}(Q) \}$$

$$(P \lor Q)$$

$$= \{ \textit{definition } \sqcap \}$$

$$P \sqcap Q$$

4. ***external choice*** *Assume* **T5**(P), **T5**$(Q), P \Box Q$. *W.T.P.* **T5**$(P \Box Q)$

$$\textbf{T5}(P \Box Q)$$

$= \{$ definition $\Box$ $\}$
$\textbf{T5}((P \land Q)[idleprefix(tt')/tt'] \land (P \lor Q))$
$= \{$ definition $\textbf{T5}$ $\}$
$(P \land Q)[idleprefix(tt')/tt'] \land (P \lor Q) \land$
$\quad \exists\, n \bullet \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(tt')) \leq n$
$= \{$ de Morgans $\}$
$(P \land Q)[idleprefix(tt')/tt'] \land$
$\quad ((P \land \exists\, n \bullet \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(tt')) \leq n) \lor$
$\quad (Q \land \exists\, n \bullet \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(tt')) \leq n))$
$= \{\ \textbf{T5}(P), \textbf{T5}(Q)\ \}$
$(P \land Q)[idleprefix(tt')/tt'] \land (P \lor Q)$
$= \{$ definition $\Box$ $\}$
$P \Box Q$

5. ***parallel*** *Assume* $\textbf{T5}(P), \textbf{T5}(Q), P \parallel_A Q$. *W.T.P.* $\textbf{T5}(P \parallel_A Q)$

$\textbf{T5}(P \parallel_A Q)$
$= \{$ definition parallel $\}$
$\textbf{T5}(\exists\, t, u \bullet P[t/tt'] \land Q[u/tt'] \land tt' \in t \parallel_A u)$
$= \{$ definition $\textbf{T5}$ $\}$
$(\exists\, t, u \bullet P[t/tt'] \land Q[u/tt'] \land tt' \in t \parallel_A u) \land$
$\quad \exists\, n \bullet \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(tt')) \leq n$
$= \{\ n \in \mathbb{N} \Rightarrow (\exists\, n_t, n_u \bullet n = n_t + n_u)\ \}$
$\exists\, t, u, n_t, n_u \bullet P[t/tt'] \land Q[u/tt'] \land tt' \in t \parallel_A u$
$\quad \land \#(tt' \upharpoonright tock) \leq k \Rightarrow (\#(trace(tt')) \leq n_u + n_t)$
$= \{\ q \in w \parallel_A x \Rightarrow \#(trace(q)) \leq \#(trace(w)) + \#(trace(u))\ \}$
$\exists\, t, u, n_t, n_u \bullet P[t/tt'] \land Q[u/tt'] \land tt' \in t \parallel_A u$
$\quad \land \#(tt' \upharpoonright tock) \leq k \Rightarrow (\#(trace(t)) + \#(trace(u)) \leq n_u + n_t)$
$= \{$ arithmetic $\}$
$\exists\, t, u, n_t, n_u \bullet P[t/tt'] \land Q[u/tt'] \land tt' \in t \parallel_A u$
$\quad \land \#(tt' \upharpoonright tock) \leq k \Rightarrow (\#(trace(t)) \leq n_t \land \#(trace(u)) \leq n_u)$
$= \{$ propositional calculus $\}$
$\exists\, t, u, n_t, n_u \bullet P[t/tt'] \land Q[u/tt'] \land tt' \in t \parallel_A u$
$\quad \land \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(t)) \leq n_t$
$\quad \land \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(u)) \leq n_u$
$= \{\ q \in w \parallel_A x \Rightarrow \#(q \upharpoonright tock) = \#(w \upharpoonright tock) \land \#(q \upharpoonright tock) = \#(x \upharpoonright tock)\ \}$
$\exists\, t, u, n_t, n_u \bullet P[t/tt'] \land Q[u/tt'] \land tt' \in t \parallel_A u$
$\quad \land \#(t \upharpoonright tock) \leq k \Rightarrow \#(trace(t)) \leq n_t$
$\quad \land \#(u \upharpoonright tock) \leq k \Rightarrow \#(trace(u)) \leq n_u$
$= \{$ contract scope $\}$

$$(\exists\, t, u \bullet P[t/tt'] \wedge Q[u/tt'] \wedge tt' \in t \parallel_A u)$$
$$\wedge\, (\exists\, n_t \bullet \#(t \upharpoonright tock) \leq k \Rightarrow \#(trace(t)) \leq n_t) \wedge$$
$$(\exists\, n_u \bullet \#(u \upharpoonright tock) \leq k \Rightarrow \#(trace(u)) \leq n_u)$$
$= \{$ ***T5***(P),***T5***(Q) $\}$
$$(\exists\, t, u \bullet P[t/tt'] \wedge Q[u/tt'] \wedge tt' \in t \parallel_A u)$$
$= \{$ *definition parallel* $\}$
$$(P \parallel_A Q)$$

6. ***hiding*** *Assume* $P \setminus A$, ***T5***(P). *W.T.P.* ***T5***$(P \setminus A)$

T5$(P \setminus A)$
$= \{$ *definition* ***T5***$(P), P \setminus A$ $\}$
$$(\exists\, t \bullet P[t/tt'] \wedge A \text{ urgent } t \wedge tt' = t \setminus A) \wedge \exists\, n \bullet \#(tt' \upharpoonright tock) \leq k \Rightarrow$$
$$\#(trace(tt')) \leq n$$
$= \{$ *substitution* $\}$
$$(\exists\, t \bullet P[t/tt'] \wedge A \text{ urgent } t \wedge tt' = t \setminus A) \wedge \exists\, n \bullet \#((t \setminus A) \upharpoonright tock) \leq k$$
$$\Rightarrow \#(trace(t \setminus A)) \leq n$$
$\Longleftarrow \{ \#(trace(s \setminus B)) \leq \#(trace(s)) \}$
$$(\exists\, t \bullet P[t/tt'] \wedge A \text{ urgent } t \wedge tt' = t \setminus A) \wedge \exists\, n \bullet \#((t \setminus A) \upharpoonright tock) \leq k$$
$$\Rightarrow \#(trace(t)) \leq n$$
$= \{ \#(s \upharpoonright tock) = \#((s \setminus B) \upharpoonright tock) \}$
$$(\exists\, t \bullet P[t/tt'] \wedge A \text{ urgent } t \wedge tt' = t \setminus A) \wedge \exists\, n \bullet \#(t \upharpoonright tock) \leq k \Rightarrow$$
$$\#(trace(t)) \leq n$$
$= \{$ ***T5***$(P[t/tt'])$ $\}$
$$(\exists\, t \bullet P[t/tt'] \wedge A \text{ urgent } t \wedge tt' = t \setminus A)$$
$= \{$ *definition* $P \setminus A$ $\}$
$$P \setminus A$$

2186 7. ***Timeout*** *Assume* ***T5***(P), ***T5***$(Q), P \overset{n}{\rhd} Q$. *Want to prove* ***T5***$(P \overset{n}{\rhd} Q)$.

2187 *case 1:* $tock^n \leq trace(tt')$

T5$(P \overset{n}{\rhd} Q)$
$= \{$ *definition timeout* $\}$
$$\textbf{T5}\left(\begin{array}{l}(\exists\, u \bullet u \leq tt' \wedge (trace(u) = tock^n) \wedge P[u/tt'] \wedge Q[tt' - u/tt'])\\ \qquad \lhd\, tock^n \leq trace(tt') \,\rhd \\ \qquad\quad P\end{array}\right)$$
$= \{$ *definition* ***T5*** $\}$
$$\left(\begin{array}{l}(\exists\, u \bullet u \leq tt' \wedge (trace(u) = tock^n) \wedge P[u/tt'] \wedge Q[tt' - u/tt'])\\ \qquad \lhd\, tock^n \leq trace(tt') \,\rhd \\ \qquad\quad P\end{array}\right)$$
$$\wedge\, \exists\, n \bullet \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(tt')) \leq n$$

$$
\begin{aligned}
&= \{\, \textit{case assumption: } tock^n \leq trace(tt') \,\}\\
&\left(\begin{array}{l}(\exists\, u \bullet u \leq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt'])\\ \quad \lhd\, tock^n \leq trace(tt') \,\rhd \\ \quad\quad P\end{array}\right)\\
&\quad \land \exists\, n \bullet \#(tt' \restriction tock) \leq k \Rightarrow \#(trace(tt')) \leq n \land tock^n \leq trace(tt')\\
&= \{\, r \lhd b \rhd s \land b = r \land b \,\}\\
&(\exists\, u \bullet u \leq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt'])\\
&\quad \land tock^n \leq trace(tt') \land \exists\, n \bullet \#(tt' \restriction tock) \leq k \Rightarrow \#(trace(tt')) \leq n\\
&= \{\, \textit{widen scope} \,\}\\
&(\exists\, u, n \bullet u \leq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt'])\\
&\quad \land tock^n \leq trace(tt') \land \#(tt' \restriction tock) \leq k \Rightarrow \#(trace(tt')) \leq n\\
&\Rightarrow \{\, k = k_p + k_q;\ n = n_p + n_q \,\}\\
&\exists\, u, n_p, n_q \bullet u \leq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt']\\
&\quad \land tock^n \leq trace(tt') \land \#(tt' \restriction tock) \leq k_p + k_q \Rightarrow \#(trace(tt') \leq n_p + n_q\\
&= \{\, a \restriction tock \frown b \restriction tock = (a \frown b) \restriction tock;\ \#(a \frown b) = \#a \frown \#b \,\}\\
&\exists\, u, n_p, n_q \bullet u \leq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt']\\
&\land tock^n \leq trace(tt') \land (\#(u \restriction tock) \leq k_p \land \#(tt' - u \restriction tock) \leq k_q)\\
&\qquad \Rightarrow \#(trace(u) \leq n_p \land \#(trace(tt' - u)) \leq n_q)\\
&= \{\, (a \Rightarrow b \land c \Rightarrow d) \Rightarrow (a \land c \Rightarrow b \land d) \,\}\\
&\exists\, u, n_p, n_q \bullet u \leq tt' \land trace(u) = tock^n \land P[u/tt'] \land Q[tt' - u/tt']\\
&\land tock^n \leq trace(tt') \land \#(u \restriction tock) \leq k_p \Rightarrow \#(trace(u)) \leq n_p\\
&\quad \land \#(tt' - u \restriction tock) \leq k_q \Rightarrow \#(trace(tt' - u)) \leq n_q\\
&= \{\, \textit{reduce scope} \,\}\\
&(\exists\, u \bullet u \leq tt' \land trace(u) = tock^n \land P[u/tt'] \land Q[tt' - u/tt'])\\
&\land tock^n \leq trace(tt') \land \exists\, n_p \bullet \#(u \restriction tock) \leq k_p \Rightarrow \#(trace(u)) \leq n_p\\
&\quad \land \exists\, n_q \bullet \#(tt' - u \restriction tock) \leq k_q \Rightarrow \#(trace(tt' - u)) \leq n_q\\
&= \{\, \textbf{T5}(Q) \,\}\\
&(\exists\, u \bullet u \leq tt' \land trace(u) = tock^n \land P[u/tt'] \land Q[tt' - u/tt'])\\
&\land tock^n \leq trace(tt') \land \exists\, n_p \bullet \#(u \restriction tock) \leq k_p \Rightarrow \#(trace(u)) \leq n_p\\
&= \{\, \textbf{T5}(P) \,\}\\
&(\exists\, u \bullet u \leq tt' \land trace(u) = tock^n \land P[u/tt'] \land Q[tt' - u/tt'])\\
&\land tock^n \leq trace(tt')\\
&= \{\, r \lhd b \rhd s \land b = r \land b \,\}\\
&\left(\begin{array}{l}(\exists\, u \bullet u \leq tt' \land trace(u) = tock^n \land P[u/tt'] \land Q[tt' - u/tt'])\\ \quad \lhd\, tock^n \leq trace(tt') \,\rhd \\ \quad\quad P\end{array}\right)\\
&\land tock^n \leq trace(tt')\\
&= \{\, \textit{case assumption: } tock^n \leq trace(tt') \,\}
\end{aligned}
$$

$$\left(\begin{array}{l}(\exists u \bullet u \leq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt']) \\ \qquad \lhd tock^n \leq trace(tt') \rhd \\ \qquad\quad P\end{array}\right)$$

$= \{$ definition timeout $\}$

$P \overset{n}{\rhd} Q$

case 2: $\neg\, tock^n \leq trace(tt')$

$\mathbf{T5}(P \overset{n}{\rhd} Q)$

$= \{$ definition timeout $\}$

$$\mathbf{T5}\left(\begin{array}{l}(\exists u \bullet u \leq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt']) \\ \qquad \lhd tock^n \leq trace(tt') \rhd \\ \qquad\quad P\end{array}\right)$$

$= \{$ definition **T5** $\}$

$$\left(\begin{array}{l}(\exists u \bullet u \leq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt']) \\ \qquad \lhd tock^n \leq trace(tt') \rhd \\ \qquad\quad P\end{array}\right)$$
$$\land \exists n \bullet \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(tt')) \leq n$$

$= \{$ case assumption: $\neg\, tock^n \leq trace(tt')$ $\}$

$$\left(\begin{array}{l}(\exists u \bullet u \leq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt']) \\ \qquad \lhd tock^n \leq trace(tt') \rhd \\ \qquad\quad P\end{array}\right)$$
$$\land \exists n \bullet \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(tt')) \leq n$$
$$\land \neg\, tock^n \leq trace(tt')$$

$= \{ r \lhd b \rhd s \land \neg\, b = s \land \neg\, b \}$

$P \land \exists n \bullet \#(tt' \upharpoonright tock) \leq k \Rightarrow \#(trace(tt')) \leq n \land \neg\, tock^n \leq trace(tt')$

$= \{$ assumption: **T5**(P) $\}$

$P \land \neg\, tock^n \leq trace(tt')$

$= \{ r \lhd b \rhd s \land \neg\, b = s \land \neg\, b \}$

$$\left(\begin{array}{l}(\exists u \bullet u \leq tt' \land trace(u) = tock^n \land P[u/tt'] \land Q[tt' - u/tt']) \\ \qquad \lhd tock^n \leq trace(tt') \rhd \\ \qquad\quad P\end{array}\right)$$
$$\land\, tock^n \leq trace(tt')$$

$= \{$ case assumption:$tock^n \leq trace(tt')$ $\}$

$$\left(\begin{array}{l}(\exists u \bullet u \leq tt' \land (trace(u) = tock^n) \land P[u/tt'] \land Q[tt' - u/tt']) \\ \qquad \lhd tock^n \leq trace(tt') \rhd \\ \qquad\quad P\end{array}\right)$$

$= \{$ definition timeout $\}$

$P \overset{n}{\rhd} Q$

2188 8. ***Recursion* T5** *is a conjunctive idempotent, and therefore recursion satisfies* **T5** *by*
2189 *the work in [11].*

Appendix B

Proof of parallel precedence

Lemma B.0.1 (parallel-precedence)

$$[tt' \in s_1 \parallel_A u_1 \land t \preceq tt' \Rightarrow \exists\, s, u \bullet s \preceq s_1 \land u \preceq u_1 \land t \in s \parallel_A u]$$

2192 *Assume*

$$[t \preceq tt' \land tt' \in p \parallel_A q \Rightarrow \exists\, s, u \bullet t \in s \parallel_A u \land s \preceq p \land u \preceq q]$$

and

$$a, b \in A; \;\; c, d \notin A$$

2193 *Proof is by induction on s_1 and u_1.*

2194 *We begin with the cases where tt' must be $\langle\rangle$.*

1. *case:* $s_1 = \langle\rangle \land u_1 = \langle\rangle$

$$\begin{array}{l}
\exists\, s, u \bullet t \in s \parallel_A u \land s \preceq s_1 \land u \preceq u_1 \\
= \{\, \textsf{case: } s_1 = \langle\rangle \,\} \\
\exists\, s, u \bullet t \in s \parallel_A u \land s \preceq \langle\rangle \land u \preceq u_1 \\
= \{\, \textsf{case: } u_1 = \langle\rangle \,\} \\
\exists\, s, u \bullet t \in s \parallel_A u \land s \preceq \langle\rangle \land u \preceq \langle\rangle \\
= \{\, \textsf{precedence: } (w \preceq \langle\rangle) = (w = \langle\rangle) \,\} \\
\exists\, s, u \bullet t \in s \parallel_A u \land s = \langle\rangle \land u = \langle\rangle \\
= \{\, \textsf{one point rule} \,\} \\
t \in \langle\rangle \parallel_A \langle\rangle \\
= \{\, \textsf{definition of } \parallel_A \,\} \\
t \in \{\langle\rangle\} \\
= \{\, t \in \{e\} = (t = e) \,\} \\
t = \langle\rangle \\
\Longleftarrow \{\, \textsf{conjunction} \,\} \\
t = \langle\rangle \land tt' = \langle\rangle \\
= \{\, \textsf{precedence: } \langle\rangle \preceq \langle\rangle \,\} \\
t \preceq tt' \land tt' = \langle\rangle \\
= \{\, (v = w) = (v \in \{w\}) \,\} \\
t \preceq tt' \land tt' \in \{\langle\rangle\} \\
= \{\, \textsf{definition } \parallel_A \,\} \\
t \preceq tt' \land tt' \in \langle\rangle \parallel_A \langle\rangle \\
= \{\, \textsf{case } s_1 = \langle\rangle \,\} \\
t \preceq tt' \land tt' \in s_1 \parallel_A \langle\rangle \\
= \{\, \textsf{case } u_1 = \langle\rangle \,\} \\
t \preceq tt' \land tt' \in s_1 \parallel_A u_1
\end{array}$$

2. case: $s_1 = \langle\rangle \land u_1 = \langle b \rangle \frown y$

$$\exists\, s, u \bullet t \in s \parallel_A u \land s \preceq s_1 \land u \preceq u_1$$
$$\Longleftarrow \{\, \mathit{false} \Rightarrow P \,\}$$
$$\mathit{false}$$
$$= \{\, \text{set theory} \,\}$$
$$t \preceq tt' \land tt' \in \{\}$$
$$= \{\, \text{definition } \parallel_A, b \in A \,\}$$
$$t \preceq tt' \land tt' \in \langle\rangle \parallel_A \langle b \rangle \frown y$$
$$= \{\, \text{case } s_1 = \langle\rangle \,\}$$
$$t \preceq tt' \land tt' \in s_1 \parallel_A \langle b \rangle \frown y$$
$$= \{\, \text{case } u_1 = \langle b \rangle \frown y \,\}$$
$$t \preceq tt' \land tt' \in s_1 \parallel_A u_1$$

3. case: $s_1 = \langle\rangle \land u_1 = \langle d \rangle \frown y$

We make the assumption here that $u_1 \neq \langle\rangle$*. Otherwise* $s_1 = u_1 = \langle\rangle$ *and the case reduces to case 1.*

$$\exists\, s, u \bullet s \preceq s_1 \land u \preceq u_1 \land t \in s \parallel_A u$$
$$= \{\, \text{case: } s_1 = \langle\rangle \,\}$$
$$\exists\, s, u \bullet s \preceq \langle\rangle \land u \preceq u_1 \land t \in s \parallel_A u$$
$$= \{\, \mathit{case} : u_1 = \langle d \rangle \frown y \,\}$$
$$\exists\, s, u \bullet s \preceq \langle\rangle \land u \preceq \langle d \rangle \frown y \land t \in s \parallel_A u$$
$$= \{\, \exists\text{-introduction: } (u \neq \langle\rangle) \,\}$$
$$\exists\, s, u, u_2 \bullet u_2 = u - \langle d \rangle \land t \in s \parallel_A u \land s \preceq \langle\rangle \land u \preceq \langle d \rangle \frown y$$
$$= \{\, \text{sequence difference} \,\}$$
$$\exists\, s, u, u_2 \bullet u = \langle d \rangle \frown u_2 \land t \in s \parallel_A u \land s \preceq \langle\rangle \land u \preceq \langle d \rangle \frown y$$
$$= \{\, \text{one point rule} \,\}$$
$$\exists\, s, u_2 \bullet t \in s \parallel_A \langle d \rangle \frown u_2 \land s \preceq \langle\rangle \land \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y$$
$$\Longleftarrow \{\, \text{precedence: } s = \langle\rangle \Rightarrow s \preceq \langle\rangle \,\}$$
$$\exists\, s, u_2 \bullet t \in s \parallel_A \langle d \rangle \frown u_2 \land s = \langle\rangle \land \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y$$
$$= \{\, \text{one point rule} \,\}$$
$$\exists\, u_2 \bullet t \in \langle\rangle \parallel_A \langle d \rangle \frown u_2 \land \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y$$
$$= \{\, \text{definition of } \parallel_A \,\}$$
$$\exists\, u_2 \bullet t \in \{\langle d \rangle \frown t_2 \mid t_2 \in \langle\rangle \parallel_A u_2\} \land \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y$$
$$= \{\, \text{axiom of comprehension} \,\}$$
$$\exists\, t_2, u_2 \bullet t = \langle d \rangle \frown t_2 \land t_2 \in \langle\rangle \parallel_A u_2 \land \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y$$
$$= \{\, \text{sequence difference} \,\}$$
$$\exists\, t_2, u_2 \bullet t_2 = t - \langle d \rangle \land t_2 \in \langle\rangle \parallel_A u_2 \land \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y$$
$$= \{\, \text{one point rule} \,\}$$

$\exists u_2 \bullet t - \langle d \rangle \in \langle\rangle \parallel_A u_2 \wedge \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y$

$= \{$ case: $s_2 = \langle\rangle \}$

$\exists u_2 \bullet t - \langle d \rangle \in \langle\rangle \parallel_A u_2 \wedge \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y$

$\{$ one point rule $\}$

$\exists s_2, u_2 \bullet t - \langle d \rangle \in s_2 \parallel_A u_2 \wedge \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y \wedge s_2 = \langle\rangle$

$\{ \langle\rangle \preceq \langle\rangle = (\langle\rangle = \langle\rangle) \}$

$\exists s_2, u_2 \bullet t - \langle d \rangle \in s_2 \parallel_A u_2 \wedge \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y \wedge s_2 \preceq \langle\rangle$

$= \{$ definition of $\preceq \}$

$\exists s_2, u_2 \bullet t - \langle d \rangle \in s_2 \parallel_A u_2 \wedge u_2 \preceq y \wedge s_2 \preceq \langle\rangle$

$\Longleftarrow \{$ Induction assumption $\}$

$t - \langle d \rangle \preceq v \wedge v \in \langle\rangle \parallel_A y \wedge \langle\rangle \preceq \langle\rangle$

$= \{ \langle\rangle \preceq \langle\rangle \}$

$t - \langle d \rangle \preceq v \wedge v \in \langle\rangle \parallel_A y$

$= \{$ sequence prefix $\}$

$t \preceq \langle d \rangle \frown v \wedge v \in \langle\rangle \parallel_A y$

$= \{$ one point rule $\}$

$t \preceq tt' \wedge tt' = \langle d \rangle \frown v \wedge v \in \langle\rangle \parallel_A y$

$= \{$ axiom of comprehension $\}$

$t \preceq tt' \wedge tt' \in \{\langle d \rangle \frown v \mid v \in \langle\rangle \parallel_A y\}$

$= \{$ definition of $\parallel_A \}$

$t \preceq tt' \wedge tt' \in \langle\rangle \parallel_A \langle d \rangle \frown y$

$= \{$ case: $u_1 = \langle d \rangle \frown y \}$

$t \preceq tt' \wedge tt' \in \langle\rangle \parallel_A u_1$

$= \{$ case: $s_1 = \langle\rangle \}$

$t \preceq tt' \wedge tt' \in s_1 \parallel_A u_1$

4. case: $s_1 = \langle\rangle \wedge u_1 = \langle T, tock \rangle \frown y$

$\exists s, u \bullet t \in s \parallel_A u \wedge s \preceq s_1 \wedge u \preceq u_1$

$\Longleftarrow \{ false \Rightarrow P \}$

$false$

$= \{$ set theory $\}$

$t \preceq tt' \wedge tt' \in \{\}$

$= \{$ definition $\parallel_A \}$

$t \preceq tt' \wedge tt' \in \langle\rangle \parallel_A \langle T, tock \rangle \frown y$

$= \{$ case $s_1 = \langle\rangle \}$

$t \preceq tt' \wedge tt' \in s_1 \parallel_A \langle T, tock \rangle \frown y$

$= \{$ case $u_1 = \langle T, tock \rangle \frown y \}$

$t \preceq tt' \wedge tt' \in s_1 \parallel_A u_1$

5. *case:* $s_1 = \langle a \rangle \frown x \land u_1 = \langle a \rangle \frown y \land a \in A$

We make the assumptions here that u,s *and* t *are not empty. Otherwise* $s = u = t = \langle\rangle$ *and the case reduces to case 1.*

$$
\begin{array}{l}
\exists\, s, u \bullet s \preceq s_1 \land u \preceq u_1 \land t \in s \parallel_A u \\
= \{\, \text{case: } s_1 = \langle a \rangle \frown x \,\} \\
\exists\, s, u \bullet s \preceq \langle a \rangle \frown x \land u \preceq u_1 \land t \in s \parallel_A u \\
= \{\, case : u_1 = \langle a \rangle \frown y \,\} \\
\exists\, s, u \bullet s \preceq \langle a \rangle \frown x \land u \preceq \langle a \rangle \frown y \land t \in s \parallel_A u \\
\Longleftarrow\ \{\, s \neq \langle\rangle \land s \preceq \langle a \rangle \frown x \Rightarrow \exists\, s_2 \bullet s_2 = s - \langle a \rangle) \,\} \\
\exists\, s, u, s_2, s_2 \bullet s_2 = s - \langle a \rangle \land u_2 = u - \langle a \rangle \land t \in s \parallel_A u \\
\qquad \land\ s \preceq \langle a \rangle \frown x \land u \preceq \langle a \rangle \frown y \\
= \{\, \text{sequence difference} \,\} \\
\exists\, s, u, s_2, u_2 \bullet s = \langle a \rangle \frown s_2 \land u = \langle a \rangle \frown u_2 \land t \in s \parallel_A u \\
\qquad \land\ s \preceq \langle a \rangle \frown x \land u \preceq \langle a \rangle \frown y \\
= \{\, \text{one point rule} \,\} \\
\exists\, s_2, u_2 \bullet t \in \langle a \rangle \frown s_2 \parallel_A \langle a \rangle \frown u_2 \land \langle a \rangle \frown s_2 \preceq \langle a \rangle \frown x \land \langle a \rangle \frown u_2 \preceq \langle a \rangle \frown y \\
= \{\, \text{definition of } \parallel_A \,\} \\
\exists\, s_2, u_2 \bullet t \in \{\langle a \rangle \frown t_2 \mid t_2 \in s_2 \parallel_A u_2\} \land \langle a \rangle \frown s_2 \preceq \langle a \rangle \frown x \\
\qquad \land\ \langle a \rangle \frown u_2 \preceq \langle a \rangle \frown y \\
= \{\, \text{axiom of comprehension} \,\} \\
\exists\, s_2, u_2, t_2 \bullet t = \langle a \rangle \frown t_2 \land t_2 \in s_2 \parallel_A u_2 \land \langle a \rangle \frown s_2 \preceq \langle a \rangle \frown x \\
\qquad \land\ \langle a \rangle \frown u_2 \preceq \langle a \rangle \frown y \\
= \{\, \text{sequence difference} \,\} \\
\exists\, s_2, u_2, t_2 \bullet t_2 = t - \langle a \rangle \land t_2 \in s_2 \parallel_A u_2 \land \langle a \rangle \frown s_2 \preceq \langle a \rangle \frown x \\
\qquad \land\ \langle a \rangle \frown u_2 \preceq \langle a \rangle \frown y \\
= \{\, \text{one point rule} \,\} \\
\exists\, s_2, u_2 \bullet t - \langle a \rangle \in s_2 \parallel_A u_2 \land \langle a \rangle \frown s_2 \preceq \langle a \rangle \frown x \land \langle a \rangle \frown u_2 \preceq \langle a \rangle \frown y \\
= \{\, \text{definition of} \preceq \,\} \\
\exists\, s_2, u_2 \bullet t - \langle a \rangle \in s_2 \parallel_A u_2 \land s_2 \preceq x \land \langle a \rangle \frown u_2 \preceq \langle a \rangle \frown y \\
= \{\, \text{definition of} \preceq \,\} \\
\exists\, s_2, u_2 \bullet t - \langle a \rangle \in s_2 \parallel_A u_2 \land s_2 \preceq x \land u_2 \preceq y \\
\Longleftarrow\ \{\, \text{Induction hypothesis} \,\} \\
t - \langle a \rangle \preceq v \land v \in x \parallel_A y \\
= \{\, \text{sequence prefix (assumption } t \neq \langle\rangle) \,\} \\
t \preceq \langle a \rangle \frown v \land v \in x \parallel_A y \\
= \{\, \text{one point rule} \,\} \\
t \preceq tt' \land tt' = \langle a \rangle \frown v \land v \in x \parallel_A y \\
= \{\, \text{axiom of comprehension} \,\}
\end{array}
$$

$t \preceq tt' \wedge tt' \in \{\langle a \rangle \frown v \mid v \in x \parallel_A y\}$
$= \{$ definition of $\parallel_A$ $\}$
$t \preceq tt' \wedge tt' \in \langle a \rangle \frown x \parallel_A \langle a \rangle \frown y$
$= \{$ case: $u_1 = \langle a \rangle \frown y$ $\}$
$t \preceq tt' \wedge tt' \in \langle a \rangle \frown x \parallel_A u_1$
$= \{$ case: $s_1 = \langle a \rangle \frown x$ $\}$
$t \preceq tt' \wedge tt' \in s_1 \parallel_A u_1$

6. *case:* $s_1 = \langle a \rangle \frown x \wedge u_1 = \langle b \rangle \frown y \wedge a, b \in A$ *for some* x *and* y.

$\exists\, s, u \bullet t \in s \parallel_A u \wedge s \preceq s_1 \wedge u \preceq u_1$
$\Longleftarrow \{$ $false \Rightarrow P$ $\}$
$false$
$= \{$ set theory $\}$
$t \preceq tt' \wedge tt' \in \{\}$
$= \{$ definition $\parallel_A$ $\}$
$t \preceq tt' \wedge tt' \in \langle a \rangle \frown x \parallel_A \langle b \rangle \frown y$
$= \{$ case $s_1 = \langle a \rangle \frown x$ $\}$
$t \preceq tt' \wedge tt' \in s_1 \parallel_A \langle b \rangle \frown y$
$= \{$ case $u_1 = \langle b \rangle \frown y$ $\}$
$t \preceq tt' \wedge tt' \in s_1 \parallel_A u_1$

7. *case:* $s_1 = \langle a \rangle \frown x \wedge u_1 = \langle d \rangle \frown y$.

We make the assumptions here that u, s *and* t *are not empty. Otherwise* $s = u = t = \langle\rangle$ *and the case reduces to case 1.*

$\exists\, s, u \bullet s \preceq s_1 \wedge u \preceq u_1 \wedge t \in s \parallel_A u$
$= \{$ case: $s_1 = \langle a \rangle \frown x$ $\}$
$\exists\, s, u \bullet s \preceq \langle a \rangle \frown x \wedge u \preceq u_1 \wedge t \in s \parallel_A u$
$= \{$ $case: u_1 = \langle d \rangle \frown y$ $\}$
$\exists\, s, u \bullet s \preceq \langle a \rangle \frown x \wedge u \preceq \langle d \rangle \frown y \wedge t \in s \parallel_A u$
$= \{$ $\exists$-introduction: $(s \neq \langle\rangle, u \neq \langle\rangle)$ $\}$
$\exists\, s, u, s_2, s_2 \bullet s_2 = s - \langle a \rangle \wedge u_2 = u - \langle d \rangle \wedge t \in s \parallel_A u$
$\quad \wedge\, s \preceq \langle a \rangle \frown x \wedge u \preceq \langle d \rangle \frown y$
$= \{$ sequence difference $\}$
$\exists\, s, u, s_2, u_2 \bullet s = \langle a \rangle \frown s_2 \wedge u = \langle d \rangle \frown u_2 \wedge t \in s \parallel_A u$
$\quad \wedge\, s \preceq \langle a \rangle \frown x \wedge u \preceq \langle d \rangle \frown y$
$= \{$ one point rule $\}$
$\exists\, s_2, u_2 \bullet t \in \langle a \rangle \frown s_2 \parallel_A \langle d \rangle \frown u_2 \wedge \langle a \rangle \frown s_2 \preceq \langle a \rangle \frown x \wedge \langle a \rangle \frown u_2 \preceq \langle d \rangle \frown y$
$= \{$ definition of $\parallel_A$ $\}$

$\exists\, s_2, u_2 \bullet t \in \{\langle d\rangle \frown t_2 \mid t_2 \in \langle a\rangle \frown s_2 \parallel_A u_2\}$
$\quad \land \langle a\rangle \frown s_2 \preceq \langle a\rangle \frown x \land \langle d\rangle \frown u_2 \preceq \langle d\rangle \frown y$
$=$ { *axiom of comprehension* }
$\exists\, s_2, u_2, t_2 \bullet t = \langle d\rangle \frown t_2 \land t_2 \in \langle a\rangle \frown s_2 \parallel_A u_2$
$\quad \land \langle a\rangle \frown s_2 \preceq \langle a\rangle \frown x \land \langle d\rangle \frown u_2 \preceq \langle d\rangle \frown y$
$=$ { *sequence difference* }
$\exists\, s_2, u_2, t_2 \bullet t_2 = t - \langle d\rangle \land t_2 \in \langle a\rangle \frown s_2 \parallel_A u_2 \land$
$\quad \langle a\rangle \frown s_2 \preceq \langle a\rangle \frown x \land \langle d\rangle \frown u_2 \preceq \langle d\rangle \frown y$
$=$ { *one point rule* }
$\exists\, s_2, u_2 \bullet t - \langle d\rangle \in \langle a\rangle \frown s_2 \parallel_A u_2 \land \langle a\rangle \frown s_2 \preceq \langle a\rangle \frown x \land \langle d\rangle \frown u_2 \preceq \langle d\rangle \frown y$
$=$ { *definition of* $\preceq$ }
$\exists\, s_2, u_2 \bullet t - \langle d\rangle \in \langle a\rangle \frown s_2 \parallel_A u_2 \land \langle a\rangle \frown s_2 \preceq \langle a\rangle \frown x \land u_2 \preceq y$
$\Longleftarrow$ { *Induction hypothesis* }
$t - \langle d\rangle \preceq v \land v \in \langle a\rangle \frown x \parallel_A y$
$=$ { *sequence prefix* ($t \neq \langle\rangle$) }
$t \preceq \langle d\rangle \frown v \land v \in \langle a\rangle \frown x \parallel_A y$
$=$ { *one point rule* }
$t \preceq tt' \land tt' = \langle d\rangle \frown v \land v \in \langle a\rangle \frown x \parallel_A y$
$=$ { *axiom of comprehension* }
$t \preceq tt' \land tt' \in \{\langle d\rangle \frown v \mid v \in \langle a\rangle \frown x \parallel_A y\}$
$=$ { *definition of* $\parallel_A$ }
$t \preceq tt' \land tt' \in \langle a\rangle \frown x \parallel_A \langle d\rangle \frown y$
$=$ { *case:* $u_1 = \langle d\rangle \frown y$ }
$t \preceq tt' \land tt' \in \langle a\rangle \frown x \parallel_A u_1$
$=$ { *case:* $s_1 = \langle a\rangle \frown x$ }
$t \preceq tt' \land tt' \in s_1 \parallel_A u_1$

2207 8. *case:* $s_1 = \langle a\rangle \frown x \land u_1 = \langle T, tock\rangle \frown y \land a \in A$ *for some* x *and* y.

$\exists\, s, u \bullet t \in s \parallel_A u \land s \preceq s_1 \land u \preceq u_1$
$\Longleftarrow$ { $false \Rightarrow P$ }
$false$
$=$ { *set theory* }
$t \preceq tt' \land tt' \in \{\}$
$=$ { *definition* $\parallel_A$ }
$t \preceq tt' \land tt' \in \langle a\rangle \frown x \parallel_A \langle T, tock\rangle \frown y$
$=$ { *case* $s_1 = \langle a\rangle \frown x$ }
$t \preceq tt' \land tt' \in s_1 \parallel_A \langle T, tock\rangle \frown y$
$=$ { *case* $u_1 = \langle T, tock\rangle \frown y$ }
$t \preceq tt' \land tt' \in s_1 \parallel_A u_1$

9. *case:* $s_1 = \langle c \rangle \frown x \wedge u_1 = \langle d \rangle \frown y$.

We make the assumptions here that u, s and t are not empty. Otherwise $s = u = t = \langle\rangle$ and the case reduces to case 1.

$$
\begin{array}{l}
\exists\, s, u \bullet s \preceq s_1 \wedge u \preceq u_1 \wedge t \in s \parallel_A u \\
= \{ \textit{case: } s_1 = \langle c \rangle \frown x \} \\
\exists\, s, u \bullet s \preceq \langle c \rangle \frown x \wedge u \preceq u_1 \wedge t \in s \parallel_A u \\
= \{ \textit{case} : u_1 = \langle d \rangle \frown y \} \\
\exists\, s, u \bullet s \preceq \langle c \rangle \frown x \wedge u \preceq \langle d \rangle \frown y \wedge t \in s \parallel_A u \\
= \{ \exists\textit{-introduction} \} \\
\exists\, s, u, s_2, s_2 \bullet s_2 = s - \langle c \rangle \wedge u_2 = u - \langle d \rangle \wedge t \in s \parallel_A u \\
\qquad \wedge\ s \preceq \langle c \rangle \frown x \wedge u \preceq \langle d \rangle \frown y \\
= \{ \textit{sequence difference} \} \\
\exists\, s, u, s_2, u_2 \bullet s = \langle c \rangle \frown s_2 \wedge u = \langle d \rangle \frown u_2 \wedge t \in s \parallel_A u \\
\qquad \wedge\ s \preceq \langle c \rangle \frown x \wedge u \preceq \langle d \rangle \frown y \\
= \{ \textit{one point rule} \} \\
\exists\, s_2, u_2 \bullet t \in \langle c \rangle \frown s_2 \parallel_A \langle d \rangle \frown u_2 \wedge \langle c \rangle \frown s_2 \preceq \langle c \rangle \frown x \wedge \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y \\
= \{ \textit{definition of } \parallel_A \} \\
\exists\, s_2, u_2 \bullet t \in \{ \langle d \rangle \frown t_2 \mid t_2 \in \langle c \rangle \frown s_2 \parallel_A u_2 \} \cup \{ \langle c \rangle \frown t_2 \mid t_2 \in s_2 \parallel_A \langle d \rangle \frown u_2 \} \\
\qquad \wedge\ \langle c \rangle \frown s_2 \preceq \langle c \rangle \frown x \wedge \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y \\
= \{ \textit{axiom of comprehension} \} \\
\exists\, s_2, u_2, t_2 \bullet (t = \langle d \rangle \frown t_2 \wedge t_2 \in \langle c \rangle \frown s_2 \parallel_A u_2) \\
\qquad \vee (t = \langle c \rangle \frown t_2 \wedge t_2 \in s_2 \parallel_A \langle d \rangle \frown u_2) \\
\wedge\ \langle c \rangle \frown s_2 \preceq \langle c \rangle \frown x \wedge \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y \\
= \{ \textit{sequence difference twice} \} \\
\exists\, s_2, u_2, t_2 \bullet (t_2 = t - \langle d \rangle \wedge t_2 \in \langle c \rangle \frown s_2 \parallel_A u_2) \\
\qquad \vee (t_2 = t - \langle c \rangle \wedge t_2 \in s_2 \parallel_A \langle d \rangle \frown u_2) \\
\wedge\ \langle c \rangle \frown s_2 \preceq \langle c \rangle \frown x \wedge \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y \\
= \{ \textit{one point rule twice} \} \\
\exists\, s_2, u_2 \bullet (t - \langle d \rangle \in \langle c \rangle \frown s_2 \parallel_A u_2) \vee (t - \langle c \rangle \in s_2 \parallel_A \langle d \rangle \frown u_2) \\
\qquad \wedge\ \langle c \rangle \frown s_2 \preceq \langle c \rangle \frown x \wedge \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y \\
= \{ \textit{definition of } \preceq \} \\
\exists\, s_2, u_2 \bullet (t - \langle d \rangle \in \langle c \rangle \frown s_2 \parallel_A u_2) \vee (t - \langle c \rangle \in s_2 \parallel_A \langle d \rangle \frown u_2) \\
\wedge\ \langle c \rangle \frown s_2 \preceq \langle c \rangle \frown x \wedge \langle d \rangle \frown u_2 \preceq \langle d \rangle \frown y \\
\qquad \wedge\ s_2 \preceq x \wedge u_2 \frown y \\
\Longleftarrow \{ \textit{Induction hypothesis twice} \} \\
t - \langle d \rangle \preceq v \wedge v \in \langle c \rangle \frown x \parallel_A y\ \vee \\
\qquad t - \langle c \rangle \preceq v \wedge v \in x \parallel_A \langle d \rangle \frown y \\
= \{ \textit{sequence prefix } (t \neq \langle\rangle) \}
\end{array}
$$

$$t \preceq \langle d \rangle \frown v \wedge v \in \langle d \rangle \frown x \parallel_A y \vee$$
$$\quad t \preceq \langle c \rangle \frown v \wedge v \in x \parallel_A \langle c \rangle \frown y$$
$$= \{ \textit{one point rule} \}$$
$$t \preceq tt' \wedge tt' = \langle d \rangle \frown v \wedge v \in \langle c \rangle \frown x \parallel_A y \vee$$
$$\quad t \preceq tt' \wedge tt' = \langle c \rangle \frown v \wedge v \in x \parallel_A \langle d \rangle \frown y$$
$$= \{ \textit{axiom of comprehension} \}$$
$$t \preceq tt' \wedge tt' \in \{\langle d \rangle \frown v \mid v \in \langle c \rangle \frown x \parallel_A y\} \cup \{\langle c \rangle \frown v \mid v \in x \parallel_A \langle d \rangle \frown y\}$$
$$= \{ \textit{definition of } \parallel_A \}$$
$$t \preceq tt' \wedge tt' \in \langle c \rangle \frown x \parallel_A \langle d \rangle \frown y$$
$$= \{ \textit{case: } u_1 = \langle d \rangle \frown y \}$$
$$t \preceq tt' \wedge tt' \in \langle c \rangle \frown x \parallel_A u_1$$
$$= \{ \textit{case: } s_1 = \langle c \rangle \frown x \}$$
$$t \preceq tt' \wedge tt' \in s_1 \parallel_A u_1$$

10. *case:* $s_1 = \langle c \rangle \frown x \wedge u_1 = \langle T, tock \rangle \frown y$. *We make the assumptions here that* u,s *and* t *are not empty. Otherwise* $s = u = t = \langle\rangle$ *and the case reduces to case 1.*

$$\exists\, s, u \bullet s \preceq s_1 \wedge u \preceq u_1 \wedge t \in s \parallel_A u$$
$$= \{ \textit{case: } s_1 = \langle c \rangle \frown x \}$$
$$\exists\, s, u \bullet s \preceq \langle c \rangle \frown x \wedge u \preceq u_1 \wedge t \in s \parallel_A u$$
$$= \{ case : u_1 = \langle T, tock \rangle \frown y \}$$
$$\exists\, s, u \bullet s \preceq \langle c \rangle \frown x \wedge u \preceq \langle T, tock \rangle \frown y \wedge t \in s \parallel_A u$$
$$= \{ \exists\textit{-introduction} \}$$
$$\exists\, s, u, s_2, s_2 \bullet s_2 = s - \langle c \rangle \wedge u_2 = u - \langle T, tock \rangle \wedge t \in s \parallel_A u \wedge s \preceq \langle c \rangle \frown x$$
$$\quad \wedge\, u \preceq \langle T, tock \rangle \frown y$$
$$= \{ \textit{sequence difference} \}$$
$$\exists\, s, u, s_2, u_2 \bullet s = \langle c \rangle \frown s_2 \wedge u = \langle T, tock \rangle \frown u_2 \wedge t \in s \parallel_A u \wedge s \preceq \langle c \rangle \frown x \wedge$$
$$\quad u \preceq \langle T, tock \rangle \frown y$$
$$= \{ \textit{one point rule} \}$$
$$\exists\, s_2, u_2 \bullet t \in \langle c \rangle \frown s_2 \parallel_A \langle T, tock \rangle \frown u_2 \wedge \langle c \rangle \frown s_2 \preceq \langle c \rangle \frown x$$
$$\quad \wedge\, \langle c \rangle \frown u_2 \preceq \langle T, tock \rangle \frown y$$
$$= \{ \textit{definition of } \parallel_A \}$$
$$\exists\, s_2, u_2 \bullet t \in \{\langle c \rangle \frown t_2 \mid t_2 \in s_2 \parallel_A \langle T, tock \rangle \frown u_2\} \wedge \langle c \rangle \frown s_2 \preceq \langle c \rangle \frown x \wedge$$
$$\quad \langle T, tock \rangle \frown u_2 \preceq \langle T, tock \rangle \frown y$$
$$= \{ \textit{axiom of comprehension} \}$$
$$\exists\, s_2, u_2, t_2 \bullet t = \langle c \rangle \frown t_2 \wedge t_2 \in s_2 \parallel_A \langle T, tock \rangle \frown u_2 \wedge \langle c \rangle \frown s_2 \preceq \langle c \rangle \frown x \wedge$$
$$\quad \langle T, tock \rangle \frown u_2 \preceq \langle T, tock \rangle \frown y$$
$$= \{ \textit{sequence difference} \}$$
$$\exists\, s_2, u_2, t_2 \bullet t_2 = t - \langle c \rangle \wedge t_2 \in s_2 \parallel_A \langle T, tock \rangle \frown u_2 \wedge \langle c \rangle \frown s_2 \preceq \langle c \rangle \frown x \wedge$$
$$\quad \langle T, tock \rangle \frown u_2 \preceq \langle T, tock \rangle \frown y$$

$$= \{ \textit{one point rule} \}$$
$$\exists\, s_2, u_2 \bullet t - \langle c \rangle \in s_2 \parallel_A \langle T, tock \rangle \frown u_2 \wedge \langle c \rangle \frown s_2 \preceq \langle c \rangle \frown x \wedge$$
$$\langle T, tock \rangle \frown u_2 \preceq \langle T, tock \rangle \frown y$$
$$= \{ \textit{definition of} \preceq \}$$
$$\exists\, s_2, u_2 \bullet t - \langle c \rangle \in s_2 \parallel_A \langle T, tock \rangle \frown u_2 \wedge s_2 \preceq x \wedge$$
$$\langle T, tock \rangle \frown u_2 \preceq \langle T, tock \rangle \frown y$$
$$\Longleftarrow \{ \textit{Induction hypothesis} \}$$
$$t - \langle c \rangle \preceq v \wedge v \in x \parallel_A \langle T, tock \rangle \frown y$$
$$= \{ \textit{sequence prefix } (t \neq \langle\rangle) \}$$
$$t \preceq \langle c \rangle \frown v \wedge v \in x \parallel_A \langle T, tock \rangle \frown y$$
$$= \{ \textit{one point rule} \}$$
$$t \preceq tt' \wedge tt' = \langle c \rangle \frown v \wedge v \in x \parallel_A \langle T, tock \rangle \frown y$$
$$= \{ \textit{axiom of comprehension} \}$$
$$t \preceq tt' \wedge tt' \in \{ \langle c \rangle \frown v \mid v \in x \parallel_A \langle T, tock \rangle \frown y \}$$
$$= \{ \textit{definition of} \parallel_A \}$$
$$t \preceq tt' \wedge tt' \in \langle c \rangle \frown x \parallel_A \langle T, tock \rangle \frown y$$
$$= \{ \textit{case: } u_1 = \langle T, tock \rangle \frown y \}$$
$$t \preceq tt' \wedge tt' \in \langle c \rangle \frown x \parallel_A u_1$$
$$= \{ \textit{case: } s_1 = \langle c \rangle \frown x \}$$
$$t \preceq tt' \wedge tt' \in s_1 \parallel_A u_1$$

11. *case:* $s_1 = \langle S, tock \rangle \frown x \wedge u_1 = \langle T, tock \rangle \frown y$.

We make the assumptions here that u,s and t are not empty. Otherwise $s = u = t = \langle\rangle$ *and the case reduces to case 1.*

$$\exists\, s, u \bullet s \preceq s_1 \wedge u \preceq u_1 \wedge t \in s \parallel_A u$$
$$= \{ \textit{case: } s_1 = \langle S, tock \rangle \frown x \}$$
$$\exists\, s, u \bullet s \preceq \langle S, tock \rangle \frown x \wedge u \preceq u_1 \wedge t \in s \parallel_A u$$
$$= \{ case : u_1 = \langle T, tock \rangle \frown y \}$$
$$\exists\, s, u \bullet s \preceq \langle S, tock \rangle \frown x \wedge u \preceq \langle T, tock \rangle \frown y \wedge t \in s \parallel_A u$$
$$= \{ \exists\textit{-introduction: } (s \neq \langle\rangle, u \neq \langle\rangle) \}$$
$$\exists\, s, u, s_2, s_2 \bullet s_2 = s - \langle S, tock \rangle \wedge u_2 = u - \langle T, tock \rangle \wedge t \in s \parallel_A u \;\wedge$$
$$s \preceq \langle S, tock \rangle \frown x \;\wedge u \preceq \langle T, tock \rangle \frown y$$
$$= \{ \textit{sequence difference} \}$$
$$\exists\, s, u, s_2, u_2 \bullet s = \langle S, tock \rangle \frown s_2 \wedge u = \langle T, tock \rangle \frown u_2 \wedge t \in s \parallel_A u \;\wedge$$
$$s \preceq \langle S, tock \rangle \frown x \wedge u \preceq \langle Tock \rangle \frown y$$
$$= \{ \textit{one point rule} \}$$
$$\exists\, s_2, u_2 \bullet t \in \langle S, tock \rangle \frown s_2 \parallel_A \langle T, tock \rangle \frown u_2 \;\wedge$$
$$\langle S, tock \rangle \frown s_2 \preceq \langle S, tock \rangle \frown x \wedge \langle T, tock \rangle \frown u_2 \preceq \langle T, tock \rangle \frown y$$
$$= \{ \textit{definition of} \parallel_A \}$$

$\exists s_2, u_2, t_2 \bullet t \in \{\langle U, tock\rangle \frown t_2 \mid U \in S \cap_A T \wedge t_2 \in s_2 \parallel_A u_2\} \;\wedge$
$\langle S, tock\rangle \frown s_2 \preceq \langle S, tock\rangle \frown x \wedge \langle T, tock\rangle \frown u_2 \preceq \langle T, tock\rangle \frown y$
$=$ { *axiom of comprehension* }
$\exists s_2, u_2, t_2 \bullet t = \langle U, tock\rangle \frown t_2 \wedge U \in S \cap_A T \wedge t_2 \in s_2 \parallel_A u_2 \;\wedge$
$\langle S, tock\rangle \frown s_2 \preceq \langle S, tock\rangle \frown x \wedge \langle T, tock\rangle \frown u_2 \preceq \langle T, tock\rangle \frown y$
$=$ { *sequence difference* }
$\exists s_2, u_2, t_2 \bullet t_2 = t - \langle U, tock\rangle \wedge U \in S \cap_A T \wedge t_2 \in s_2 \parallel_A u_2 \;\wedge$
$\langle S, tock\rangle \frown s_2 \preceq \langle S, tock\rangle \frown x \wedge \langle T, tock\rangle \frown u_2 \preceq \langle T, tock\rangle \frown y$
$=$ { *one point rule* }
$\exists s_2, u_2 \bullet \; U \in S \cap_A T \wedge t - \langle U, tock\rangle \in s_2 \parallel_A u_2 \;\wedge$
$\qquad \langle S, tock\rangle \frown s_2 \preceq \langle S, tock\rangle \frown x \wedge \langle T, tock\rangle \frown u_2 \preceq \langle T, tock\rangle \frown y$
$=$ { *definition of* $\preceq$ *twice* }
$\exists s_2, u_2 \bullet U \in S \cap_A T \wedge t - \langle U, tock\rangle \in s_2 \parallel_A u_2 \wedge s_2 \preceq x \wedge u_2 \preceq y$
$\Longleftarrow$ { *Induction hypothesis* }
$U \in S \cap_A T \wedge t - \langle U, tock\rangle \preceq v \wedge v \in x \parallel_A y$
$=$ { *sequence prefix (assumption t* $\neq \langle\rangle$*)* }
$U \in S \cap_A T \wedge t \preceq \langle U, tock\rangle \frown v \wedge v \in x \parallel_A y$
$=$ { *one point rule* }
$U \in S \cap_A T \wedge t \preceq tt' \wedge tt' = \langle U, tock\rangle \frown v \wedge v \in x \parallel_A y$
$=$ { *axiom of comprehension* }
$t \preceq tt' \wedge tt' \in \{\langle U, tock\rangle \frown v \mid U \in S \cap_A T \wedge v \in x \parallel_A y\}$
$=$ { *definition of* $\parallel_A$ }
$t \preceq tt' \wedge tt' \in \langle S, tock\rangle \frown x \parallel_A \langle T, tock\rangle \frown y$
$=$ { *case:* $u_1 = \langle T, tock\rangle \frown y$ }
$t \preceq tt' \wedge tt' \in \langle S, tock\rangle \frown x \parallel_A u_1$
$=$ { *case:* $s_1 = \langle S, tock\rangle \frown x$ }
$t \preceq tt' \wedge tt' \in s_1 \parallel_A u_1$