SEVENTH FRAMEWORK PROGRAMME

Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

CML Definition 4

Deliverable Number: D23.5

Version: 1

Date: 31 March 2014

Public Document

http://www.compass-research.eu

Contributors:

Jeremy Bryans, Newcastle
Samuel Canham, York
Jim Woodcock, York

Editors:

Jeremy Bryans, Newcastle

Reviewers:

Zoe Andrews, Newcastle
Joey Coleman, Aarhus
Marcel Oliveira, UFPE

Document History

Ver	Date	Author	Description
0.1	02-10-2013	Jeremy Bryans	Initial document version
0.2	04-02-2014	Jeremy Bryans	Taking account of internal discussions
0.3	18-02-2014	Jeremy Bryans	Minor changes to text
0.4	28-02-2014	Jeremy Bryans	Addition of JW's chapter on reference semantics
1	23-03-2014	Jeremy Bryans	Comments from reviewers incorporated

Abstract

This report contains the fourth iteration 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 [39], (with a subsequent update in D31.2c [11], and a further one in D31.3c [12]) and previous versions of the semantics can be found in D23.2 [6], D23.3 [3] and D23.4 [4]. D23.4 has four parts: (i) a Denotational Semantics (D23.4a); (ii) a version of the operational semantics for the kernel language (D23.4b); (iii) a semantics for the object-oriented features of *CML*, given a formal treatment in D23.4c; and (iv) a Hoare-logic for *CML* is given in (D23.4d). This document refines and extends D23.4a.

We start with a summary of the relevant theories from UTP: the alphabetised relational calculus and the theory of designs. Next, we give background on how Galois connections can be used to link the different language paradigms of *CML*. Next, we give a denotational semantics to *CML*, including stateful reactive constructs. This is a revised and updated form of material in D23.3 and D23.4a. Our semantics covers the core *CML* language, so the correspondences between additional language features are treated either as a shallow embedding of the *CML* expression language in UTP, or as derived operators. Example Galois connections between language paradigms are given, and finally we present some initial work on a UTP theory of references based on separation logic.

Contents

Correspondence between the CML notation in this deliverable and the CML tool notation.

Name	CML mathematics	CML tool notation
Deadlock	$STOP$	**`Stop`**
Termination	$SKIP$	**`Skip`**
Assignment	$v := e$	`v := e`
Specification	$P \vdash Q$	[**`pre`** `P` **`post`** `Q`]
Prefixed skip	$a \rightarrow SKIP$	`a ->` **`Skip`**
Divergence	$CHAOS$	**`Chaos`**
Sequential composition	$P \mathbin{;} Q$	`P ; Q`
Prefixed action	$a \rightarrow P$	`a -> P`
Input prefix	$a?x \rightarrow P$	`a?x -> P`
Input filter	$a?x : p \rightarrow P$	`a?x:(p) -> P`
Output prefix	$a!x \rightarrow P$	`a!x -> P`
Multiple prefix	$a!x!y \rightarrow P$	`a!x!y -> P`
Conditional	$P \lhd b \rhd Q$	**`if`** `b` **`then`** `P` **`else`** `Q`
Internal choice	$P \sqcap Q$	`P |~| Q`
External choice	$P \Box Q$	`P [] Q`
Parallel composition	$P \parallel_{\{cs\}} Q$	`P [|{|cs|}|] Q`, or `P [|{cs}|] Q`
Interleaving parallel	$P \interleave Q$	`P|||Q`
Hiding	$P \setminus A$	`P \\ A`
Recursion	$\mu X \bullet P(X)$	**`mu`** `X,Y,...` **`@`** `(P,Q,...)`
Timeout	$P \overset{n}{\rhd} Q$	`P [_n_> Q`
Untimed timeout	$P \rhd Q$	`P [_> Q`
Wait	$Wait(n)$	**`Wait`**`(n)`
Interrupt	$P \bigtriangleup Q$	`P /_\ Q`
Timed interrupt	$P \overset{n}{\bigtriangleup} Q$	`P /_n_\ Q`
Startsby	$P \; startsby(n)$	`P` **`startsby`** `n`
Endsby	$P \; endsby(n)$	`P` **`endsby`** `n`
While	$b * P$	**`while`** `b` **`do`** `P`
Guarded actions	$[g] \,\&\, P$	`[g] & P`

Chapter 1

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 [14], CSP [32], and *Circus* [24, 17]. The main objective of work package WP23 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.

Our chosen formalism for this work is Hoare & He's Unifying Theories of Programming (UTP) [16]. In Chapter 2, 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 2.3, we describe UTP's fundamental theory: the alphabetised relational calculus. In Section 2.5, 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. This material is carried forward from previous deliverables.

In Chapter 3 we give background on how Galois connections can be used to link the different language paradigms of *CML*.

We describe the denotational semantics for the behavioural kernel of *CML* in Chapter 4, including imperative reactive processes. This is a revised and updated form of material in D23.3 and D23.4. 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. 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.

The approach taken is based on Lowe & Ouaknine's timed testing traces semantics for CSP [20]. The major changes between their work and ours are the addition of sequential composition and a specification statement that corresponds to a VDM operation. Lowe & Ouaknine use a closed presentation for the semantics, following an established tradition.[1] In Section 4.2, we discuss their axioms and posit most of them as theorems of *CML*'s

[1] The standard semantics for CSP [32] defines a process as a pair of sets; this is a closed presentation. The presentation in UTP is based on the characteristic predicates of these two sets, with free variables for the semantic objects; this is known as an open presentation.

basic semantics. ***CML*** is strictly more powerful than 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. In Chapter 5 we present some results on Galois connections between different ***CML*** language paradigms. In Chapter 6 we introduce a UTP theory of references, based on work on separation logic.

Some of the binary forms of composition operators given in Chapter 4 may be generalised. An appendix contains the semantics of these generalisations, as well as a discussion of expressions within ***CML.***

These material updates that in [5] by:

- adding healthiness conditions ***RT6*** and ***RT7***, which correspond to ***CSP3*** and ***CSP4*** respectively
- updating the definition of *SKIP* to remove the insistence that it terminates before any other observations are made,
- updating the definition of external choice to make the set of valid preconditions more precise,
- adding a chapter on reference semantics.

This document is produced as output to Task 2.3.1 within Work Package 23 [13]. 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. An initial syntax definition for ***CML*** can be found in [39], and an updated grammar in [12].

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 2 on analysis techniques and to tasks within Theme 3 on tool development. Feedback from these tasks has been taken into account in this document.

The semantic approach taken is that set out by Hoare & He in their book Unifying Theories of Programming [16]. 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.

This is the final semantics document to be produced within COMPASS.

Chapter 2

Unifying Theories of Programming

2.1 Background

Unifying Theories of Programming is originally the work of Hoare & He [16]. 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 [15] 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 ([29, 41]), hardware/software co-design ([7]) and component-based systems ([40]). 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 [37].

2.2 Introduction

The book by Hoare & He [16] 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 [38] 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 2.2.1 (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, $P = P \land B$, and then the following function on predicates $and_B \mathrel{\widehat{=}} \lambda P \bullet P \land B$ 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 [16]. 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 2.5 restricts the general theory to designs. Next, in Section 2.6, 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.

2.3 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, ...) and dashed variables (x', a', ...); 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 [16]: 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 given as $\forall a, b, a', b' \bullet P$ 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 2.3.1 (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 2.3.2 (Proof of Unreachable Branch (*L6*))

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

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 2.3.1 (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 2.3.1

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

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

A compositional argument is also available for conjunctions.

Law 2.3.2 (Separation of requirements)

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

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.

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

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.

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

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

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

Skip is the identity of sequence.

$$\begin{array}{lll} \textbf{\textit{L5}} & P \,;\, I\!I_{\alpha P} = P = I\!I_{\alpha P} \,;\, P & \textit{unit} \end{array}$$

We keep the numbers of the laws presented in [16] 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.

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

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

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 2.3.3 (Refinement and nondeterminism)

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

Proof

$$\begin{array}{ll}
\quad P \sqcap Q = P & \textit{[antisymmetry]} \\
= (P \sqcap Q \sqsubseteq P) \land (P \sqsubseteq P \sqcap Q) & \textit{[definition of } \sqsubseteq\textit{, twice]} \\
= [\, P \Rightarrow P \sqcap Q \,] \land [\, P \sqcap Q \Rightarrow P \,] & \textit{[definition of } \sqcap\textit{, twice]} \\
= [\, P \Rightarrow P \lor Q \,] \land [\, P \lor Q \Rightarrow P \,] & \textit{[propositional calculus]} \\
= \mathit{true} \land [\, P \lor Q \Rightarrow P \,] & \textit{[propositional calculus]} \\
= [\, Q \Rightarrow P \,] & \textit{[definition of } \sqsubseteq\textit{]} \\
= P \sqsubseteq Q & \square
\end{array}$$

Another fundamental result is that reducing nondeterminism leads to refinement.

Law 2.3.4 (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.

L6 $(\,\textbf{var}\,x\,;\,\textbf{end}\,x\,) = I\!I$

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

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}.

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

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

L1 $\textbf{var}\,x\,;\,Q_{+x}\,;\,P\,;\,\textbf{end}\,x \;=\; Q\,;\,\textbf{var}\,x\,;\,P\,;\,\textbf{end}\,x$

L2 $\textbf{var}\,x\,;\,P\,;\,Q_{+x}\,;\,\textbf{end}\,x \;=\; \textbf{var}\,x\,;\,P\,;\,\textbf{end}\,x\,;\,Q$

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.

2.4 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 $\bot_A$, 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.

L1 $\bot_A \sqsubseteq P$ *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 2.4.1 (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.*

$$
\begin{array}{ll}
P \odot R & \text{[assumption } (P \sqsubseteq Q) \text{ and Law 2.3.3]} \\
= (P \sqcap Q) \odot R & \text{[assumption } (\odot \text{ disjunctive})] \\
= (P \odot R) \sqcap (Q \odot R) & \text{[thin nondeterminism]} \\
\sqsubseteq Q \odot R &
\end{array}
$$

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 $X = P \mathbin{;} X$, and $\bot$ is the weakest.

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

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

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.

$$\mu X \bullet \mathcal{C}(X) \mathrel{\widehat=} \mu F \qquad \textbf{where } F \mathrel{\widehat=} \lambda X \bullet \mathcal{C}(X)$$

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*

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

Proof of *L2*

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

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 2.4.2 (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 $\mu\, X \bullet X$.*

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

A surprising, but simple, consequence of Example 2.4.2 is that a program can recover from a non-terminating loop!

Example 2.4.3 (Aborting loop) *Suppose that the sole state variable is x and that c is a constant.*

$$
\begin{array}{ll}
(b * P);\ x := c & \text{[Example 2.4.2]} \\
= \bot;\ x := c & \text{[definition of } \bot\text{]} \\
= \textbf{true};\ x := c & \text{[definition of assignment]} \\
= \textbf{true};\ x' = c & \text{[definition of composition]} \\
= \exists\, x_0 \bullet \textbf{true} \wedge x' = c & \text{[predicate calculus]} \\
= x' = c & \text{[definition of assignment]} \\
= x := c & \square
\end{array}
$$

Example 2.4.3 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 2.5.

2.5 Designs

The problem pointed out in Section 2.4 can be explained as the failure of general alphabetised predicates P to satisfy the equation below.

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

In particular, in Example 2.4.3 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 $\textbf{false} = \textbf{false} ; \textbf{true}$.

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 [18], Z [38], and refinement calculi [21, 2, 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.

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

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 2.5.1 (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 2.5.2 (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 2.5.1 *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.

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

Proof

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

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

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

$$I\!I_D \;\widehat{=}\; (\,\textbf{true} \vdash I\!I\,)$$

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

$$
\begin{array}{ll}
\textbf{\textit{L2}} & (v := e \,;\, v := f(v)) = (v := f(e)) \\
\textbf{\textit{L3}} & (v := e \,;\, (P \lhd b(v) \rhd Q)) = ((v := e \,;\, P) \lhd b(e) \rhd (v := e \,;\, Q)) \\
\textbf{\textit{L4}} & (I\!I_D \,;\, (P \vdash Q)) = (P \vdash Q)
\end{array}
$$

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

Proof of *L2*

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

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.

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

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.

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

where $Q_1 \,\textbf{wp}\, P_2$ is the weakest precondition under which execution of Q_1 is guaranteed to achieve the postcondition P_2. It is defined in [16] as

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

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.

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

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.

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

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

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).

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

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

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

Theorem 2.5.1 *Meets and joins*

$$\sqcap_i (P_i \vdash Q_i) = (\bigwedge_i P_i) \vdash (\bigvee_i Q_i)$$

$$\sqcup_i (P_i \vdash Q_i) = (\bigvee_i P_i) \vdash (\bigwedge_i P_i \Rightarrow Q_i)$$

As with the binary choice, the choice $\sqcap_i (P_i \vdash Q_i)$ 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.

2.6 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.

2.6.1 *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.

$$\textbf{true} \; ; R = \textbf{true} \quad \text{and} \quad I\!I_D \; ; R = R$$

We now present a proof of these results.

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

$$\begin{array}{ll}
R & \text{[assumption } (I\!I_D \text{ is left-unit)]} \\
= I\!I_D \; ; R & [I\!I_D \text{ definition]} \\
= (\textbf{true} \vdash I\!I_D) \; ; R & \text{[design definition]} \\
= (ok \Rightarrow ok' \wedge I\!I) \; ; R & \text{[relational calculus]} \\
= (\neg ok \; ; R) \vee (I\!I \; ; R) & \text{[relational calculus]} \\
= (\neg ok \; ; \textbf{true} \; ; R) \vee (I\!I \; ; R) & \text{[assumption } (\textbf{true} \text{ is left-zero)]} \\
= \neg ok \vee (I\!I \; ; R) & \text{[assumption } (I\!I \text{ is left-unit)]} \\
= \neg ok \vee R & \text{[relational calculus]} \\
= ok \Rightarrow R & \square
\end{array}$$

H1 designs have a left-zero

$$
\begin{aligned}
& \textbf{true} \; ; \; R && \text{[assumption (R is \textbf{H1})]} \\
= \; & \textbf{true} \; ; \; (\, ok \Rightarrow R \,) && \text{[relational calculus]} \\
= \; & (\, \textbf{true} \; ; \; \neg\, ok \,) \lor (\, \textbf{true} \; ; \; R \,) && \text{[relational calculus]} \\
= \; & \textbf{true} \lor (\, \textbf{true} \; ; \; R \,) && \text{[relational calculus]} \\
= \; & \textbf{true} && \Box
\end{aligned}
$$

H1 designs have a left-unit

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

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

2.6.2 *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 $R^t = R[true/ok']$.

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

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

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

Designs are *H2*

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

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

2.6.3 *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, $x' = 2 \vdash true$ 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

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

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.

2.6.4 *H4*: feasibility

The final healthiness condition is also algebraic: $R \,;\, \textbf{true} = \textbf{true}$. 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 3

Linking Paradigms

3.1 Introduction

3.1.1 The COMPASS Modelling Language

Currently, *CML* contains several language paradigms.

1. **State-based description.** The theory of designs in UTP provides a nondeterministic programming language with pre- and postcondition specifications as contracts. The concrete realisation of this theory is the VDM language with its type system and structuring mechanisms.

2. **Concurrency and communication.** The theory of reactive processes in UTP provides a way of constructing networks of processes that communicate by passing messages. The concrete realisation is the CSP_M language with its rich collection of process combinators.

3. **Object orientation.** The theory of object orientation in UTP is built on the theory of designs and provides a way of structuring state-based descriptions through subtyping, inheritance, and dynamic binding. It has mechanisms for object creation, type testing, type casting, and state-component access.

4. **Pointers.** The theory of pointers in UTP provides a way of modelling heap storage and its manipulations, as found in implementations of object orientation, for which we have a reference semantics. Crucially, it supports modular reasoning about the heap.

5. **Time.** The theory of timed traces in UTP supports the observation of events in discrete time. It is used in a theory of Timed CSP [36].

3.1.2 Linking Paradigms

The semantic models mentioned in the last section are each formalised as sets of relations. For example, state-based descriptions are represented as pre- and postconditions, which are familiar from languages such as VDM and B. In UTP, an operation to

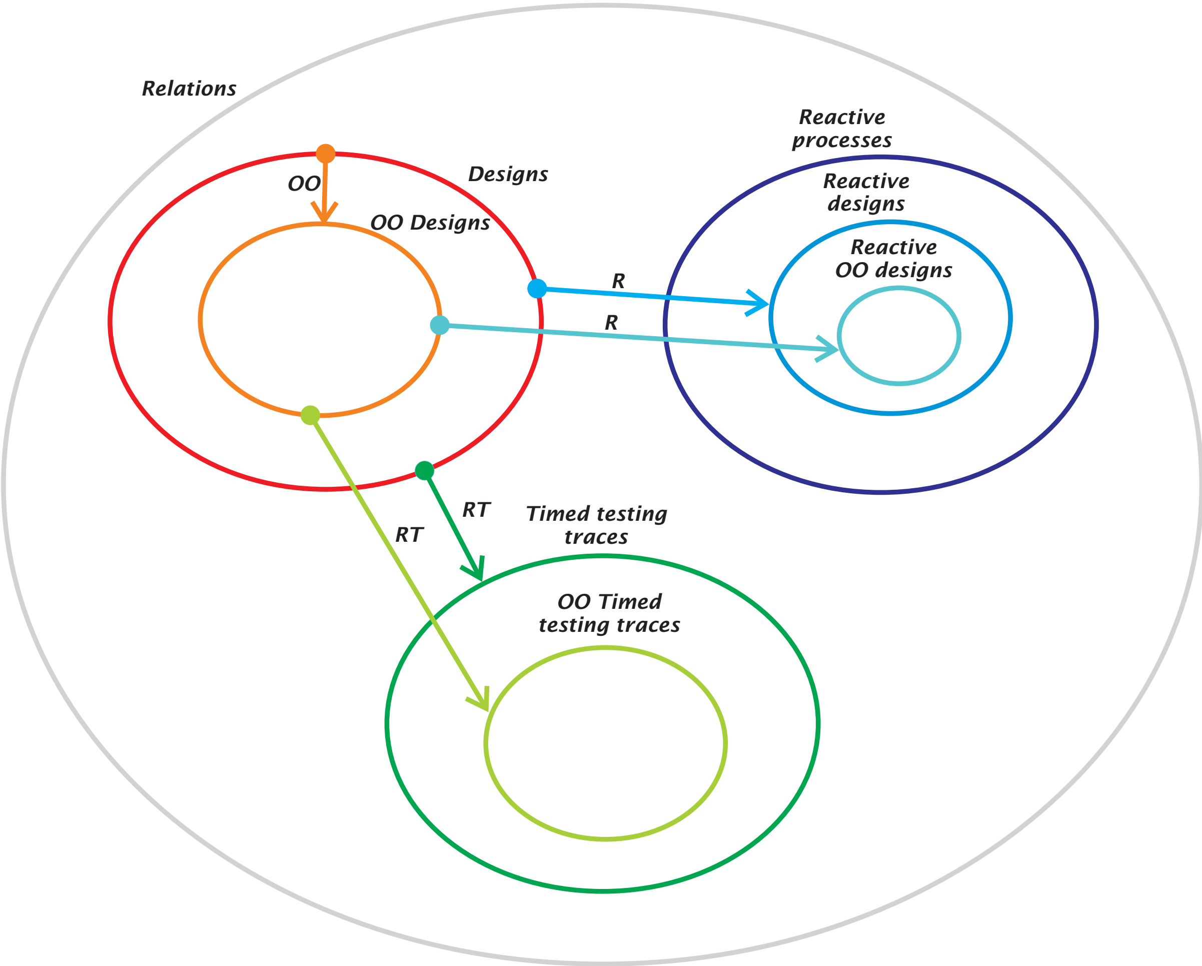

Figure 3.1: *CML* semantic models.

decrement a variable x, which must be invariantly non-negative, would be written as $(x > 0 \vdash x' = x - 1)$. The precondition requires that $x > 0$ and the postcondition ensures that the after-value of x, written as x', is exactly one less than the before-value of x, written $x - 1$. This pair of predicates is modelled as a single predicate (a relation) with two observational variables: $(ok \land x > 0 \Rightarrow ok' \land x' = x - 1)$. This is read as "if the operation is started (the observation ok is true) and $x > 0$, then the operation must terminate (the observation ok' is true) and when it does, $x' = x - 1$ must be true". Designs are organised into a lattice of relations ordered by refinement. At the bottom of the lattice is the aborting operation and at the top is the infeasible operation that can never be started. All other designs are somewhere in between. The process of correctness by construction starts by specifying the requirements for an operation as a design, moving upwards through the lattice in a series of refinement steps, until an implementation is reached. As usual, the specification is chosen to make the formalisation of requirements as easy and clear as possible, whilst the implementation is chosen to be executable on the chosen technology platform. Choices between alternative, correct refinements in this process are usually determined by non-functional requirements.

Mappings exist between the different semantic lattices, and some of these are shown in Figure 3.1. These mappings can be used to translate a model in one lattice into a corresponding model in another lattice. For example, the lattice of designs is completely disjoint from the lattice of reactive processes, but the mapping $\mathbf{R}$ maps every design into a corresponding reactive process. Intuitively, the mapping equips the design with the crucial properties of a reactive process: that it has a trace variable that records the history of interactions with its environment and that it can wait for such interactions. A vital healthiness condition is that this trace increases monotonically: this ensures that once an event has taken place it cannot be retracted—even when the process aborts.

But there is another mapping that can undo the effect of $\mathbf{R}$: it is called $\mathbf{H}$, and it is the function that characterises what it is to be a design. $\mathbf{H}$ puts requirements on the use of ok and ok', and it is the former that concerns us here. It states that, until the operation has started properly (ok is true), no observation can be made of the operation's behaviour. So, if the operation's predecessor has aborted, nothing can be said about any of the operation's variables, not even the trace observational variable. This destroys the requirement of $\mathbf{R}$ that says that the trace increases monotonically.

This pair of mappings form a structure known as a *Galois connection* [30]. Galois connections exist between all the semantic domains mentioned in the last section. One purpose of a Galois connection is to embed one theory within another, and this is what gives the compositional nature of UTP and ***CML***, since Galois connections compose to form another Galois connection. For example, if we establish a Galois connection between reactive processes and timed reactive processes (see Section 5.3), then we can compose the connection between designs and reactive processes with this new Galois connection to form a connection between designs and timed reactive processes.

As an example of a Galois connection, consider again the embedding of designs within the theory of reactive processes. Every relation, including designs, can be transformed into a reactive process using the reactive healthiness condition $\mathbf{R}$; similarly, every relation, including reactive processes, can be transformed into a design using the design healthiness condition $\mathbf{H}$. This pair of mappings will be shown to form a Galois connection in Section 5.2.

This apparently obscure mathematical fact, that designs and relations are related by a Galois connection, is of great practical value. One of the most important features of designs is assertional reasoning, including Hoare logic and Dijkstra's weakest precondition calculus. It has been reported that assertional methods are "one of the most useful automated techniques available for detecting faults and providing information about their locations" [10]. Assertional reasoning can be incorporated into the theory of reactive processes by means of $\mathbf{R}$. Consider the Hoare triple $p \{ Q \} r$, where p is a precondition, r is a postcondition, and Q is a reactive process. We can give this the following meaning: $(\mathbf{R}(p \vdash r') \sqsubseteq Q)$. This is a refinement assertion. The specification is $\mathbf{R}(p \vdash r')$; here the precondition p and the postcondition r have been assembled into a design (note that r becomes a condition on the after-state; this design is then translated into a reactive process by the mapping $\mathbf{R}$). This reactive specification must then be implemented correctly by the reactive process Q. In this way, reasoning with preconditions and postconditions can be extended from state-based operations to cover all operators of the reactive language, including non-terminating processes, concurrency, and communication.

This is the foundation of the contractual approach used in COMPASS: preconditions and postconditions (designs) can be embedded in each of the semantic domains and this brings uniformity through a familiar reasoning technique.

In summary, semantic heterogeneity in ***CML*** is reconciled by using UTP to include new semantic domains within the COMPASS framework. New domains are built as lattices of relations and equipped with Galois connections to compose and map semantic models.

3.2 Formal Links

In this section, we introduce the concept of a Galois connection formally and give three examples of their use in UTP and ***CML***.

3.2.1 Galois Connections

Our fundamental notion is that of a Galois connection on lattices [30]; actually, much of what we say applies equally to posets (partially ordered sets), but we are particularly interested in lattices.

Example 3.2.1 (Arithmetic) *Consider the following inequation:*

$$x + y \leq z, \ \textit{for } x, y, z : \mathbb{Z}$$

We can shunt the variable y to the other side of the inequation without changing the validity of the inequality: $x \leq z - y$. Writing $L(n) = n + y$ and $R(n) = n - y$, we can summarise this arithmetic law as

$$L(x) \leq z \quad \textbf{iff} \quad x \leq R(z)$$

This law is an example of a so-called shunting rule that is often useful in manipulating arithmetic expressions. □

This law is our first example of a Galois connection, a mathematical structure with the following definition.

Definition 3.2.1 (Galois connection) *A Galois connection between two lattices $(S, \sqsubseteq)$ and $(T, \overline{\sqsubseteq})$ is a pair of functions (L, R) with $L : S \to T$ (the left adjoint) and $R : T \to S$ (the right adjoint) satisfying, for all X in S and Y in T*

$$L(X) \sqsupseteq Y \quad \textbf{iff} \quad X \mathrel{\overline{\sqsupseteq}} R(Y)$$

In much of what follows, the lattices share the same order. □

We depict a Galois connection as a diagram. Suppose that S is a lattice with order relation $\sqsupseteq$ and T is a lattice with order $\overline{\sqsupseteq}$, and $L : S \to T$ and $R : T \to S$. Suppose further that (L, R) constitutes a Galois connection, then we denote this by the diagram

$$(S, \sqsupseteq) \underset{R}{\overset{L}{\rightleftarrows}} (T, \overline{\sqsupseteq})$$

Example 3.2.2 (Cartesian Relations and Kleisli Functions) *Relations can be modelled in at least two distinct ways. First, they can be modelled as sets of pairs; each pair represents an element and its relative. For example, the COMPASS consortium relation has_members relates countries and consortium members; it is represented by the set of pairs*

$$\begin{aligned} & has\_members_C \\ & \quad = \{ UK \mapsto Atego, UK \mapsto Newcastle, UK \mapsto York, \\ & \qquad Denmark \mapsto Aarhus, Denmark \mapsto B\&O, \\ & \qquad Germany \mapsto Bremen, \\ & \qquad Italy \mapsto Insiel, \\ & \qquad Brazil \mapsto UFPE \} \end{aligned}$$

(Here, we write $UK \mapsto Atego$ *(pronounced "UK maps to Atego") for the pair* $(UK, Atego)$*.) This model is known as the Cartesian relation between sets* A *and* B *and is of type* $\mathbb{P}(A \times B)$*.*

Another way to represent this relation is as a function from an element to the set of all its relatives:

$$\begin{aligned} & has\_members_K \\ & \quad = \{ UK \mapsto \{Atego, Newcastle, York\}, \\ & \qquad Denmark \mapsto \{Aarhus, B\&O\}, \\ & \qquad Germany \mapsto \{Bremen\}, \\ & \qquad Italy \mapsto \{Insiel\}, \\ & \qquad Brazil \mapsto \{UFPE\} \} \end{aligned}$$

This model is known as the Kleisli relation between two sets A *and* B *and is of type* $A \to \mathbb{P}\, B$*.*

Cartesian relations can be arranged in a lattice ordered by subset inclusion: $(\mathbb{P}(A \times B), \subseteq)$*. For simplicity, assume that these relations are all total; that is, for relation* $R : \mathbb{P}(A \times B)$*,* $\operatorname{dom} R = A$*.*

Kleisli functions form a lattice $(A \to \mathbb{P}\, B, \sqsupseteq)$*, where the order is defined as*

$$K_1 \sqsupseteq K_2 = (\forall\, x : A \bullet K_1(x) \subseteq K_2(x))$$

Assume that the functions are total and that their ranges contain non-empty sets.

Now, there is a Galois connection $(\mathbb{P}(A \times B), \subseteq) \underset{R}{\overset{L}{\rightleftarrows}} (A \to \mathbb{P}\, B, \sqsupseteq)$*, where*

$$L(C) = (\,\lambda\, a \bullet \{\, b \mid (a, b) \in C \,\}\,)$$

$$R(K) = \{\, (a, b) \mid b \in K(a) \,\}$$

Proof 1

$$\begin{aligned} & L(C) \sqsupseteq K \\ & = \{\ L\ \} \end{aligned}$$

$$
\begin{array}{l}
(\lambda\, a \bullet \{\, b \mid (a, b) \in C \,\}) \sqsupseteq K \\
= \{ \sqsupseteq \} \\
\forall\, x : A \bullet (\lambda\, a \bullet \{\, b \mid (a, b) \in C \,\})(x) \subseteq K(x) \\
= \{ \beta\text{-reduction} \} \\
\forall\, x : A \bullet \{\, b \mid (x, b) \in C \,\} \subseteq K(x)) \\
= \{ \text{subset} \} \\
\forall\, x : A;\ y : B \bullet y \in \{\, b \mid (x, b) \in C \,\} \Rightarrow y \in K(x) \\
= \{ \text{comprehension} \} \\
\forall\, x : A;\ y : B \bullet (x, y) \in C \Rightarrow y \in K(x) \\
= \{ \text{comprehension} \} \\
\forall\, x : A;\ y : B \bullet (x, y) \in C \Rightarrow (x, y) \in \{\, (a, b) \mid b \in K(a) \,\} \\
= \{ \text{subset} \} \\
C \subseteq \{\, (a, b) \mid b \in K(a) \,\} \\
= \{ R \} \\
C \subseteq R(K)
\end{array}
$$

□

There is another, alternative definition of a Galois connection, where we consider

- $L(X)$ as the strongest element Y with $X \sqsupseteq R(Y)$
- $R(Y)$ as the weakest element X with $L(X) \sqsupseteq Y$

providing that L and R are monotonic. We formalise this in the following law.

Law 3.2.1 (Alternative Galois Connection)

$$
\begin{array}{l}
(L, R) \textit{ is a Galois connection between lattices } S \textit{ and } T \\
\quad \textbf{iff} \left\{ \begin{array}{ll}
\textbf{Prop. 3.2.1.1} & L, R \text{ monotonic} \\
\textbf{Prop. 3.2.1.2} & L \circ R \sqsupseteq \mathrm{id}_T \\
\textbf{Prop. 3.2.1.3} & \mathrm{id}_S \sqsupseteq R \circ L
\end{array} \right.
\end{array}
$$

The function $L \circ R$ is strengthening and the function $R \circ L$ is weakening. □

A useful property of a Galois connection is that L is a pseudo-inverse of R and vice versa.

Law 3.2.2 (Pseudo-inverse) *For any Galois connection (L,R), each function is a pseudo-inverse of the other:*

$$
\begin{array}{ll}
\textbf{Law 3.2.2.1} & L = L \circ R \circ L \\
\textbf{Law 3.2.2.2} & R = R \circ L \circ R
\end{array}
$$

□

As interesting specialisation of a Galois connection is when the function L is surjective; that is, when $\operatorname{ran} L = T$, where T is the set of elements in the right-hand lattice. As we see below in Law 3.2.3, L's surjectivity is equivalent to R's injectivity, which in turn is equivalent to the existence of a left inverse for R, which turns out to be L itself.

This special case is known as a *retract* (L is a retraction of R); elsewhere, it is known variously as a Galois injection or a Galois insertion. If it is R that is surjective, then L will be injective and R will be its left inverse; this special case is known as a *coretract*. If both functions are surjective, then they are also both injective and this very special case is known as a Galois bijection. Such structures are still of practical interest.

Example 3.2.3 (Logarithms) *The Galois connection* $(\ln(m) = n) = (m = e^n)$ *relates natural logarithms and natural exponents. The Galois connection* $(\ln, (\lambda\, n \bullet e^n))$ *is bijective, but logarithms are still practically useful as as a means to simplify calculations because of the fact that the logarithm of a product is the sum of the logarithms of the factors.*

Definition 3.2.2 (Retract and Coretract) *For any Galois connection* (L, R)*:*

Def 3.2.2.1	(L, R)	*is a retract*	*if* $L \circ R = \mathrm{id}_T$	(Galois insertion)
Def 3.2.2.2	(L, R)	*is a coretract*	*if* $R \circ L = \mathrm{id}_S$	(Galois injection)

□

We are nearly ready to give a collection of useful equivalences about retracts and coretracts, but first we need one more definition. Recall that if F is monotonic, then $[\,(P \sqsubseteq Q) \Rightarrow (F(P) \sqsubseteq F(Q))\,]$. If the implication also holds in the opposite direction, then F is an order similarity.

Definition 3.2.3 (Order Similarity) $F : S \rightarrow S$ *is an order similarity if, for every* $P, Q : S$*:*

$$(F(P) \sqsubseteq F(Q)) = (P \sqsubseteq Q)$$

□

Another term for a function being monotonic is that it is order preserving; another term for the converse is that the function is order reflecting; the pair of implications is then termed an order embedding or an order monomorphism.

This now gives us three equivalent ways of characterising a retract.

Law 3.2.3 (Retract Property)

(L, R) *is a retract*
iff (**Law 3.2.3.1**) L *is surjective*
iff (**Law 3.2.3.2**) R *is injective*
iff (**Law 3.2.3.3**) R *is an order similarity*

□

Similarly, there are four equivalent ways of characterising a coretract.

Law 3.2.4 (Coretract Property)

(R, L) *is a coretract*
iff (**Law 3.2.4.1**) R *is surjective*
iff (**Law 3.2.4.2**) L *is injective*
iff (**Law 3.2.4.3**) L *is an order similarity*

□

There are four more useful properties of Galois connections between complete lattices. The first two tell us that it is necessary to have only one of the two functions, since the other can be determined uniquely. The second two properties are about distribution through the lattice operators: L preserves least upper-bounds (L is a complete join-morphism); and R preserves greatest lower-bounds (R is a complete meet-morphism).

Law 3.2.5 (Galois Connection Properties) *For any Galois connection* (L, R) *on complete lattices* S *and* T*, we have:*

Law 3.2.5.1	R uniquely determines L	$L(P) = \bigsqcap \{ Q \in S \mid P \sqsubseteq R(Q) \}$
Law 3.2.5.2	L uniquely determines R	$R(Q) = \bigsqcup \{ P \in T \mid L(P) \sqsubseteq Q \}$
Law 3.2.5.3	L preserves least upper-bounds	$L(\bigsqcup X) = \bigsqcup \{ L(P) \mid P \in X \}$
Law 3.2.5.4	R preserves greatest lower-bounds	$R(\bigsqcap Y) = \bigsqcap \{ R(Q) \mid Q \in Y \}$

□

The last two properties in Law 3.2.5 are interesting because they link the lattice operators involved in a Galois connection. In UTP a theory typically consists of a set of predicates over a particular alphabet ordered in a lattice. The lattice is accompanied by a signature that describes the operators of the theory. There may be other similar operators in the signatures of the two theories involved in the Galois connection, and the links between them can be investigated as morphisms in a similar way to those for the lattice operators. For example, in the Galois connection between designs and reactive processes, each theory has an imperative assignment, and we would expect that them to be related so that $(x :=_{\boldsymbol{R}} y) = \boldsymbol{R}(x :=_{\boldsymbol{H}} y)$.

The following definition describes the links that might be made by L between the function symbol F in the two lattice signatures and by a set of such function symbols.

Definition 3.2.4 (Σ-morphism)

L is an F-morphism	$L \circ F_S = F_T \circ L$
L is an $F_{\sqsubseteq}$-morphism	$L \circ F_S \sqsubseteq F_T \circ L$
L is an $F_{\sqsupseteq}$-morphism	$L \circ F_S \sqsupseteq F_T \circ L$
L is a Σ-morphism	*L is an F-morphism, for all F in Σ*

□

If the Galois connection is a retract, then there is a very precise relationship between F in the two lattices and L.

Law 3.2.6 (Retract Morphism) *If* (L, R) *is a retract and* L *is an* F*-morphism, then*

$$F_S = R \circ F_T \circ L$$

Proof 2 *Assume L is an F-morphism :*

$$L \circ F_S = F_T \circ L$$
$$= \{ \textit{identity} \}$$
$$L \circ F_S = \mathrm{id}_T \circ F_T \circ L$$

$= \{$ assumption: (L, R) is a retract, Def 3.2.2.1 $\}$

$\{$ Retract and Coretract $(L \circ R = \mathrm{id}_T)$ $\}$

$L \circ F_S = L \circ R \circ F_T \circ L$

$= \{$ assumption: (L, R) is a retract, $\}$

$\{$ Law 3.2.3.3 Retract Property $(R$ is an order similarity$)$ $\}$

$F_S = R \circ F_T \circ L$

□

A dual property exists for a coretract.

Law 3.2.7 (Coretract Morphism) *If (L, R) is a coretract and R is an F-morphism, then*

$$F_S = R \circ F_T \circ L$$

Proof 3 *Assume R is an F-morphism*

$R \circ F_T = F_S \circ R$

$= \{$ identity $\}$

$R \circ F_T = \mathrm{id}_S \circ F_S \circ R$

$= \{$ assumption: (L, R) is a coretract, $\}$

$\{$ Def 3.2.2.2 Retract and Coretract $(R \circ L = \mathrm{id}_S)$ $\}$

$R \circ F_T = R \circ L \circ F_S \circ R$

$= \{$ assumption: (L, R) is a coretract, $\}$

$\{$ Law 3.2.4.3 Coretract Property $(L$ is an order similarity $)$ $\}$

$F_T = L \circ F_S \circ R$

□

We can use these morphisms to calculate a function in one lattice in terms of another. For example, suppose that L is an F morphism, then we can calculate the strongest definition for F_T in terms of F_S and the functions L and R. This is described in the following lemma.

Lemma 3.2.1 (Strongest Solution) *The strongest solution for F_T in*

$$F_S(X) \sqsupseteq R \circ F_T \circ L(X)$$

is denoted by $F^{\#}$, where

$$F^{\#}(Y) = L \circ F_S \circ R(Y)$$

□

This concludes our brief description of Galois connections and their properties. A more detailed description of can be found in [30]. In Chapter 5, we give three examples linking some of the semantic domains of *CML*.

Chapter 4

UTP Semantics for ***CML***

The behavioural part of ***CML*** is given a semantics closely related to Lowe & Ouaknine's Timed Testing Traces [20], and this in turn is related to the standard semantics for CSP. The fundamental notions of these semantic models for CSP are those of events, traces and refusals. The relationship between the semantics we develop and the Timed Testing Traces semantics given by Lowe & Ouaknine [20] is considered in more detail at the end of the chapter. We begin with an informal introduction to these fundamental notions.

An *event* is an atomic and instantaneous interaction between a ***CML*** 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.

An observation of a ***CML*** process is a *timed trace*. Consider first untimed traces, as understood in the context of CSP. These are sequences of events recorded by an observer. In CSP a 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 fragment of ***CML:*** $a \to b \to STOP$. 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*.

4.1 Timed Testing Traces

Our semantic domain consists of traces with embedded refusal sets. It is close to the Lowe and Ouaknine timed testing model [20], which records the passing of time with an explicit *tock* event and allows refusal experiments to be made only before *tock*s. We do not observe the *tock* event directly and so *tock* is not an observable event. Instead, we observe the passage of time through the refusal experiments. At the end of each time interval either a refusal experiment is made or the empty refusal set is recorded.

If we let Σ be the universe of events, then the traces that we may observe are drawn from the following set:

Definition 4.1.1

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

This definition uses a variation on the standard notation for regular expression where $+$ is to be understood as a choice, and x^* is the expression that describes all the finite sequences containing only x. All members of this set are potential traces of ***CML*** processes. We do not use untimed traces in the remainder of our report, and so may refer to members of this set as *traces* from now on.[1] For example, the trace

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

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.
- At the end of the second time interval, no events were refused.
- The third time interval is incomplete, but the trace $\langle c\rangle$ was observed so far.

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

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

[1]This definition of timed traces has changed from previous deliverables. *tock* events are no longer explicitly included; the passing of time is observed through the presence of refusal sets in the trace. Definition 4.1.5 allows us to replace these refusal sets with explicit *tock* events if required.

A trace is given a mathematical interpretation as a function from an initial segment of the natural numbers. For example, the trace above has the representation

$$\{1 \mapsto \{a, b\}, 2 \mapsto \{b\}, 3 \mapsto \{a, b\}, 4 \mapsto \{b\}, 5 \mapsto \{a, b\}\}$$

Typically, elements of the range of the function are the same type, but the range of a timed testing trace may contain both events (communications) and sets of events (refusals).

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 *squash* is taken from Z [35]). 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, $\langle a, b, b, c, d\rangle \upharpoonright \{b, d\} = \langle b, b, d\rangle$.

Definition 4.1.2 (Squash and Filter)

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

We can use the definitions above to define functions to extract information from a timed trace. The function $events(t)$ throws away the refusal sets in t, leaving just the trace of events. The function $refsduring(t)$ collects together the set of refusal sets in t, throwing away ordering information and the event component. The function $refusals(t)$ calculates all the events that are refused at some point during the trace t.

Definition 4.1.3 *Let $A \subseteq \Sigma$, $a \in \Sigma$ and $t \in timedTrace$. Then*

$$\begin{aligned}
&events(t) = t \upharpoonright \Sigma\\
&refsduring(t) = \operatorname{ran}(t \rhd \mathbb{P}(\Sigma))\\
&refusals(t) = \bigcup refsduring(t)
\end{aligned}$$

The idle prefix of a trace t is denoted $idleprefix(t)$ and describes the longest prefix of t containing no observable events. For example, the trace

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

has the idle prefix $\langle \emptyset, \{a\}\rangle$. The idle suffix of t is the remainder of the trace after the first visible event has been removed. In the example above b is removed, and the idle suffix is $\langle c, \{a, c\}\rangle$.

Definition 4.1.4 (idleprefix and idlesuffix) *Let $A \subseteq \Sigma$, $a \in \Sigma$ and $t \in timedTrace$. Then*

$$\begin{aligned}
&idleprefix(\langle\rangle) = \langle\rangle\\
&idleprefix(\langle A\rangle \frown t) = \langle A\rangle \frown idleprefix(t)\\
&idleprefix(\langle a\rangle \frown t) = \langle\rangle
\end{aligned}$$

and

$$\begin{aligned}
&idlesuffix(\langle\rangle) = \langle\rangle\\
&idlesuffix(\langle A\rangle \frown t) = idlesuffix(t)\\
&idlesuffix(\langle a\rangle \frown t) = t
\end{aligned}$$

We also define a function $tocks(t)$ that replaces all the refusal sets in a timed trace t with a new event, $tock \notin \Sigma$. This proves useful for placing conditions on the time taken by a trace.

Definition 4.1.5 *Let $A \subseteq \Sigma$, $a \in \Sigma$ and $t \in timedTrace$. Then*

$$\begin{aligned}
&tocks(\langle\rangle) = \langle\rangle\\
&tocks(\langle a\rangle \frown t) = \langle a\rangle \frown tocks(t)\\
&tocks(\langle A\rangle \frown t) = \langle tock\rangle \frown tocks(t)
\end{aligned}$$

The trace precedence relation $t \preceq u$ holds when t contains less information than u, either because t is a prefix of u, or the refusal sets in t are subsets of the corresponding refusal sets in u, or a combination of the two conditions.

Definition 4.1.6 (Testing trace precedence) *Let $a \in \Sigma$, $X \subseteq Y \subseteq \Sigma$ and $t, u \in timedTrace$. Then*

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

For example $\langle a, \{b\}, c, \{d, e\}\rangle \preceq \langle a, \{b, d\}, c, \{d, e\}\rangle$. This is a stronger relation than the usual prefix relation on *timedTraces*: $\_ \leq \_$:

Lemma 4.1.1 (Precedence traces)

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

Proof *by induction on t.*

A similar result holds for the refusals over testing traces:

Lemma 4.1.2 (Precedence refusals)

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

Proof *by induction on t.*

We will introduce other operators on timed traces as we need them.

4.1.1 The *CML* language

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

- Successful termination: $SKIP$ (Section 4.1.2).
- Deadlocked process: $STOP$ (Section 4.1.3).
- Assignment: $(v := e)$ (Section 4.1.4).
- Prefixed termination: $a \to SKIP$ (Section 4.1.5).
- Divergence: $CHAOS$ (Section 4.1.6).
- Miracle: $MIRACLE$ (Section 4.1.7).
- Specification statement: $w : [\, pre, post \,]$ (Section 4.1.8).
- Sequential composition: $P \ ; \ Q$ (Section 4.1.9).
- Prefixed action: $a \to P$ (Section 4.1.10).
- Internal choice: $P \sqcap Q$ (Section 4.1.11).
- External choice: $P \Box Q$ (Section 4.1.12).
- Parallel composition: $P \parallel_{cs} Q$ (Section 4.1.13).
- Interleaving parallel: $P \interleave Q$ (Section 4.1.14).
- Abstraction: $P \setminus A$ (Section 4.1.15).
- Recursion: $\mu X \bullet P(X)$ (Section 4.1.16).
- Timeout: $P \overset{n}{\rhd} Q$ (Section 4.1.17).
- Untimed timeout: $P \rhd Q$ (Section 4.1.18).
- Wait: $Wait(n)$ (Section 4.1.19).
- Interrupt: $P \bigtriangleup Q$ (Section 4.1.20).
- Timed interrupt: $P \overset{n}{\bigtriangleup} Q$ (Section 4.1.21).
- Startsby: $P\ startsby(n)$ (Section 4.1.22).
- Endsby: $P\ endsby(n)$ (Section 4.1.23).
- While: $b * P$ (Section 4.1.24).
- Guarded actions: $[g] \,\&\, P$ (Section 4.1.25).

4.1.2 Observation Variables and Healthiness Conditions

Observations of *CML* processes contain four pairs of variables.

- ok, ok': These are the observation variables from designs [16, 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 [16, Chapter 8]. 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.
- rt, rt': These are the observations of the trace of the previous process (rt) and the current process (rt'). 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.
- v, v': These are the variables that record our observations of the initial and final state of the current process.

We also introduce a derived variable: tt' is equal to $rt' - rt$ whenever that expression is defined, and undefined otherwise. Intuitively, tt' represents the portion of the trace carried out by the currently active process. However, it is not an observation variable and is therefore not quantified by [ ] (universal quantification over alphabets) or by sequential composition — it can always be replaced by $rt' - rt$ in any expression.

We will make use of the notational shorthand introduced in [9]:

Definition 4.1.7

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

which denotes the substitution of the boolean variables b and c for the variables ok' and $wait$.

There are five healthiness conditions. We note in passing that we do not need to restrict the structure of the trace variables with a healthiness condition, since all elements of the set $timedTrace$ are structurally valid observations of timed reactive processes.

The first requirement is that tt' is well-defined. This requires that the observation of rt prefixes the observation of rt'. ***RT1*** is similar to ***R1*** in that it ensures that a process cannot alter the part of the trace that has already been observed; all it may do is append to rt.

Definition 4.1.8 (*RT1*)

$$\textbf{\textit{RT1}}(P) = P \wedge rt \leq rt'$$

Our next healthiness condition is similar to ***R2*** in Hoare & He's theory of reactive processes (see [16, 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 on a particular amount of time having elapsed under its predecessor's control.

Definition 4.1.9 (*RT2*)

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

The healthiness condition ***RT3*** is a modified form of ***R3*** in the theory of reactive processes (see [16, p.196]). It is similar to the condition ***R3h*** proposed in [8], and describes the behaviour of a process that has not been started: it may not extend the trace ($tt' = \langle\rangle$), and it may not observe the internal state of its predecessor. ***R3h*** in [8] differs from ***R3*** by removing the insistence that the state does not change while the process is waiting for external interaction. Changes to the internal state of a process are permitted by ***RT3***, but should remain unobservable until some interaction takes place. This inability to observe internal interaction has the consequence that a choice between two processes cannot be resolved by internal state changes, but only external events or the termination of one of the processes.

Definition 4.1.10 (*RT3*)

$$\textbf{\textit{RT3}}(P) = \exists\, v' \bullet I\!I \lhd wait \rhd P$$

Our fourth healthiness condition corresponds to ***CSP1*** in Hoare & He's theory of CSP (see [16, 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 cannot alter the behaviour that has already been observed (***RT1***), but otherwise it can behave arbitrarily.

Definition 4.1.11 (*RT4*)

$$\textbf{\textit{RT4}}(P) = \textbf{\textit{RT1}}(\neg\, ok) \lor P$$

Our fifth healthiness condition is analogous to ***CSP2*** in [16], and states that P must be monotonic in the value of the ok' variable, just like a design: P cannot demand instability and nontermination. In other words, it is always possible to terminate: if P is capable of reaching a state with $\neg\, ok'$, then it must be capable of reaching the same state with ok'. In [9] in the context of designs, the authors show that this healthiness property is equivalent to $[P^f \Rightarrow P^t]$.

Definition 4.1.12 (*RT5*)

$$\textbf{\textit{RT5}}(P) = P \,;\, (I\!I_{\{rt, wait, v\}} \land (ok \Rightarrow ok'))$$

We subscript the relational identity $I\!I$ with the set of observation variables that are required to be constant. Notice that ***RT4*** and ***RT5*** are the timed reactive versions of ***H1*** and ***H2***, respectively (in the same way that ***CSP1*** and ***CSP2*** are the reactive versions).

The remaining healthiness conditions involve the process *SKIP*. *SKIP* has precondition true, and it must terminate before any time passes. It is impossible for *SKIP* to engage in any events before it terminates. When it terminates, it does not change the state, although it can conceal its intermediate state before terminating.

Definition 4.1.13 (*SKIP*)

$$SKIP = \textbf{\textit{RT3}} \circ \textbf{\textit{RT4}}(ok' \land tt' = \langle\rangle \land (\neg\, wait' \Rightarrow v' = v))$$

The sixth healthiness condition corresponds to ***CSP3***, insisting that *SKIP* is a left unit of sequential composition. In *CSP*, the effect of this was to make processes independent of the refusals of their predecessor, but since refusal information is captured in the trace in *CML*, this is already guaranteed by ***RT2***. Instead, ***RT6*** ensures that processes never engage in urgent initial actions: it is always possible to make a stable initial observation of any process.

Definition 4.1.14 (*RT6*)

$$\mathbf{RT6}(P) = SKIP \; ; \; P$$

The seventh healthiness condition corresponds to ***CSP4***, insisting that $SKIP$ is a right unit of sequential composition. Any process which has an ***RT3*** process as a right unit, such as $SKIP$, will conceal the value of its intermediate states when it hasn't terminated. Any process with has an ***RT4*** process as a right unit can engage in arbitrary behaviour after it has diverged, and so its precondition must be prefix closed. Additionally, any process which has $SKIP$ as a right unit cannot terminate urgently: if it is possible to observe the process terminated after any behaviour, it was also possible to observe the process with the same trace before it had terminated.

Definition 4.1.15 (*RT7*)

$$\mathbf{RT7}(P) = P \; ; \; SKIP$$

Lemma 4.1.3 (*RT* functions are commuting monotonic idempotents)

1. ***RT1***–***RT7*** *are all monotonic idempotents.*
2. ***RT1***–***RT7*** *all commute.*

Definition 4.1.16 (*RT*)

$$\mathbf{RT} \mathrel{\widehat=} \mathbf{RT1} \circ \mathbf{RT2} \circ \mathbf{RT3} \circ \mathbf{RT4} \circ \mathbf{RT5} \circ \mathbf{RT6} \circ \mathbf{RT7}$$

We can now proceed to define our process combinations. We define processes as timed reactive designs in the style of *Circus* (for an introduction to this style, see [9]).

In the definitions that follow, we will always make the assumption that any constituent processes in a process definition are themselves ***RT***-healthy.

4.1.3 Deadlock

The inactive language construct is the deadlocked process: $STOP$. This process is an ***RT***-healthy design with precondition true that never engages in any events and is perpetually waiting ($wait'$). In the postcondition, we also have that $events(tt') = \langle\rangle$: no events are ever observed. $STOP$ deadlocks events but it cannot deadlock the clock, so refusal experiments can happen freely and no further trace restriction is required. $STOP$ also says nothing about the final value of the program variables v', which are left unconstrained.

Definition 4.1.17 (Deadlock)

$$STOP = \mathbf{RT}(\mathbf{true} \vdash events(tt') = \langle\rangle \wedge wait')$$

4.1.4 Assignment

For the assignment $v := e$, we make the simplifying assumption that the expression e is well defined.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 ($\neg\, wait'$) without any events ($tt' = \langle\rangle$), but having completed the assignment ($v' = e$). Elements of state other than v are unaffected. This design is then made healthy with the application of the healthiness conditions.

Notice that since it is ***RT7***, it will be possible to observe an assignment before it has terminated or for it to be preempted by other processes. Assignment can be used to represent *SKIP* as a reactive design, since $(v := v) = SKIP$.

Definition 4.1.18 (Assignment)

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

4.1.5 Prefixed termination

Prefixed termination is the process that is willing to perform a single, given event, and having done it terminates immediately. It has precondition *true*, and a postcondition that has two parts. Either the process is still waiting to engage in its event (a, say), in which case no events will occur and a will not be refused. Alternatively, the a has occured, in which case it was the only event, and the process terminated immediately with the state unchanged. Prefixed termination is used together with relational composition in Section 4.1.10 to define the general prefix process $a \rightarrow P$.

Definition 4.1.19 ($a \rightarrow SKIP$)

$$a \rightarrow SKIP = \mathbf{RT}\left(\mathbf{true} \vdash \begin{pmatrix} a \notin refusals(tt') \wedge \\ events(tt') = \langle\rangle \\ \lhd\, wait' \,\rhd \\ tt' = idleprefix(tt') \frown \langle a \rangle \wedge v' = v \end{pmatrix}\right)$$

4.1.6 Divergence

CHAOS is the least predictable process that satisfies the healthiness conditions. The precondition of *CHAOS* never holds, so its behaviour is always divergent.

Definition 4.1.20 (*CHAOS*)

$$CHAOS = \mathbf{RT}(\mathbf{false} \vdash \mathbf{true})$$

4.1.7 Miracle

MIRACLE is the infeasible process – its postcondition can never be established.

Definition 4.1.21 (*MIRACLE*)

$$MIRACLE = \mathbf{RT}(\mathbf{true} \vdash \mathbf{false})$$

4.1.8 Specification statement

CML inherits a specification statement from VDM as a way of describing operations and functions. If feasible, this may be refined into a *CML* action. If the post condition holds, the specification statement will terminate immediately and successfully. w is the framed variables: only these may be changed by the postcondition.

Definition 4.1.22 (Specification statement)

$$w : [\mathbf{pre}\, P\, \mathbf{post}\, Q] = \mathbf{RT}(P \vdash Q \land \neg\, wait' \land tt' = tt \land (v' \setminus w) = (v \setminus w))$$

4.1.9 Sequential Composition

Sequential composition of two reactive processes (written P $;_{RT}$ Q) is simply relational composition, given our healthiness conditions. It is closed under the healthiness conditions. We will dispense with the subscript on this operator after the definition.

Definition 4.1.23 (Sequential composition)

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

The sequential composition of two designs has two parts to its precondition: there must be no possible values for v' and $wait'$ that would have allowed process P to diverge at any point along its trace $(\neg\, (P^f_f \;;\; \mathbf{RT1}(\mathbf{true})))$ and the successful termination of P must not lead to a point where Q diverges $(\neg\, (P^t_f[false/wait'] \;;\; \mathbf{RT1}(Q^f_f)))$. Given the conjunction of these two conditions, P can be expected to terminate and lead to a stable observation of Q.

Lemma 4.1.4 (Precondition/Postcondition form)

$$P \;;\; Q = \mathbf{RT}(\neg\, (P^f_f \;;\; \mathbf{RT1}(\mathbf{true})) \land \neg\, (P^t_f[false/wait'] \;;\; \mathbf{RT1}(Q^f_f)) \vdash P^t_f \;;\; Q^{tt})$$

The abbreviation Q^{cd} is used to stand for $Q[c, d/ok, ok']$.

4.1.10 Prefix

The prefixed processes $a \rightarrow P$ is determined to engage in the event a and nothing else; after engaging in a it behaves like P. It is defined as a derived operator using prefixed termination (Section 4.1.5) and sequential composition (Section 4.1.9).

Definition 4.1.24 (Prefixing)

$$a \rightarrow P = a \rightarrow SKIP \;;\; P$$

As a design, this operator is defined as follows: The only possibility of divergence of the process $a \rightarrow P$ is if the process P diverges, and this is possible only if the event a is the initial visible event of the trace: $\langle a \rangle \leq events(tt')$. In the period before this observation, a cannot be refused: $a \notin refusals(idleprefix(tt'))$. Provided that the trace subsequent to this does *not* cause P to diverge $(\neg\ P^f_f[\mathit{idlesuffix}(tt')/tt'])$ then the postcondition will hold.

The postcondition describes the two possible states of $a \rightarrow P$. Either no events have been observed ($\mathit{events}(tt') = \langle\rangle$), in which case the process is waiting for input. The event a must not have been refused: $a \notin \mathit{refusals}(\mathit{idleprefix}(tt'))$, which is equivalent to $a \notin \mathit{refusals}(tt')$ because no event has been observed. The process cannot diverge in this case. Alternatively, an event has been observed and it must have been the a-event: the first event must be a. It is still the case that the event a must not be refused before it occurred, and after it occurs the process will continue as P. The future behaviour of the process is given by $P^t_f[\mathit{idlesuffix}(tt')/tt']$.

Lemma 4.1.5 (Precondition/Postcondition form)

$$
\begin{array}{l}
a \rightarrow P = \\
\quad \textbf{\textsf{RT}} \left(\begin{array}{l}
(\langle a \rangle \leq \mathit{events}(tt') \land a \notin \mathit{refusals}(\mathit{idleprefix}(tt'))) \Rightarrow \\
\qquad \neg\ P^f_f[\mathit{idlesuffix}(tt')/tt'] \\
\vdash \\
(a \notin \mathit{refusals}(\mathit{idleprefix}(tt')) \land \\
\qquad \left(\begin{array}{l}
\mathit{events}(tt') = \langle\rangle \\
\qquad \lhd\ \mathit{wait}'\ \rhd \\
\qquad\qquad \langle a \rangle \leq \mathit{events}(tt') \land P^t_f[\mathit{idlesuffix}(tt')/tt']
\end{array}\right)
\end{array}\right)
\end{array}
$$

4.1.11 Internal Choice

Internal choice is simply disjunction, as usual.

Definition 4.1.25 (Internal choice)

$$P \sqcap Q = P \lor Q$$

An internal choice can diverge if either of its component processes diverges, and guarantees termination only if both guarantee termination.

Lemma 4.1.6 (Precondition/Postcondition form)

$$P \sqcap Q = \textbf{\textsf{RT}}(\neg\ P^f_f \land \neg\ Q^f_f \vdash P^t_f \lor Q^t_f)$$

4.1.12 External Choice

An external choice can be resolved if the process diverges, engages in a visible event or terminates. Either the observed behaviour is unresolved and does not diverge, in which case processes must agree on the observed behaviour, or something observable happens, the choice is resolved and the process which was responsible for observable event is now responsible for the subsequent observation.

However, in order to guarantee prefix closure of the observed behaviour, it must still be the case after the choice has resolved that the initial behaviours that didn't resolve the choice were compatible with both branches. The relevant behaviours are the prefixes of the trace which contain no events and on which the precondition of both branches still holds.

Definition 4.1.26 (External choice)

$$P \Box Q \;\widehat{=}\; \left(\begin{array}{l} (P \wedge Q \wedge events(tt') = \langle\rangle \wedge wait' \wedge ok' \\ \vee \\ (P \vee Q) \wedge \forall\, tt_0 \leq idleprefix(tt') \bullet \\ \qquad\qquad (\neg\,(P \wedge Q)^f \Rightarrow (P \wedge Q)^t)[tt_0, true/tt', wait'] \end{array} \right)$$

The design form is, of course, more involved than internal choice.

The process $P \Box Q$ can diverge in any way that either of its branches could diverge, provided that initial prefixes of that behaviour do not violate the other branch of the choice. To enforce this condition, we require that on any initial behaviour which does not resolve the choice and satisfies the preconditions of both processes, it must be possible to satisfy the postconditions of both processes:

$$\forall\, tt_0 \leq idleprefix(tt') \bullet ((\neg\, P^f_f \wedge \neg\, Q^f_f) \Rightarrow (P^t_f \wedge Q^t_f))[tt_0, true/tt', wait']$$

If this condition holds, the behaviour is feasible and the preconditions of both branches must be satisfied: $\neg\, P^f_f \wedge \neg\, Q^f_f$.

Provided the process does not diverge, its behaviour must be compatible with at least one of its branches: $P^t_f \vee Q^t_f$. Additionally, it must satisfy both branches on any initial behaviour that does not resolve the choice: $(P^t_f \wedge Q^t_f)[idleprefix(tt'), true/tt', wait']$. The choice has not yet resolved when $events(tt') = \langle\rangle \wedge wait'$ holds, in which case this condition will insist that both branches still hold: $P^t_f \wedge Q^t_f$.

Lemma 4.1.7 (Precondition/postcondition form)

$$P \Box Q = \boldsymbol{RT} \left(\begin{array}{l} \left(\begin{array}{l} \forall\, tt_0 \leq idleprefix(tt') \bullet \\ \qquad ((\neg\, P^f_f \wedge \neg\, Q^f_f) \Rightarrow (P^t_f \wedge Q^t_f))[tt_0, true/tt', wait'] \end{array} \right) \\ \qquad \Rightarrow (\neg\, P^f_f \wedge \neg\, Q^f_f) \\ \vdash \\ (P^t_f \vee Q^t_f) \wedge (P^t_f \wedge Q^t_f)[idleprefix(tt'), true/tt', wait'] \end{array} \right)$$

4.1.13 Parallel Composition

A parallel composition specifies the set of events that require synchronisation between two processes; outside this set events happen independently, without needing the participation of the other process. Parallel composition is then a form of restricted conjunction, where the behaviour of each process is seen as a projection of the overall trace.

We call two timed reactive designs *disjoint* if they share no programming variables; they are allowed, of course, to share the observational variables rt, $wait$, and ok. Parallel composition is restricted to disjoint processes. 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. 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 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.

To take care of tt', we define a semantic operator on traces that *merges* a pair of traces together to give the set of traces that can result if the pair of traces are observed in parallel. To define this, we start by defining an intersection operator for refusal sets that will tell us what the refusal set will be for the parallel composition. Suppose that P has a refusal set X, Q has a refusal set Y, and A is the synchronisation alphabet. Our intersection operator (written $X \cap_A Y$) has 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 P and Q.

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

Definition 4.1.27 (Refusal set intersection)

$$X \cap_A Y \mathrel{\widehat=} (X \cap A) \cup (Y \cap A) \cup (X \cap Y)$$

Now we are ready to define our semantic operator on timed testing traces. This is similar to the one defined in [32].

Definition 4.1.28 (Trace interleaving)

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

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

The process *Skip* in parallel with an active process P will suspend termination until P finishes. They will both then synchronise on termination.

The traces formed by merging a single pair of timed testing traces are maximal: none is a prefix of any other.

Lemma 4.1.8 (Maximality of trace composition)

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

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

The definition of alphabetised parallel operator, where process P can write to state variables ns_1, and process Q can write to state variables ns_2, and both processes synchronise on the channels cs, is given by

Definition 4.1.29

$$P \; [| \, ns_1 | cs | ns_2 |] \; Q \;\; \widehat{=} \;\; (P;\; U0(out\alpha(P)) \parallel Q;\; U1(out\alpha(Q)))_{+\{v,tt\}};\; M_{CML}$$

Here Ui are relabelling functions. They map each observational variable obs of their arguments to $i.obs$. This ensures that the two sides of the parallel composition do not share variables, and hence do not interfere with each other. Subscripting an action with a set of events adds that set of events to the alphabet: $R_{\{+n\}} = R \land n = n'$. Here we add the original values of v and tt back.

M_{CML} is the ***CML*** merge function, and

Definition 4.1.30

$$M_{CML} \;\; \widehat{=} \;\; \begin{pmatrix} (ok' = P.ok \land Q.ok) \land \\ (wait' = P.wait \lor Q.wait) \land \\ (tt' \in P.tt \parallel_A Q.tt) \land \\ (v' = merge(v, P.v, Q.v)) \end{pmatrix};\; Skip$$

The function $merge$ merges the copies of state that the operands have taken. If process P is restricted to the name space ns_1, and process Q to ns_2, then $merge$ is defined as

Definition 4.1.31 (merging state)

$$merge(v, v1, v2) = (ns_1 \lhd v_1) \cup (ns_2 \lhd v_2) \cup ((ns_1 \cup ns_2) \ntriangleleft v)$$

where the domain restriction operator $ns \lhd v$ restricts the domain of the mapping v to the nameset ns, and the domain subtraction operator $ns \ntriangleleft v$ removes the nameset ns from the domain of the mapping v.

Using these new operators, we are in a position to define parallel composition.

A parallel composition will diverge on a trace tt' if it can be constructed as the trace composition of two traces tt_1 and tt_2, and one of the operands diverges at some point along either tt_1 or tt_2.

Lemma 4.1.9

$$(P \; [| \, ns_1 | cs | ns_2 |] \; Q)^f_f$$
$$=$$
$$\exists tt_1, tt_2 \bullet \begin{pmatrix} (P^f_f[tt_1/tt'] \land Q^t_f[tt_2/tt'];\; \boldsymbol{RT1}(true)) \\ \lor \\ (P^t_f[tt_1/tt'] \land Q^f_f[tt_2/tt'];\; \boldsymbol{RT1}(true)) \end{pmatrix} \land tt' \in tt_1 \parallel_A tt_2$$

Definition 4.1.32 (Parallel composition)

$$P \;[\mid ns_1 | cs | ns_2 \mid]\; Q = \boldsymbol{RT} \left(\begin{array}{l} \forall\, tt_1, tt_2 \bullet \\ (tt' \in tt_1 \mid\mid_A tt_2) \Rightarrow \\ \qquad \neg \left(\begin{array}{l} ((P^f_f[tt_1/tt'] \wedge Q^t_f[tt_2/tt']);\; \boldsymbol{RT1}(true)) \\ \vee \\ ((P^t_f[tt_1/tt'] \wedge Q^f_f[tt_2/tt']);\; \boldsymbol{RT1}(true)) \end{array} \right) \\ \vdash \\ \exists\, wait_1, wait_2, tt_1, tt_2 \bullet \\ \qquad tt' \in tt_1 \parallel_A tt_2 \wedge \\ \qquad wait' = wait_1 \vee wait_2 \wedge \\ \qquad P^t_f[wait_1, tt_1, v_1/wait', tt', v'] \wedge \\ \qquad Q^t_f[wait_2, tt_2, v_2/wait', tt', v'] \wedge \\ \qquad v' = merge(v, v_1, v_2) \end{array} \right)$$

4.1.14 Interleaving parallel

Interleaving of two processes is a straightforward derived operator: it is formed as the parallel composition of two processes, communicating on an empty set of events.

Definition 4.1.33 (Interleaving) *If P and Q may modify variables in ns_1 ns_2 respectively, and $ns_1 \cap ns_1 = \emptyset$, then*

$$P \;|||\; Q = P \;[\mid ns_1 | \emptyset | ns_2 \mid]\; Q$$

4.1.15 Abstraction

The abstraction or 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; they must happen as soon as they become available. Once more, the definition is given using semantic functions.

The assumption of maximal progress is modelled by considering only the A-urgent traces of P: the traces where all possible occurrences of every event in A happen as soon as they become available. In an A-urgent trace all events in the set A will appear in all refusal sets: no further events from A can be performed at any point.

Definition 4.1.34 (Urgency) *If $A \subseteq \Sigma$ and $t \in timedTrace$, then*

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

The semantic trace hiding operator is defined inductively:

Definition 4.1.35 (Trace hiding) *If $A, S \subseteq \Sigma$; $a \in A$; $b \notin A$, then*

$$\begin{array}{rcl} \langle\rangle \setminus A & = & \langle\rangle \\ (\langle S \rangle \frown tt) \setminus A & = & \langle S \setminus A \rangle \frown (tt \setminus A) \\ (\langle a \rangle \frown tt) \setminus A & = & tt \setminus A \\ (\langle b \rangle \frown tt) \setminus A & = & \langle b \rangle \frown (tt \setminus A) \end{array}$$

The behaviour of a process with an internalised set of events A is derived only from the A-urgent traces of the process:

Definition 4.1.36 (Hiding)

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

An abstracted process will diverge on an observed trace in the case where we are able to identify a prefix of the observed trace for which there is a corresponding observation (t) which (1) is an observation of the unabstracted process (A urgent $t \wedge tt' = t \setminus A$), and (2) represents a divergence of the unabstracted process ($P^f_f[t/tt']$). ***RT1***$(true)$ indicates that we do not care what happens in the latter potion of the trace. Obviously this is a case to avoid.

$$\neg\,((\exists\, t \bullet P^f_f[t/tt'] \wedge A \;\mathsf{urgent}\; t \wedge tt' = t \setminus A)\;;\; \textbf{\textit{RT1}}(true))$$

Only A-urgent observations of the unabstracted process contribute to the postcondition of the abstracted process.

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

We form the design by ensuring that these observations are ***RT*** healthy.

$$P \setminus A = \textbf{\textit{RT}} \left(\begin{array}{l} \neg\,((\exists\, t \bullet P^f_f[t/tt'] \wedge A \;\mathsf{urgent}\; t \wedge tt' = t \setminus A)\;;\; \textbf{\textit{RT1}}(true)) \\ \vdash \\ (\exists\, t \bullet P^t_f[t/tt'] \wedge A \;\mathsf{urgent}\; t \wedge tt' = t \setminus A) \end{array} \right)$$

4.1.16 Recursion

In a timed semantics such as ours, the risk that recursive processes face is that they might attempt to engage in infinitely many events in a finite time. In [20] this risk is avoided by introducing a bounded speed constraint as an axiom (Definition 4.2.5 in Section 4.2), to apply to the whole semantic space. We use the same constraint, but apply it only in the definition of recursive processes.

Definition 4.1.37 (T5: Zeno freedom)

$$\textbf{\textit{T5}}(P) = P \wedge \forall\, k \bullet \exists\, n \bullet \forall\, tt' \bullet P \Rightarrow (dur(tt') \leq k \Rightarrow \#(trace(tt')) \leq n)$$

Given a process P, and a maximum observation duration k, it is always possible to identify a limit n on the amount of activity that P can exhibit, no matter which trace tt' it performs. Recursion is defined as the ***RT***-healthy least fixed point that satisfies ***T5***.

Definition 4.1.38 (Recursion)

$$\mu\, X \bullet F(X) = \textbf{\textit{RT}}(\textbf{\textit{T5}}(\sqcap\{\, P \mid F(P) \sqsubseteq P\}))$$

If a process is *well-timed*, as described in Section 4.2.6, we can be sure that it is Zeno free.

4.1.17 Timeout

The timeout $P \overset{n}{\triangleright} Q$ is the process which offers to behave as P for the first n time units. If P fails to begin communication by then, process Q silently takes over.

The precondition for the timeout process $P \overset{n}{\triangleright} Q$ comes in two parts, to exclude two possible behaviours. The first part deals with the case where the process has waited up to n time units without any visible event. The behaviours that we wish to exclude here are those in which the precondition of P has failed. P's precondition will fail to hold on any trace that we can divide up into two ***RT***-healthy portions, the first of which is of duration less than or equal to n time units, at the end of which P_f^f holds. ***RT1***$(true)$ ensures that no constraints are imposed on the second part of the trace. Obviously, we do not want this situation.

$$\neg\ ((events(tt') = \langle\rangle \land \#tt' \leq n) \land P_f^f\ ;\ \mathbf{RT1}(\mathbf{true}))$$

The second part of the precondition excludes the case where P's precondition held successfully over an interval of n time units, but at that point P's postcondition fails to establish the precondition for Q:

$$\neg\ ((P_f^t \land events(tt') = \langle\rangle \land \#tt' = n)\ ;\ (wait \land Q_f^f))$$

This can only occur if Q is initiated when P's value for $wait'$ is $true$; in which case this value will be transferred (by the sequential composition) to $wait$ for Q. Of course, Q is now initiated, so Q_f holds.

The postcondition for the timeout process $P \overset{n}{\triangleright} Q$ also has two parts. It initially offers to behave like P. If this offer is taken up before n time units have passed, then the entire observable behaviour is due to P:

$$(P_f^t \land \#(idleprefix(tt')) \leq n)$$

If no events occur and P hasn't terminated in the first n time units, the combined process proceeds to behave like Q, initialised with an arbitrary choice of state. Allowing Q to begin with the state of the previous process would mean that the timeout operator allowed the state of deadlocked processes to be exposed, and so $(v := 1)\ ;\ STOP \overset{2}{\triangleright} SKIP$ would be distinguishable from $(v := 2)\ ;\ STOP \overset{2}{\triangleright} SKIP$, even though $(v := 1)\ ;\ STOP$ is not distinguishable from $(v := 2)\ ;\ STOP$.

$$\exists\, v_0 \bullet ((P_f^t \land events(tt') = \langle\rangle \land \#tt' = n \land v = v_0)\ ;\ (wait \land Q_f^t[v_0/v]))$$

Note that the timeout operator is *non-strict* in the sense of [26, 28]: events of P can be performed (unstably) between the nth and n+1th tock.

Definition 4.1.39 (Timeout)

$$P \overset{n}{\rhd} Q = \mathbf{RT} \left(\begin{array}{l} \neg\left((events(tt') = \langle\rangle \wedge \#tt' \leq n) \wedge P_f^f \mathbin{;} \mathbf{RT1}(\mathbf{true})\right) \\ \wedge \\ \neg\left((P_f^t \wedge events(tt') = \langle\rangle \wedge \#tt' = n) \mathbin{;} (wait \wedge Q_f^f)\right) \\ \vdash \\ (P_f^t \wedge \#(idleprefix(tt')) \leq n) \\ \vee \\ \exists v_0 \bullet ((P_f^t \wedge events(tt') = \langle\rangle \wedge \#tt' = n \wedge v = v_0) \mathbin{;} (wait \wedge Q_f^t[v_0/v])) \end{array} \right)$$

4.1.18 Untimed Timeout

The untimed variant of the timeout operator $P \rhd Q$ allows the first process to be timed out at any time. It is defined directly in the same way as the timeout operator (Section 4.1.17), but with no restriction on the point at which the timeout may occur.

Definition 4.1.40 (Untimed timeout)

$$P \rhd Q = \mathbf{RT} \left(\begin{array}{l} \neg\left((events(tt') = \langle\rangle \wedge P_f^f) \mathbin{;} \mathbf{RT1}(\mathbf{true})\right) \\ \wedge \\ \neg\left((P_f^t \wedge events(tt') = \langle\rangle) \mathbin{;} (wait \wedge Q_f^f)\right) \\ \vdash \\ P_f^t \\ \vee \\ \exists v_0 \bullet ((P_f^t \wedge events(tt') = \langle\rangle \wedge v = v_0) \mathbin{;} (wait \wedge Q_f^t[v_0/v])) \end{array} \right)$$

4.1.19 Wait

The delay operator is defined as a timeout process. It behaves like $STOP$ for a specified number of time units, then terminates successfully.

Definition 4.1.41 (Delay)

$$Wait(n) = STOP \overset{n}{\rhd} SKIP$$

4.1.20 Interrupt

Like the timeout operator, the interrupt operator also has timed and untimed versions. Unlike the timeout operator, however, the basic form of the interrupt operator is the untimed version, written $P \bigtriangleup Q$. The untimed interrupt initially behaves like P, except that it cannot refuse to engage in the initial events of Q. If any of the initial events of Q occurs, the combined process starts to behave like Q.

We begin by defining the set of initial events of a process. An event a is an initial event of R if it is the initial event of a trace of R.

Definition 4.1.42

$$initials(R) = \{a \mid \langle a\rangle \leq events(tt_0) \bullet R[tt_0/tt']\}$$

The condition $notoffered(A, t)$ holds when no events in A have been offered during the observation of t. In this case, no events in A will have been observed or refused in the trace t.

Definition 4.1.43 *If* $A \subseteq \Sigma$ *and* $t \in timedTrace$*, then*

$$notoffered(A, t) = t \upharpoonright A = \langle\rangle \wedge A \cap refusals(t) = \emptyset$$

Like the timeout operator, the precondition comes in two parts to exclude two possible cases. The first is the case where we can divide the trace up into two parts, and identify an initial seqment over which the initial events of Q have not been offered, and on which the precondition of P fails to hold. This is a case to exclude.

$$\neg\, (notoffered(initials(Q), tt') \Rightarrow P_f^f \,;\, \boldsymbol{RT1}(true))$$

The second case to avoid is when the interrupt is triggered at a point at which P is failing to establish the precondition of Q. P must have held up to the point of interruption. As in the corresponding case for timeout, P must not have terminated after the first segment, so $wait'$ holds and this value is transferred via the relational composition to $wait$.

$$\neg\, ((P_f^t \wedge notoffered(initials(Q), tt')) \,;\, (wait \wedge Q_f^f))$$

The postcondition describes two cases. In the first, P has terminated successfully (which is only possible if none of the initial events of Q were offered) and the relational composition transfers control to $SKIP$. In the second control is transferred via an initial event of Q. Q begins with a non-deterministic choice of state, again to avoid the possibility of exposing the state of a deadlocked process.

$$\exists\, v_0 \bullet ((P_f^t \wedge notoffered(initials(Q), tt')) \,;\, (SKIP \vee (wait \wedge Q_f^t[v_0/v])))$$

These parts combine in the definition of the interrupt operator.

Definition 4.1.44 (Interrupt)

$$P \triangle Q = \boldsymbol{RT}\left(\begin{array}{l} \neg\, (notoffered(initials(Q), tt') \Rightarrow P_f^f \,;\, \boldsymbol{RT1}(true)) \wedge \\ \neg\, ((P_f^t \wedge notoffered(initials(Q), tt')) \,;\, (wait \wedge Q_f^f)) \\ \vdash \\ \exists\, v_0 \bullet ((P_f^t \wedge notoffered(initials(Q), tt')) \,;\, (SKIP \vee (wait \wedge Q_f^t[v_0/v]))) \end{array}\right)$$

4.1.21 Timed Interrupt

The timed version of the interrupt operator allows the first process to continue for a specified number of time units, after which it will be interrupted by Q. Note that the timed interrupt operator is not invoked if the first process *terminates* before the time value, whereas a timeout operator is not invoked if the first process *begins* before the time value.

We begin with the definition of the *duration* of a trace, written $dur(t)$, as being the number of time units that elapse during a trace. This definition ignores the events in the trace, and so differs from the way that we calculate the passage of time in Section 4.1.17, where we know that no events have occurred.

Definition 4.1.45 *Let $A \subseteq \Sigma$, $a \in \Sigma$ and $t \in timedTrace$. Then*

$$dur(t) = \#(tocks(t) \upharpoonright \{tock\})$$

The precondition of timed interrupt has two parts, again disallowing two ways in which divergence can arise. The first is the case where an initial segment of the trace has a duration of less than or equal to the interrupt parameter, and leads to the case where P diverges.

$$\neg\,(dur(tt') \leq n \Rightarrow P^f_f \; ; \; \mathbf{RT1}(true))$$

The second case is the one where P fails to establish the precondition for P at the point of interruption.

$$\neg\,(dur(tt') = n \wedge P^t_f) \; ; \; (wait \wedge Q^f_f)$$

The postcondition also has two parts. Either the duration of the trace is less than the interrupt parameter, or P is interrupted and the subsequent behaviour is from Q.

Definition 4.1.46

$$P \overset{n}{\triangle} Q = \mathbf{RT}\left(\begin{array}{l} \neg\,(dur(tt') \leq n \Rightarrow P^f_f \; ; \; \mathbf{RT1}(true)) \\ \wedge \\ \neg\,(dur(tt') = n \wedge P^t_f) \; ; \; (wait \wedge Q^f_f) \\ \vdash \\ P^t_f \wedge dur(tt') \leq n \\ \vee \\ (P^t_f \wedge dur(tt') = n) \; ; \; (wait \wedge Q^t_f) \end{array}\right)$$

During the first n time units control cannot transfer to Q, and P is unhindered. Either it terminates before the n time units, or at n time units it will be interrupted by Q.

4.1.22 Startsby

The *startsby* operator insists that a process begins communication by a deadline. The process $P\ startsby(n)$ behaves like P, and exhibits miraculous behaviour if P hasn't engaged in an event in the first n time units.

Definition 4.1.47 (startsby)

$$P\ startsby(n) = P \overset{n}{\triangleright} MIRACLE$$

4.1.23 Endsby

The *endsby* operator insists that a process terminates by a deadline, otherwise it exhibits infeasible behaviour.

Definition 4.1.48

$$P\ endsby(n) = P \overset{n}{\triangle} MIRACLE$$

4.1.24 While

The while loop recusively behaves like P so long as the condition b holds. If b fails, it terminates. It is defined as a derived operator, using recursion and the conditional operator (defined in Section 2.3.)

Definition 4.1.49 (While loop)

$$b * P = \mu X \bullet (P \mathbin{;} X) \lhd b \rhd \mathit{SKIP}$$

4.1.25 Guarded actions

The guarded action $[g] \& P$ behaves as P if the guard g holds, otherwise it stops.

Definition 4.1.50

$$[g] \& P = P \lhd g \rhd \mathit{STOP}$$

4.2 Lowe & Ouaknine's Axioms

Our semantic domain is inspired by that of Lowe & Ouaknine [20]. They start with five axioms, some of which we can consider as theorems of our definitions. We present them below, modified to take into account the removal of *tock* from the semantic domain. The proofs of these theorems (for a slightly earlier version of *CML*) have been given in Deliverable 23.3.

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)

$$\boldsymbol{T1}(P) = \exists\, v', \mathit{wait}' \bullet P[\langle\rangle / \mathit{tt}']$$

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

4.2.2 Prefix Closure

The second axiom states that the traces of every process are precedence closed: if tt' is a trace of P, then so is every u such that $u \preceq \mathit{tt}'$. This ensures that the history of a system evolves in a smooth way, event by event, and that the refusal sets observed are downward closed.

Definition 4.2.2 (T2: Precedence closure of timed traces)

$$\boldsymbol{T2}(P) = P \land \forall\, u \preceq \mathit{tt}' \bullet \exists\, v' \bullet P[u, \mathit{true} / \mathit{tt}', \mathit{wait}]$$

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

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)

$$\textbf{\textit{T3}}(P) = P \wedge wait \Rightarrow \left(\begin{array}{l} \forall\, A, a \mid \\ \left(\begin{array}{l} \exists\, v', wait' \bullet P[tt' \frown \langle A \rangle / tt'] \wedge \\ \forall\, v', wait' \bullet \neg\, P[tt' \frown \langle a \rangle / tt'] \end{array} \right) \Rightarrow \\ \quad \exists\, v', wait' \bullet P[tt' \frown \langle A \cup \{a\} \rangle / tt'] \end{array} \right)$$

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

4.2.4 Timelock Freedom

Most processes always allow time to pass. Assignment and SKIP terminate immediately so there is no opportunity for time events to occur. *Timelock freedom* allows time to pass under all circumstances.

Definition 4.2.4 (T4: Timelock freedom)

$$\textbf{\textit{T4}}\,(P) = P \wedge wait' \Rightarrow \exists\, v', wait' \bullet P[tt' \frown \langle \emptyset \rangle / tt']$$

Note that ***T3*** means that insisting on the empty refusal is sufficient to ensure that the remaining refusal sets will also be present.

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

4.2.5 Zeno Freedom

Lowe & Ouaknine's bounded-speed condition gives a bound n on the number of events that can be performed in the first k time units.

Definition 4.2.5 (T5: Zeno freedom)

$$\textbf{\textit{T5}}\,(P) = P \wedge \forall\, k \bullet \exists\, n \bullet \forall\, tt' \bullet P \Rightarrow (\#tocks(tt') \leq k \Rightarrow \#(trace(tt')) \leq n)$$

We say that a recursive process is well-timed if it cannot recurse without time passing. A syntactic check for well-timed processes is given in Section 4.2.6.

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

4.2.6 Well-timed processes

As mentioned in Section 4.1.16, a syntactic check is available to ensure that a ***CML*** process is time-guarded.

The following terms and definitions are adapted from [27] and [33].

A ***CML*** synactic term is *time-active* if some strictly positive amount of time must elapse before the term terminates. A term is *time-guarded* for X if any execution of it must consume some strictly positive amount of time before a recursive call for X can be reached. Lastly, a program is *well-timed* when all of its recursions are time-guarded. Note that, because all delays are integral, some "strictly positive amount of time" in this context automatically means at least one time unit.

Definition 4.2.6 (Time-active) *If $f : \Sigma \to \Sigma$ then the collection of time-active terms is the smallest set of terms such that:*

- *$STOP$ is time-active;*
- *$Wait(n)$ is time-active for $n \geq 1$;*
- *If P is time-active, then so are $a \to P$, $P \parallel_A Q$, $Q \parallel_A P$, $P \; ; \; Q$, $Q \; ; \; P$, $P \setminus A$, $f(P)$, $f^{-1}(P)$, $\mu\, X \bullet P(X)$ and $P \overset{n}{\rhd} Q$ for $n \geq 1$;*
- *If P and Q are time-active, then so are $P \overset{n}{\rhd} Q$, $P \bigtriangleup Q$, $P \,\Box\, Q$, $P \sqcap Q$, $P \parallel_A Q$, and $P \parallel\!| \; Q$.*

Definition 4.2.7 (Time-guardedness) *If X, Y are* ***CML*** *process variables, and A,$B \subseteq \Sigma$, then the* ***CML*** *terms which are* time-guarded *for X, is the smallest set of terms that may be constructed from the following rules:*

- *$STOP$, $SKIP$, $Wait(n)$ and $\mu\, X \bullet P(X)$ are time-guarded for X;*
- *$Y \neq X$ is time-guarded for X;*
- *If P is time-guarded for X, then so are $a \to P$, $P \setminus A$, $f(P)$, $\mu\, Y \bullet P(X)$ and $P \overset{n}{\rhd} Q$, for $n \geq 1$;*
- *If P and Q are time-guarded for X then so are $P \overset{n}{\rhd} Q$, $P \,\Box\, Q$, $P \sqcap Q$, $P \; ; \; Q$, $P \parallel_A Q$, and $P \parallel\!| \; Q$;*
- *If P is time-guarded for X and time-active, then $P \; ; \; Q$ is time-guarded for X.*

We may construct *well-timed* processes using the above definition.

Definition 4.2.8 (well-timed) *A term is* well-timed *if every subterm of the form $\mu\, X \bullet P(X)$ is such that P is time-guarded for X.*

Chapter 5

Example Galois Connections

This chapter presents three examples of Galois connections between ***CML*** theories.

5.1 From Relations to Designs

Our first example is the most basic. We look for a Galois connection between the lattice of nondeterministic programs provided by the theory of relations and the lattice with the same signature provided by the theory of designs. These two theories lie at the heart of ***CML***. We start by defining the left adjoint, which we call Des. This maps pure relations into the lattice of designs: Des : Relations $\to$ Designs. Both lattices are ordered by refinement.

The semantics of nondeterministic programs in Relations famously exclude a treatment of termination, so when we map a relation R into a design, we have to decide how to handle this. R can have no description of when it terminates, and its correctness against a specification must be judged with the assumption that it terminates (it is a statement of partial correctness). We can encode both these decisions using the healthiness condition for designs, ***H***, together with the requirement that the program must terminate.

Definition 5.1.1 **(*Des*)**

$$Des(R) \mathrel{\widehat{=}} \boldsymbol{H}(R \land ok')$$

□

The following law is more explicit about $Des(R)$. First recall from [9] the alternative characterisation of ***H2*** and its monotonicity in the ok variable:

$$\boldsymbol{H2}(P) = P \,;\, J$$

$$J \mathrel{\widehat{=}} (ok \Rightarrow ok') \land I\!I(\alpha P \setminus \{ok, ok'\})$$

A key property of this definition is know as J-splitting:

$$P \,;\, J = P^f \lor (P^t \land ok')$$

Law 5.1.1 (*Des* **Design**)

$$Des(R) = \textbf{true} \vdash R$$

Proof 4

$$\begin{aligned}
& Des(R) \\
& = \{\ \text{Def 5.1.1 } Des\ (Des(R)\ \widehat{=}\ \mathbf{H}(R \land ok'))\ \} \\
& \mathbf{H}(R \land ok') \\
& = \{\ \mathbf{H} \text{ and } \mathbf{H2}\ \} \\
& \mathbf{H1}(R \land ok'\ ;\ J) \\
& = \{\ J\text{-splitting}\ \} \\
& \mathbf{H1}((R \land ok')^f \lor ((R \land ok')^t \land ok')) \\
& = \{\ \text{substitution}\ \} \\
& \mathbf{H1}((R \land \textbf{false}) \lor (R \land \textbf{true} \land ok')) \\
& = \{\ \text{propositional calculus}\ \} \\
& \mathbf{H1}(\textbf{false} \lor (R \land ok')) \\
& = \{\ \text{propositional calculus}\ \} \\
& \mathbf{H1}(R \land ok') \\
& = \{\ \mathbf{H1}\ \} \\
& ok \Rightarrow ok' \land R \\
& = \{\ \text{design}\ \} \\
& \textbf{true} \vdash R
\end{aligned}$$

□

The right adjoint is called Rel, and it maps from Designs into Relations. Its job is to throw away the information about initiation and termination in a design to extract the underlying relation. It does this by considering only the case that the design is started and finishes properly: $Rel(D) = D[true, true/ok, ok']$. There is a shorthand for this particular substitution: D^{tt}.

Definition 5.1.2 (*Rel*)

$$Rel(D)\ \widehat{=}\ D^{tt}$$

□

Again we can be more explicit.

Law 5.1.2 (*Rel* **Design**)

$$Rel(P \vdash Q) = P \Rightarrow Q$$

Proof 5

$Rel(P \vdash Q)$
$= \{$ Def 5.1.2 Rel $(Rel(P \vdash Q) \;\widehat{=}\; (P \vdash Q)^{tt})$ $\}$
$(P \vdash Q)^{tt}$
$= \{$ design $\}$
$(ok \wedge P \Rightarrow ok' \wedge Q)^{tt}$
$= \{$ substitution $\}$
$(ok \wedge P \Rightarrow ok' \wedge Q)[\,\mathbf{true}, \mathbf{true}/ok, ok'\,]$
$= \{$ substitution $\}$
$\mathbf{true} \wedge P \Rightarrow \mathbf{true} \wedge Q$
$= \{$ propositional calculus $\}$
$P \Rightarrow Q$

□

This pair of functions form a Galois connection: $(\mathsf{Designs}, \sqsupseteq) \underset{Rel}{\overset{Des}{\rightleftarrows}} (\mathsf{Relations}, \sqsupseteq)$.

Theorem 5.1.1 ((Rel) Galois connection)

(Des, Rel) is a Galois connection

Proof 6 *S.T.P.*[1] $Des(R) \sqsupseteq P \vdash Q$ **iff** $R \sqsupseteq Rel(P \vdash Q)$

$Des(R) \sqsupseteq P \vdash Q$
$= \{$ Def 5.1.1 Des $(Des(R) \;\widehat{=}\; \mathbf{H}(R \wedge ok'))$ $\}$
$(\mathbf{true} \vdash R) \sqsupseteq (P \vdash Q)$
$= \{$ design refinement $\}$
$[\,P \Rightarrow \mathbf{true}\,] \wedge [\,P \wedge R \Rightarrow Q\,]$
$= \{$ predicate calculus $\}$
$\mathbf{true} \wedge [\,P \wedge R \Rightarrow Q\,]$
$= \{$ propositional calculus $\}$
$[\,P \wedge R \Rightarrow Q\,]$

$R \sqsupseteq Rel(P \vdash Q)$
$= \{$ Def 5.1.2 Rel $(Rel(P \vdash Q) \;\widehat{=}\; (P \vdash Q)^{tt})$ $\}$
$R \sqsupseteq P \Rightarrow Q$
$= \{$ refinement $\}$
$[\,R \Rightarrow P \Rightarrow Q\,]$
$= \{$ propositional calculus $\}$
$[\,P \wedge R \Rightarrow Q\,]$

□

[1] "S.T.P." is a shorthand in mathematical proofs for "it is sufficient to prove…"

The Galois connection (Des, Rel) is a coretract.

Lemma 5.1.1 **(Des injective)**

Des is injective

Proof 7 *S.T.P.* $Rel \circ Des = \mathrm{id}_R$

$$
\begin{array}{l}
Rel \circ Des(R) \\
= \{ \text{Law 5.1.1 } Des \text{ Design } (Des(R) = \textbf{true} \vdash R) \} \\
Rel(\textbf{true} \vdash R) \\
= \{ \text{Def 5.1.2 } Rel\ (Rel(P \vdash Q) \ \widehat{=}\ (P \vdash Q)^{tt}) \} \\
\textbf{true} \Rightarrow R \\
= \{ \text{propositional calculus} \} \\
R
\end{array}
$$

□

Lemma 5.1.2 **((Des, Rel) Properties)**

1. *Rel is surjective*
2. *(Des, Rel) is a coretract*
3. *Des is an order similarity:* $(Des(R) \sqsubseteq Des(S)) = (R \sqsubseteq S)$

Proof 8 *Since Des is injective.*

□

5.2 From Designs to Reactive Processes

The third semantic domain in *CML* is that of reactive processes. In Hoare & He's work [16], there are three reactive healthiness conditions (***R1***, ***R2***, and ***R3***) and an additional two specific to CSP.

$$
\begin{array}{lcl}
\textbf{R1}(P) & = & P \wedge tr \leq tr' \\
\textbf{R2}(P(tr, tr')) & = & P(\langle\rangle, tr' - tr) \\
\textbf{R3}(P) & = & I\!I_{rea} \lhd wait \rhd P
\end{array}
$$

The first says that the trace of events only ever increases; the second that a process is oblivious to the preceding trace; and the third that a process is sensitive to whether the preceding process has terminated or not. The semantics of basic *CML* is given in terms of the composition of the healthiness conditions for designs and for reactive processes, exploiting the fact that the two CSP conditions are the reactive analogues of the two design healthiness conditions. In this section we formalise this notion by setting out the key Galois connection between designs and reactive processes.

The healthiness conditions ***R2*** and ***R3*** commute with both ***H1*** and ***H2***. This means that they preserve designs. ***R1***, on the other hand, does not commute with ***H1***:

$$\begin{aligned}\mathbf{H1} \circ \mathbf{R1}(P) &= ok \Rightarrow P \wedge (tr \le tr') \\ \mathbf{R1} \circ \mathbf{H1}(P) &= (ok \Rightarrow P) \wedge (tr \le tr')\end{aligned}$$

In fact, $\mathbf{R1} \circ \mathbf{H1} = \mathbf{CSP1}$. For this reason, it is interesting to study the relationship between ***R1*** and ***H***, which, as we see below, turns out to be a retract.

Theorem 5.2.1 (($\mathbf{H}, \mathbf{R1}$) is a Galois connection)

($\mathbf{H}, \mathbf{R1}$) is a Galois connection

Proof 9 *S.T.P. that the following two conditions hold:*

$$\mathbf{H} \circ \mathbf{R1}(P \vdash Q) \sqsupseteq \mathrm{id}$$
$$\mathrm{id} \sqsupseteq \mathbf{R1} \circ \mathbf{H}$$

$$\begin{aligned}
&\mathbf{H} \circ \mathbf{R1}(P \vdash Q)\\
&= \{\ \textit{design}\ \}\\
&\mathbf{H} \circ \mathbf{R1}(\neg\, ok \vee \neg\, P \vee (ok' \wedge Q))\\
&= \{\ \textit{\textbf{R1} disjunctive}\ \}\\
&\mathbf{H}(\mathbf{R1}(\neg\, ok) \vee \mathbf{R1}(\neg\, P) \vee \mathbf{R1}(ok' \wedge Q))\\
&= \{\ \textit{property of \textbf{H}: } (\mathbf{H}(P) = \mathbf{H}(P[true/ok])\ \}\\
&\mathbf{H}(\mathbf{R1}(false) \vee \mathbf{R1}(\neg\, P) \vee \mathbf{R1}(ok' \wedge Q))\\
&= \{\ \textit{propositional calculus}\ \}\\
&\mathbf{H}(\mathbf{R1}(\neg\, P) \vee \mathbf{R1}(ok' \wedge Q))\\
&= \{\ \mathbf{H}\ \}\\
&\mathbf{H1} \circ \mathbf{H2}(\mathbf{R1}(\neg\, P) \vee \mathbf{R1}(ok' \wedge Q))\\
&= \{\ \textit{\textbf{H2}-}J\textit{-splitting}\ \}\\
&\mathbf{H1}\left((\mathbf{R1}(\neg\, P) \vee \mathbf{R1}(ok' \wedge Q))^f \vee (ok' \wedge (\mathbf{R1}(\neg\, P) \vee \mathbf{R1}(ok' \wedge Q))^t)\right)\\
&= \{\ \textit{substitution}\ \}\\
&\mathbf{H1}\left(((\mathbf{R1}(\neg\, P))^f \vee (\mathbf{R1}(ok' \wedge Q))^f \vee (ok' \wedge ((\mathbf{R1}(\neg\, P))^t \vee (\mathbf{R1}(ok' \wedge Q))^t))\right)\\
&= \{\ \textit{substitution}\ \}\\
&\mathbf{H1}\left(\mathbf{R1}(\neg\, P^f) \vee \mathbf{R1}((ok' \wedge Q)^f) \vee (ok' \wedge (\mathbf{R1}(\neg\, P^t) \vee \mathbf{R1}((ok' \wedge Q)^t)))\right)\\
&= \{\ \textit{substitution}\ \}\\
&\mathbf{H1}\left(\mathbf{R1}(\neg\, P^f) \vee \mathbf{R1}(false \wedge Q^f) \vee (ok' \wedge (\mathbf{R1}(\neg\, P^t) \vee \mathbf{R1}(true \wedge Q^t)))\right)\\
&= \{\ \textit{propositional calculus}\ \}\\
&\mathbf{H1}\left(\mathbf{R1}(\neg\, P^f) \vee \mathbf{R1}(false) \vee (ok' \wedge (\mathbf{R1}(\neg\, P^t) \vee \mathbf{R1}(true \wedge Q^t)))\right)\\
&= \{\ \mathbf{R1}\ \}\\
&\mathbf{H1}\left(\mathbf{R1}(\neg\, P^f) \vee false \vee (ok' \wedge (\mathbf{R1}(\neg\, P^t) \vee \mathbf{R1}(true \wedge Q^t)))\right)\\
&= \{\ \textit{propositional calculus}\ \}
\end{aligned}$$

$$
\begin{array}{l}
\mathbf{H1}\left(\mathbf{R1}(\neg\, P^f) \vee (ok' \wedge (\mathbf{R1}(\neg\, P^t) \vee \mathbf{R1}(Q^t)))\right) \\
= \{\ \text{assumption: } ok' \text{ not free in } P \text{ or } Q\ \} \\
\mathbf{H1}\,(\mathbf{R1}(\neg\, P) \vee (ok' \wedge (\mathbf{R1}(\neg\, P) \vee \mathbf{R1}(Q)))) \\
= \{\ \mathbf{H1}\ \} \\
ok \Rightarrow \mathbf{R1}(\neg\, P) \vee (ok' \wedge (\mathbf{R1}(\neg\, P) \vee \mathbf{R1}(Q))) \\
= \{\ \text{propositional calculus}\ \} \\
ok \wedge \neg\, \mathbf{R1}(\neg\, P) \Rightarrow ok' \wedge (\mathbf{R1}(\neg\, P) \vee \mathbf{R1}(Q)) \\
= \{\ \text{propositional calculus}\ \} \\
ok \wedge \neg\, \mathbf{R1}(\neg\, P) \Rightarrow ok' \wedge \mathbf{R1}(Q) \\
= \{\ \text{design}\ \} \\
\neg\, \mathbf{R1}(\neg\, P) \vdash \mathbf{R1}(Q) \\
= \{\ \text{conjugate } \mathbf{R1}\ \} \\
\overline{\mathbf{R1}}(P) \vdash \mathbf{R1}(Q) \\
\sqsupseteq \{\text{since } [\, P \Rightarrow \overline{\mathbf{R1}}(P)\,] and [\, P \wedge \mathbf{R1}(Q) \Rightarrow Q\,]\ \} \\
P \vdash Q
\end{array}
$$

$$
\begin{array}{l}
\mathbf{R1} \circ \mathbf{H}(P) \\
= \{\ \mathbf{R1\text{-}H\text{-}CSP1\text{-}CSP2}\ \} \\
\mathbf{CSP1} \circ \mathbf{CSP2}(P) \\
= \{\ \text{assumption: } P \text{ is } \mathbf{CSP}\ \} \\
P
\end{array}
$$

□

5.3 From Reactive Processes to Time

Our final example links basic *CML* to timed *CML*. Recall that the trace variable tr and the refusal variable ref in basic *CML* are replaced by the single timed trace rt in timed *CML*. We establish a Galois connection that links these variables. All we need to do is to specify one of the adjuncts and then calculate the other. We choose the left adjoint L : Timed $\rightarrow$ Reactive, as it is easy to specify since it forgets all the information about time represented in rt and rt'.

Definition 5.3.1

$$
\begin{array}{lll}
L(P) & \widehat{=} & \exists\, rt, rt' \bullet \quad P \\
& & \qquad \wedge\ (tr = events(rt)) \wedge (tr' = events(rt')) \\
& & \qquad \wedge\ (ref = last(refsduring(rt)) \wedge (ref' = last(refsduring(rt'))
\end{array}
$$

As we know, one adjoint in a Galois connection uniquely determines the other. We can think of $R(Q)$ as finding a schedule for the events and refusals in Q, but which schedule would be appropriate? The answer is provided by the calculation needed for R.

Definition 5.3.2

$$R(Q) \mathrel{\widehat{=}} \bigsqcap \{ P \mid L(P) \sqsupseteq Q \}$$

This is the weakest possible schedule.

L and R can be used to check properties of *CML* processes, to structure them into architectural patterns, and as part of system development techniques.

5.4 Conclusion

Our initial work on Galois connections opens up some interesting avenues of work.

- If P is a fixed point of $R \circ L$ (Defs 5.3.1 and 5.3.2, then it is *time insensitive*. This may be an important structural property.
- Sherif [34] uses a similar Galois connection as an architectural pattern for real-time systems. In his work, a *CircusTime* process is translated into a timeless *Circus* that interacts with a set of clocks; collectively, they implement the timed specification. The strategy for translating the specification is based on using the left adjoint to forget timing information, whilst introducing the required clock interactions.
- A recommended development strategy for Handel-C programs on FPGAs is to ignore timing properties initially and produce a network of communicating processes with the required basic functionality. Once this is completed, communications and state assignments should then be scheduled synchronously. Handel-C is similar to *CML* and *Circus*, and so the scheduling could be carried out as a translation based on our right adjoint R.

Chapter 6

Reference Semantics

Our semantics for object orientation is a copy semantics: objects denote values, not references. In this chapter, we introduce an independent UTP theory of references based on Reynolds' and O'Hearn's separation logic [23, 31]. We cast our theory as an embedding in the most basic UTP setting of the theory of relations. It can then be added to any of the other theories in CML's semantics in a simple and uniform way as an embedding.

6.1 Separation Logic in UTP

We present separation logic as a programming theory in UTP. Separation logic was originally conceived as an extension of Hoare logic for reasoning about programs that use pointers [23, 31], although it is also applicable to reasoning about the ownership of resources and about virtual separation between parallel programs with shared state. To understand the problem being addressed, consider the assignment rule in Hoare logic.

$$\frac{}{\{\, p[e/x] \,\}\ x := e\ \{\, p \,\}}$$

We use this rule to calculate the precondition for $x := 10$ to achieve a postcondition $x = 10 \land y = 0$ as $(x = 10 \land y = 0)[10/x]$, which is simply $y = 0$.

Now suppose that x is a reference variable denoting an address in memory, not a simple value. Let the expression $[x]$ be the value obtained by dereferencing x; that is, by looking up the address and reading its contents, which could be a constant, another address, or a record combining a mixture of both. Reference variables are created on the heap, which is memory set aside for dynamic allocation. Two reference variables can be aliases for the same address, so that modifying the value addressed by a reference variable will implicitly modify the values associated with all aliases, and this may be surprising. As a result, aliasing makes it particularly difficult to understand, analyse, and optimise programs.

Consider the assignment $[x] := 10$ and the postcondition $[x] = 10 \land [y] = 0$. We calculate the precondition $([x] = 10 \land [y] = 0)[10/[x]]$, which simplifies to $[y] = 0$. So the before-value of $[x]$ is unimportant and the before-value of $[y]$ must be 0; providing the latter holds, the assignment makes the postcondition true; but what if x and y point to the same address? Afterwards, this address must be both 10 and 0; this can mean only that the standard rule for assignment is unsound in the presence of aliasing. The problem can be fixed in an ad hoc way by adding the precondition that there is no aliasing.

Separation logic is specifically designed to overcome this problem. We show how to give a semantics in UTP to separation logic and its characteristic frame rule that allows compositional reasoning about reference variables and the heap. In UTP, we avoid using an environment to describe the current state of a program; instead, we identify a program variable with its meaning as a mathematical variable. We extend this by adding an observation variable to represent the heap. Instead of talking about memory addresses, we abstract a little and discuss object identifiers and field names. For example, if we have an object type with two fields *int* and *next*, then an observation of our heap could be the function:

$$\begin{array}{l}\{(o_1, int) \mapsto 3, (o_2, int) \mapsto 4, (o_3, int) \mapsto 2, \\ \quad (o_1, next) \mapsto o_2, (o_2, next) \mapsto o_3, (o_3, next) \mapsto \mathsf{null}\}\end{array}$$

If our reference variable x has the object identifier o_1 as its reference, then this heap describes a linked list that represents the sequence $\langle 3, 4, 2 \rangle$. In what follows, we treat the object identifier and field name pair as though it were simply an object identifier. Let Obj be the set of object identifiers and Val be the set of values (constants or object identifiers, or the special null value); then $hp : \mathsf{Obj} \nrightarrow \mathsf{Val}$ represents the state of the heap. Heap predicates constrain hp; they do not make sense unless all their reference variables are defined to be on the heap. We formalise this as a healthiness condition. Let $\mathsf{fv}(P)$ be the set of free program variables mentioned in P.

Definition 6.1.1 (Heap predicate SL1) *P is a healthy heap predicate providing it is a fixed point of the function: $\mathbf{SL1}(P) = P \land \mathsf{fv}(P) \subseteq \operatorname{dom} hp$. $\mathbf{SL1}$-healthy predicates are called simply "heap predicates".* □

Definition 6.1.2 (Compatible join) *Define the compatible join of two heaps as:* $hp \circledast (h_1, h_2) \mathrel{\widehat{=}} \operatorname{dom} h_1 \cap \operatorname{dom} h_2 = \emptyset \land hp = h_1 \cup h_2$. □

The key operator in separation logic is the separating conjunction. In its definition, we use the shorthand: $p_h = p[h/hp]$; later, we also use $Q_h^{h'} = Q[h, h'/hp, hp']$.

Definition 6.1.3 (Separating conjunction) *The binary operator $*$ (pronounced "star" or "separating conjunction") asserts that the heap can be split into two disjoint parts where its two arguments hold, respectively.*

$$p * q \mathrel{\widehat{=}} \exists h_1, h_2 \bullet hp \circledast (h_1, h_2) \land p_{h_1} \land q_{h_2}$$ □

We introduce a healthiness condition on relations on heaps to capture the idea that a program will depend only upon, or change the part of, the initial heap within its footprint.

Definition 6.1.4 (Frame property SL2) *Suppose that the heap can be partitioned into subheaps h_1 and h_2 and that all of Q's reference variables are on the h_1 subheap:* $hp \circledast (h_1, h_2) \land \mathsf{fv}(Q_{hp}^{hp'}) \subseteq \operatorname{dom} h_1$. *Then Q is independent of the subheap h_2 if it is a fixed point of the function:*

$$\mathbf{SL2}(Q_{hp}^{hp'}) = Q_{hp}^{hp'} \land \exists h_1' \bullet hp' \circledast (h_1', h_2) \land Q_{h_1}^{h_1'}$$

$\mathbf{SL2}$-healthy predicates are said to have the frame property. □

In the standard account of separation logic, the frame property is proved as a theorem of the operational semantics of the programming language, but we make it a basic healthiness condition. The set of healthy predicates must then be shown to be closed under (the denotational semantics of) the program operators.

Frame-property-healthy predicates support modular reasoning in separation logic. To demonstrate this, we define Hoare triples and prove the frame rule.

Definition 6.1.5 (Hoare triple) *The correctness of a program Q is a refinement assertion:* $\{\, p \,\}\ Q\ \{\, r \,\} \ \widehat{=}\ (p \Rightarrow r') \sqsubseteq Q$, *providing* $[\, p \Rightarrow \mathsf{fv}(Q) \subseteq \mathrm{dom}\, hp \,]$. □

Now we are ready for the central result in separation logic, the frame rule, which is the basis for the logic's local reasoning technique. This says that if a program Q can execute safely in a local state satisfying p, then it can also execute in any bigger state satisfying $p * s$ and that its execution will not affect this additional part of the state, and so s will remain true after execution.

Theorem 6.1.1 (Frame Rule) $\dfrac{\{\, p \,\}\ Q\ \{\, r \,\}}{\{\, p * s \,\}\ Q\ \{\, r * s \,\}}\ [\, \mathsf{fv}(Q) \cap \mathsf{fv}(s) = \emptyset \,]$

Proof: *See Fig. 6.1.*[1] □

The main purpose of separation logic is to reason about programs that manipulate the heap and, in particular, the assignment statement. Before we can give the inference rule for doing this, we need two more pieces of notation: separating implication and heaplets.

Separation logic has a separating implication ($-\!\!*$, known as "magic wand") that asserts that extending the heap with a disjoint part that satisfies its first argument results in a heap that satisfies the second argument.

Definition 6.1.6 $p -\!\!* q \ \widehat{=}\ \forall\, h_1, h_2 \bullet h_1 \circledast (hp, h_2) \land p_{h_2} \Rightarrow q_{h_1}$ □

Separating conjunction and separating implication form an adjunction that is analogous to that between ordinary conjunctiona and implication:

Lemma 6.1.1 (Galois)

$$((p * q) \Rightarrow r)_{hp \cup h} \ \textbf{iff}\ \ p \Rightarrow (q -\!\!* r) \qquad \textbf{if}\ \ \mathrm{dom}\, h \cap \mathrm{dom}\, hp = \emptyset$$ □

The heaplet $x \mapsto v$ asserts that the heap is a singleton map:

Definition 6.1.7 (Heaplet) $x \mapsto_{hp} v \ \widehat{=}\ hp = \{x \mapsto v\}$ □

In practice, we drop the subscript and write simply $x \mapsto v$. If we do not care what the value is on the heap, then we write $x \mapsto \_$.

Now we return to verifying an assignment statement. Here is the rule in separation Hoare logic for assignment:

$$\dfrac{}{\{\, (x \mapsto \_) * ((x \mapsto e) -\!\!* p) \,\}\ x := e\ \{\, p \,\}}$$

[1] The hints on each proof step are either definitions or quote the use of Gentzen sequent calculus inference rules. Thus, I is the identity rule, $\exists$-R is existential-right, $\Rightarrow$-L is implication-left, and so on.

$$
\begin{array}{cl}
\vdash & \\ \cline{1-1}
\begin{array}{c} s_{h_2}, hp' \circledast (h_1', h_2), r_{h_1'} \\ \vdash hp' \circledast (h_1', h_2) \wedge r_{h_1'} \wedge s_{h_2} \end{array} & [\mathrm{I}] \\ \cline{1-1}
\begin{array}{c} s_{h_2}, hp' \circledast (h_1', h_2), r_{h_1'} \\ \vdash \exists\, h_1, h_2 \bullet hp' \circledast (h_1, h_2) \wedge r_{h_1} \wedge s_{h_2} \end{array} & [\exists\text{-R}] \\ \cline{1-1}
\begin{array}{c} s_{h_2}, hp' \circledast (h_1', h_2), r_{h_1'} \\ \vdash (r * s)_{hp'} \end{array} & [*\text{-def}] \\ \cline{1-1}
\begin{array}{c} p_{h_1}, s_{h_2}, hp' \circledast (h_1', h_2), Q_{h_1}^{h_1'}, p_{h_1} \wedge Q_{h_1}^{h_1'} \Rightarrow r_{h_1'} \\ \vdash (r * s)_{hp'} \end{array} & [\Rightarrow\text{-L, thin-L}] \\ \cline{1-1}
\begin{array}{c} p_{h_1}, s_{h_2}, hp' \circledast (h_1', h_2), Q_{h_1}^{h_1'}, [\, p_{hp} \wedge Q_{hp}^{hp'} \Rightarrow r_{hp'} \,] \\ \vdash (r * s)_{hp'} \end{array} & [\forall\text{-L}] \\ \cline{1-1}
\begin{array}{c} \{\, p_{hp} \,\}\ Q_{hp}^{hp'}\ \{\, r_{hp} \,\}, p_{h_1}, s_{h_2}, hp' \circledast (h_1', h_2), Q_{h_1}^{h_1'} \\ \vdash (r * s)_{hp'} \end{array} & [\text{Hoare-def}] \\ \cline{1-1}
\begin{array}{c} \{\, p_{hp} \,\}\ Q_{hp}^{hp'}\ \{\, r_{hp} \,\}, p_{h_1}, s_{h_2}, \exists\, h_1' \bullet hp' \circledast (h_1', h_2) \wedge Q_{h_1}^{h_1'} \\ \vdash (r * s)_{hp'} \end{array} & [\exists\text{-L}, \wedge\text{-L}] \\ \cline{1-1}
\begin{array}{c} \{\, p_{hp} \,\}\ Q_{hp}^{hp'}\ \{\, r_{hp} \,\}, hp \circledast (h_1, h_2), p_{h_1}, s_{h_2}, Q_{hp}^{hp'} \\ \vdash (r * s)_{hp'} \end{array} & [Q_{hp}^{hp'} \text{ is } \textbf{SL2}] \\ \cline{1-1}
\begin{array}{c} \{\, p_{hp} \,\}\ Q_{hp}^{hp'}\ \{\, r_{hp} \,\}, \exists\, h_1, h_2 \bullet hp \circledast (h_1, h_2) \wedge p_{h_1} \wedge s_{h_2}, Q_{hp}^{hp'} \\ \vdash (r * s)_{hp'} \end{array} & [\exists\text{-L}, \wedge\text{-L}] \\ \cline{1-1}
\begin{array}{c} \{\, p_{hp} \,\}\ Q_{hp}^{hp'}\ \{\, r_{hp} \,\}, (p * s)_{hp}, Q_{hp}^{hp'} \\ \vdash (r * s)_{hp'} \end{array} & [*\text{-def}] \\ \cline{1-1}
\begin{array}{c} \{\, p_{hp} \,\}\ Q_{hp}^{hp'}\ \{\, r_{hp} \,\} \\ \vdash [\, (p * s)_{hp} \wedge Q_{hp}^{hp'} \Rightarrow (r * s)_{hp'} \,] \end{array} & [\forall\text{-R}, \Rightarrow\text{-R}, \wedge\text{-L}] \\ \cline{1-1}
\begin{array}{c} \{\, p_{hp} \,\}\ Q_{hp}^{hp'}\ \{\, r_{hp} \,\} \\ \vdash \{\, (p * s)_{hp} \,\}\ Q_{hp}^{hp'}\ \{\, (r * s)_{hp} \,\} \end{array} & [\text{Hoare-def}]
\end{array}
$$

Figure 6.1: Proof of the frame rule

An example helps to explain the use of the rule. For our assignment $[x] := 10$, a suitable postcondition is: $x \mapsto 10 * y \mapsto 0$, which gives us a precondition of

$$(x \mapsto \_) * ((x \mapsto 10) -\!\!* (x \mapsto 10 * y \mapsto 0))$$

A sufficient condition is that $y \mapsto 0$, which follows directly from Lemma 6.1.1.

Further healthiness conditions are needed for a complete treatment of separation logic; in particular, heaps must be internally consistent for successful evaluation of heap variables.

6.2 Conclusions

We have described the basis of separation logic in UTP. The next step is to prove that the basic nondeterministic imperative programming language in CML is a closure with respect to the healthiness conditions in our theory. Following that, we need to prove similar results for the theory of object orientation.

The Frame Rule (Theorem 6.1.1) is an important enabler for modular reasoning about processes using reference semantics. Future work will involve studying verification patterns for the object-oriented VDM descriptions in CML processes. At present, we regard references as encapsulated within processes, as with all state, but we will investigate whether modelling examples may profit from a notion of global state and passing references through channels between processes.

As well as providing a reference semantics for CML, our theory of object orientation opens up another interesting possibility in contributing to a theory of process mobility for CML. There is a duality between mobile processes and mobile channels, and both could be described using references; the former could be a higher-order theory involving references to processes; the latter could be a simpler first-order theory involving references to channel ends. We intend to pursue this in our future work on mobile CML. The Frame Rule would allow interesting patterns of modular reasoning about networks of processes with mobile channels, and this needs to be investigated further.

Finally, the interpretation of separation logic as dealing with resources rather than heap locations could be more widely useful in an SoS context.

Bibliography

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

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

[3] Jeremy Bryans, Samual Canham, Ana Cavalcanti, Augusto Sampiao, Thiago Santos, and Jim Woodcock. CML Definition 2. Public Document. Deliverable Number: D23.3, Version: 1.0, COMPASS Project, University of York, March 2013.

[4] Jeremy Bryans, Samual Canham, and Jim Woodcock. CML Definition . Public Document. Deliverable Number: D23.4, Version: 1.0, COMPASS Project, University of York, March 2013.

[5] Jeremy Bryans, Samual Canham, and Jim Woodcock. CML Definition 3. Public Document. Deliverable Number: D23.4, Version: 1.0, COMPASS Project, University of York, September 2013.

[6] Jeremy Bryans, Andy Galloway, and Jim Woodcock. CML Definition 1. Public Document. Deliverable Number: D23.2, Version: 1.0, COMPASS Project, University of York, Sept 2012.

[7] 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.

[8] Andrew Butterfield, Pawel Gancarski, and Jim Woodcock. State visibility and communication in unifying theories of programming. *Theoretical Aspects of Software Engineering*, 0:47–54, 2009.

[9] 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.

[10] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31(3):25–37, May 2006.

[11] Joey Coleman. Second release of the compass tool: Tool grammar reference. Public Document. Deliverable Number: D31.2c, COMPASS Project, 2013.

[12] Joey Coleman. Third release of the compass tool: Cml grammar reference. Public Document. Deliverable Number: D31.3c, COMPASS Project, November 2013.

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

[14] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.

[15] 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.

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

[17] *Circus* homepage. `http://www.cs.york.ac.uk/circus/`, accessed March 28, 2014.

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

[19] Peter Gorm Larsen and Wieslaw Pawlowski. The Formal Semantics of ISO VDM-SL, 1995.

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

[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] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10–13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[24] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for Circus. *Formal Aspects of Computing*, **21**(1):3 – 32, 2007.

[25] 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.

[26] Joël Ouaknine. Discrete analysis of continuous behaviour in real-time concurrent systems. D.phil thesis, Oxford University, 2000.

[27] Joel Ouaknine. Discrete analysis of continuous behaviour in real-time concurrent systems. Technical Report PRG-RR-01-06, Oxford University, 2001. PhD. thesis.

[28] Joël Ouaknine. Digitisation and full abstraction for dense-time model checking. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2002.

[29] 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.

[30] Hilary A. Priestley. Ordered sets and complete lattices. In Roland Carl Backhouse, Roy L. Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 21–78. Springer, 2000.

[31] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22–25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

[32] A.W. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[33] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[34] Adnan Sherif. *A Framework for Specification and Validation of Real-Time Systems using Circus Actions*. PhD thesis, Center of Informatics - Federal University of Pernambuco, Brazil, 2006.

[35] J. Michael Spivey. *The Z Notation: A Reference Manual*. Series in Computer Science. Prentice Hall International, 2nd edition, 1992.

[36] Kun Wei, Jim Woodcock, and Alan Burns. Timed Circus: Timed CSP with the Miracle. In *ICECCS*, pages 55–64, 2011.

[37] 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.

[38] Jim Woodcock and Jim Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[39] Jim Woodcock and Alvaro Miyazawa. CML Definition 0. Public Document. Deliverable Number: D23.1, Version: 1.0, COMPASS Project, University of York, June 2012.

[40] 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.

[41] 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

Additional Operators

Chapter 4 has given a semantic interpretation of the reactive subset of *CML,* including imperative and sequential processes. This appendix shows how that treatment may be extended to cover (i) expressions and type operators (Section A.1), (ii) the remaining operators (either in terms of definitions in operators we have already defined, or directly in terms of the timed trace semantics, (Section A.2 and Section A.3)), (iii) replication of actions (Section A.4) and processes (Section A.6), (iv) control statements (Section A.5), and (v) parameters (Section A.7).

This is a notational guide; it is not intended to give a complete definition of the language. The purpose of this Chapter is to show how the semantic definition in Chapter 4 may be extended to cover more general forms of the operators, and to illustrate the meaning of operators not treated explicitly in Chapter 4.

A.1 Expressions

Expressions are the basic building blocks of the *CML* language. Their syntax is given in Deliverables 23.1 [39, Section 17] (the initial language syntax document) and 31.3c [12] (an updated version).

UTP focuses on giving meaning to the higher-level constructs of a language, such as program operators, treating expressions as a shallow embedding. The *CML* semantics under development does not need to refer explicitly to an 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 as UTP augmented with that of *CML*, which in turn originates from VDM. Intuitively, the expressions of *CML* are interpreted according to the VDM semantics [19, 25], giving rise to the alphabetised relations required by the *CML* semantics.

In *CML*, arithmetic connectives are given the obvious meaning. Propositional connectives and quantifiers are synonyms for corresponding UTP operators: *CML*'s **`not`**, **`and`**, **`or`**, `=>` and `<=>` correspond to UTP's operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\iff$. *CML*'s **`forall`** and **`exists`** are equivalent to the use of the quantifiers $\forall$ and $\exists$ in UTP.

In *CML* predicates can be used as expressions, for example, to define the value of Boolean variables. Set and sequence operators are given the straightforward interpretation in UTP.

A.1.1 Maps

CML inherits maps from VDM. A map type records a relation between elements of two types. If *Type*1 and *Type*2 are types, then new type *maptype* is created as

$$maptype = \textbf{map}\ Type1\ \textbf{to}\ Type2$$

An instance of *maptype* can be thought of as an unordered collection of pairs. The first element of each distinct pair must be unique, and the set of all first elements is called the *domain* of the map. The set of the second elements of all pairs is called the *range* of the map.

Maps can be created by *enumeration*: If x_1 and x_2 are of type *Type*1 and y_1 and y_2 are of type *Type*2 such that $x_1 = x_2 \Rightarrow y_1 = y_2$, then

$$m = \{x_1 \mapsto y_1, x_2 \mapsto y_2\}$$

describes a *map* m with type **map** *Type*1 **to** *Type*2.

We can treat a mapping as a set of pairs: the map m above would then be written as

$$m = \{(x_1, y_1), (x_2, y_2)\}$$

CML inherits map manipulation operators from VDM. The *CML* operator **dom** returns the domain of a mapping:

$$\textbf{dom}\, m = \{x \mid (x, y) \in m\}$$

The *CML* operator **rng** returns the range of a mapping. The range is made of members of *Type*2 which are mapped to by an element of **dom** m.

$$\textbf{rng}\, m = \{y \mid (x, y) \in m\}$$

The *CML* operator **inmap** declares a mapping which is *injective*: distinct elements of the domain are mapped to distinct elements of the range. If

$$m = \ \textbf{inmap}\ \ Type1\ \textbf{to}\ Type2$$

then

$$m = \ \textbf{map}\ Type1\ \textbf{to}\ Type2 \land \forall x_1, x_2 \in \ \textbf{dom}\, m \bullet x_1 \neq x_2 \Rightarrow m(x_1) \neq m(x_2)$$

CML defines mapping operators to manipulate these constructs. In the following definitions, m, n : **map** *Type*1 **to** *Type*2.

The *CML* operator **munion**, or *mapping union*, combines two mappings. It is only defined for mappings with distinct domains.

$$m\ \mathbf{munion}\ n = \{(x, y) \mid (x, y) \in m \cup n \wedge \mathbf{dom}\ m \cap \mathbf{dom}\ n = \emptyset\}$$

If the domains of the mappings overlap, then $m\ \mathbf{munion}\ n$ is undefined. In general, a map may be *overwritten* with another map using the operator ++. This operator is total. If the domains of the two mappings overlap, the mapping on the right hand side overrides the mapping on the left hand side.

$$m ++ n = \{(x, y) \mid (x, y) \in n\} \cup \{(x, y) \mid (x, y) \in m \ \wedge \ x \in \mathbf{dom}\ m \setminus \mathbf{dom}\ n\}$$

A *domain subtraction* operator is defined within *CML*. Given a set of elements s, and a map m of type **map** *Type*1 **to** *Type*2, the mapping obtained by removing a set s (of type *Type*1) from the domain of m is written as

$$s \mathrel{<\!-\!:} m$$

and defined as

$$s \mathrel{<\!-\!:} m = \{(x, y) \mid (x, y) \in m \ \wedge \ x \in (\mathbf{dom}\ m) \setminus s\}$$

The *mapping restriction* operator that *CML* provides restricts a mapping to a subset of its domain. If a mapping m is to be restricted by a set of elements s, we write

$$s \mathrel{<\!:} m$$

and this is defined as

$$s \mathrel{<\!:} m = \{(x, y) \mid (x, y) \in m \ \wedge \ x \in s \cap \mathbf{dom}\ m\}$$

Similar operators are defined for manipulating a mapping with respect to its range. The *range subtraction* operator removes all elements from an identified set from the range. It is written

$$m \mathrel{:\!-\!>} s$$

and defined as

$$m \mathrel{:\!-\!>} s = \{(x, y) \mid (x, y) \in m \ \wedge \ y \in (\mathbf{rng}\ m) \setminus s\}$$

The *range restriction* operator limits the mapping to the subset of the mapping where the range intersects with the identified set. It is written

$$m \mathrel{:\!>} s$$

and defined as

$$m \mathrel{:\!>} s = \{(x, y) \mid (x, y) \in m \ \wedge \ y \in \mathbf{rng}\ m \cap s\}$$

A.2 Non-parallel Action Constructors

The *actions* in a ***CML*** process define the reactive behaviour of the process. This section deals with the basic actions and action constructors from D23.1, Section 15. The action constructors include those that deal with flow-of-control, sequencing, choice parallelism and timed behaviour. These are discussed in detail in the following subsections.

A.2.1 Replicated Prefix

IF $P(x)$ identifies a finite set, then the complex form of action prefix

```
c!e?x:(P(x)) -> A(x)
```

can be translated into a choice of simple action prefix actions

$$\Box_{x:P(x)} \bullet (c!e.x \rightarrow A(x))$$

A.2.2 Channel renaming

Channel renaming is written as

```
A[[c<-nc]]
```

where `c` is renamed with `nc` in `A`, provided `nc` is free in `A`. It is defined as $A[nc/c]$.

A.2.3 Mutual recursion

Mutual recursion is the vector of least fixed points of a mutually-recursive system of equations.

$$\mu\, X_1..X_n \bullet < F_1(X_1..X_n)..F_n(X_1..X_n) >$$

A.3 Parallel Action Constructors

The parallel operators are shown in Table 4 of D23.1. The denotational semantics defines the semantics for a single general operator and the remainder are expressed in terms of this one operator.

The general parallel operator is defined in Section 4.1.13. It has the form:

```
A [| ns_1 | cs | ns_2 |] B
```

where $\alpha(A) = \{ns\_1, wait, ok, tt\}$ and $\alpha(B) = \{ns\_2, wait, ok, tt\}$. It behaves as `A` and `B` executed in parallel and synchronising on the set of channels in `cs`. Following this, the definitions of the derived operators are straightforward, and are shown below.

A.3.1 Interleaving with state

The interleaving operator that allows actions P and Q to write to the state of their process is written as

```
P [|| ns_1 | ns_2 ||] Q
```

Channel names are introduced within channel brackets

```
{| a,b |}
```

and the empty set of channel names is written `{||}`. The interleaving operator is defined to be

```
P [| ns_1 | {||} | ns_2 |] Q
```

A.3.2 Interleaving without state

The interleaving operator which does not allow actions A and B to write to their state variables is written as

```
A ||| B
```

and defined as

```
A [| {} | {||} | {} |] B
```

A.3.3 Generalised parallelism without state

In generalised parallelism without state, written as

A `[| cs |]` B

the actions A and B must synchronise on all events in the channel set cs. It is defined as

A `[| cs |]` $B =A$ `[| {} | cs | {} |]` B

A.4 Replicated Action Constructors

The replicated actions generalise the binary action operators over sets (or sequences in the case of sequential composition) of actions. Each takes a declaration and instantiates the action supplied, over the operator of interest, for each parameter binding admitted by the declaration. They are described in Table 2 of D23.1.

A.4.1 Replicated sequential composition

Replicated sequential composition is written in *CML* as

```
; i in seq s @ A(i)
```

The construct `s` must be a sequence, and for each `i` in the sequence `s`, the actions `A(i)` are executed in order.

The meaning of replicated sequential composition is defined using recursion over the sequence `s` using a local variable c.

$$\textbf{var}\ c, seq := s;\ \mu X \bullet \left(\begin{array}{l} c := head(seq);\ A(c);\ seq := tail(seq);\ X \\ \quad \lhd\ seq \neq \langle\rangle\ \rhd \\ \qquad \textbf{end}\ c, seq \end{array} \right)$$

A.4.2 Replicated external choice

Replicated external choice is written as

```
[] i in set s @ A(i)
```

for choice over a set `s`, or

```
[] i:type @ A(i)
```

for replicated external choice over a type `type`.

In each case its meaning is a generalisation of binary external choice. Either no observable event has occurred, in which case all components `A(i)` must be capable of delaying their activity for the elapsed time, or one of the `A(i)` components must have been chosen. If the type `type` (set `s`) is empty, the construct is equivalent to $STOP$. In the case of replication over a set, the semantics is given as a generalisation of the semantics for binary choice. The obvious modification is made for replication over a type.

$$\bigwedge_{i \in s} A(i)[idleprefix(tt')/tt'] \wedge \bigvee_{i \in s} A(i)$$

A.4.3 Replicated internal choice

Replicated internal choice over a set `s` is written as

```
|~| i in set s @ A(i)
```

and replicated internal choice over a type `type` is written as

```
|~| i:type @ A(i)
```

In each case its meaning is a generalisation of the meaning of binary internal choice, which is disjunction. The set `s` (type `type`) must not be empty. The obvious modification is made for replication over a type.

$$\bigvee_{i \in e} A(i)$$

A.4.4 Replicated interleaving

Replicated interleaving over a set `s` is written as

```
||| i in set s @ [ns(i)]A(i)
```

and replicated interleaving over a type `type` is written as

```
||| i:type @ [ns(i)]A(i)
```

In each case `ns(i)` is the name set of variables in action `A(i)`'s state.

When the set `s` (type `type`) has precisely two elements, say `1` and `2`, the operator is defined as the binary stateful form of the interleaving operator defined in Section A.3.1. For simplicity, we assume here that the set `s` is a set of contiguous natural numbers starting from 1.

```
A(1) [|| ns(1) | ns(2) ||] B(2)
```

and if

`||| i in set` $\{1 \dots j-1\}$`@[ns(i)]A(i)`

is defined for the first $j-1$ elements, then the meaning of replicated interleaving over a set of j elements is given as

`A(j) [|| ns(j) |` $\texttt{ns}_{j-1}$ `||] (||| i in set` $\{1 \dots j-1\}$`@[ns(i)]A(i))`

where $\texttt{ns}_j$ is defined as $\bigcup_{i \in 1 \dots j}$ `ns(i)`.

A.4.5 Replicated generalised parallelism

Replicated generalised parallelism over a set `s` is written as

```
[| cs |] i in set s @ [ns(i)]A(i)
```

and replicated generalised parallelism over the type `type` is written as

```
[| cs |] i:type @ [ns(i)]A(i)
```

In the binary case this operator reduces to the operator defined in Section 4.1.13. When combining a larger number of processes (using replication over a set) the definition may be recursively constructed as follows.

If

`[| cs |] i in set` $\{1 \dots j-1\}$`@[ns(i)]A(i)`

is defined for a set with $j-1$ elements, then we extend the definition to a set with j elements as

`A(j) [| ns(j) | cs |` $\texttt{ns}_{j-1}$ `|] ([| cs |] i in set` $\{1 \dots j-1\}$`@[ns(i)]A(i))`

where $\texttt{ns}_j$ is again defined as $\bigcup_{i \in 1 \dots j}$ `ns(i)`.

A.4.6 Replicated alphabetised parallelism

Replicated alphabetised parallelism over a set `s` is written as

```
|| i in set s @ [ns(i)|cs(i)]A(i)
```

and replicated alphabetised parallelism over a type `type` is written as

```
|| i: type @ [ns(i)|cs(i)]A(i)
```

In each case, the binary version of the operator is defined as in Section **??**.

To extend a definition of replicated parallelism for a set with $j-1$ elements

`|| i in set`$\{1 \ldots j-1\}$`@ [ns(i)|cs(i)]A(i)`

to one for a set with j elements, we write

`A(j)`
  `[| ns(j) | cs(j) || `$\mathtt{cs}_{j-1}$` | `$\mathtt{ns}_{j-1}$` |]`
    `(|| i in set`$\{1 \ldots j-1\}$`@ [ns(i)|cs(i)]A(i))`

where $\mathtt{cs}_k$ is defined as $\bigcup_{i\in\{1\ldots k\}}$ `cs(i)` and $\mathtt{ns}_k$ is defined as $\bigcup_{i\in 1\ldots k}$ `ns(i)`. So, for any given i the process $A(i)$ must synchronise on the set $cs(i) \cap (\bigcup_{j\neq i} cs(j))$.

A.5 Control Statements

Control statements (presented in Table 7 of D23.1) comprise guarded commands, conditionals and loop statements. The following describes how each ***CML*** construct is interpreted in UTP.

The first set of operators considered are Dijkstra's guarded commands [39, Section 15.7]. The conditional statement diverges if no guard holds, otherwise 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.

$e^{\top}$ is defined as $I\!I \lhd e \rhd \top_D$. It is used in the definitions in Section A.5.1 and Section A.5.4. Each construct may be generalised in the obvious way.

A.5.1 Nondeterministic if statement

The nondeterministic if statement is evaluated by initially evaluating all the guards `e_i`. If none are true, the statement diverges. Otherwise, one of the true guards is picked nondeterminstically and the corresponding action is executed.

The binary form of the nondeterministic if statement is written

```
if e_1 -> a_1
|  e_2 -> a_2
end
```

which means

$$(e_1{}^{\top};\ a_1 \sqcap e_2{}^{\top};\ a_2) \lhd e_1 \vee e_2 \rhd \bot$$

The generalised form, with n guards, is written

$$\sqcap_{i \in \{1..n\}} (e_i{}^{\top};\ a_i) \lhd \bigvee_{i \in \{1..n\}} e_i \rhd \bot$$

A.5.2 Nondeterministic do statement

The nondeterministic do statement terminates if all guards are false. Otherwise, an action corresponding to a true guard is executed, and the do statement is repeated.

The binary form of the nondeterministic do statement is written

```
do e_1 -> a_1
|  e_2 -> a_2
end
```

and the meaning is given as a combination of recursion and conditional.

$$\mu\, X \bullet (e_1^{\top};\ a_1 \sqcap e_2^{\top};\ a_2);\ X \lhd e_1 \vee e_2 \rhd I\!I$$

The generalised form is given as

$$\mu\, X \bullet \sqcap_{i \in \{1..n\}} (e_i{}^{\top};\ a_i);\ X \lhd \bigvee_{i \in \{1..n\}} e_i \rhd I\!I$$

A.5.3 Conditionals and case statements

Next are the conditional and cases statements, see Table A.1. They may be generalised in the obvious ways. We provide only a basic example of the cases statement to avoid the semantic treatment of patterns and pattern lists.

A.5.4 Loops

The final set of control statements concern loops, which are presented in [39, Section 15.9]. These comprise the sequence for loop, the set for loop, the index for loop, and the while loop.

Syntax	Semantic equivalent	Notes
if e **then** a	$a \lhd e \rhd \mathit{II}$	no elseif/else
if e **then** a_1 **else** a_2	$a_1 \lhd e \rhd a_2$	without elseif
if e_1 **then** a_1 **elseif** e_2 **then** a_2 **else** a_3	$a_1 \lhd e_1 \rhd (a_2 \lhd e_2 \rhd a_3)$	with else/elseif
cases e_1: (e_2) **then** a_1 (e_3) **then** a_2 ... (e_n+1) **then** a_n **others then** a_n+1 end	$a_1 \lhd (e_1 = e_2) \rhd (a_2 \lhd (e_1 = e_3) \rhd (\ \dots (a_n \lhd e_1 = e_{n+1} \rhd a_{n+1}\ )\cdots)\ )))$	Cases statement

Table A.1: Conditionals

Sequence for loop

The sequence for loop is written as below. It requires `s` to be a sequence, and it performs the action `a` for each element of `s` in order. The behaviour of `a` may depend on `x`.

```
for all x in seq s do a
```

It is defined using conditional and recursion. First, fresh variables `x` and `v` are created, and `v` is set to `s`. While `v` is not empty, `x` is set to the head of `v`, `v` is reduced to the tail of `v`, and `a` is performed. When `v` is empty, the conditional terminates and the new variables `x` and `v` are removed.

$$\textbf{var}\ x, v := s;\\ \quad \mu X \bullet ((x, v := \mathit{head}(v), \mathit{tail}(v);\ a;\ X) \lhd v \neq \langle\rangle \rhd \mathit{II});\\ \qquad \textbf{end}\ v, x$$

Set for loop

The set for loop is written as below. The action `a` is performed for every element of the set `s`. It is non-deterministic, since elements from `s` can be chosen in any order. The behaviour of `a` may depend on `x`.

```
for all x in set s do a
```

The structure of the definition is very similar to the structure of the sequence for loop. The new variable `T` is defined initially to be the set `s`. An element `x` is selected from the set `T` and removed. The action `a` is performed using `x`. This is continued until `T` is empty, at which point the construct terminates and the new variables are removed.

$$\textbf{var}\ x, T := s;\\ \quad \mu X \bullet ((\textstyle\bigsqcap_{x \in T} a;\ T := T \setminus \{x\};\ X) \lhd T \neq \emptyset \rhd \mathit{II});\\ \qquad \textbf{end}\ T, x$$

Index for loop

The index for loop steps through a range in user-defined increments. It is written as below. The behaviour of `a` may depend on `i`.

```
for i = e1 to e2 by e3 do a
```

The variable `c` takes the value `e1` initially, and performs the action `a`, increments `c` and recursively calls the conditional. If `c` is greater than `e2` the construct terminates gracefully. If `c` is less than or equal to `e2` the conditional is again recursively called.

$$\textbf{var } c := e1;\ f := e2;\ step := e3;\ a;\\ \quad \mu X \bullet ((a;\ c := c + step;\ X) \lhd c \leq f \rhd I\!I);\\ \qquad \textbf{end } c, f, step$$

While loop

The while loop executes the action `a` while `e` evaluates to true. It is written as

```
while e do a
```

It is again defined as a combination of recursion and conditional.

$$\mu X \bullet ((a;\ X) \lhd e \rhd I\!I)$$

A.6 Processes

A *CML* process has an encapsulated state and a number of local definitions as well as a main action. Certain process constructors are shared with actions. The basic process operators such as sequential composition have already been defined for actions. Since state is encapsulated, replicated process operators do not allow the sharing of namespaces.

A.6.1 Replicated generalised parallelism

Replicated generalised parallelism over a set `s` is written as

```
[| cs |] i in set s @ A(i)
```

and replicated generalised parallelism over a type `type` is written as

```
[| cs |] i:type @ A(i)
```

When the set (or type) has two elements, say `x` and `y`, the above are defined as generalised parallelism:

```
A(x) [| cs |] A(y)
```

In the case of replicated generalised parallelism over a set of cardinality greater than two, replicated generalised parallelism is defined as

$\sqcap_j$`(A(j) [| cs |] ([| cs |] i in set s \ {j}@A(i)))`

Note that the operator $\sqcap_j$ is used only to perform the non-deterministic choice of an element from a set.

If the set or type has cardinality one then replicated generalised parallelism is defined as `A(i)`, where `i` is the sole element of the set or type. Replicated generalised parallelism is undefined if `s` is empty.

A.6.2 Replicated alphabetised parallelism

Replicated alphabetised parallelism over a set `s` is written as

```
|| i in set s @[cs(i)]A(i)
```

and replicated alphabetised parallelism over a type `type` is written as

```
|| i:type @[cs(i)]A(i)
```

In each case all processes synchronise on the intersection of the sets `cs(i)`, where `i` is a member of the type or an element of the set. It is defined in a similar way to the generalised parallelism operator. In the case of replication over a set with two elements, say `x` and `y`, the definition is:

```
A(cs(x))[cs(x)||cs(y)]A(cs(y))
```

and when the cardinality of `s` is greater than two, it is defined as

$\sqcap_j$`(A(j)[cs(j) ||` $\cap_{i \in e \setminus \{j\}}$`] || i:e \ {j}@A(i))`

A.6.3 Replicated interleaving

Replicated interleaving over a set is written as

```
||| i in set s @A(i)
```

and replicated interleaving over a type is written as

```
||| i:type @A(i)
```

All actions proceed independently in parallel, and there is no synchronisation. It can be defined in terms of parallelism, where the synchronisation alphabet is the empty set. In the case of replication over a set, this is

```
|| i in set s @[{}]A(j)
```

A.7 Parameters

Parameters can be used to pass variables to and from processes and actions, as well as to declare local variables to be used within processes or actions.

A.7.1 Result parameter

A result is written from a process using the parameter `res`. This is written as

```
res x : T @ P
```

The meaning is given by the lambda function

`res x : T @ P` $= \lambda\, y : var(T) \bullet (\mathbf{var}\, x;\ P;\ y := x;\ \mathbf{end}\, x)$

where the fresh variable y ranges over variables of type T, and stores the result of P on its completion.

A.7.2 Value parameter

A value can be passed to a process using the keyword `val`. This is written as

```
val x : T @ P
```

The meaning is given by the lambda function

`val x : T @ P` $= \lambda\, y : T \bullet (\mathbf{var}\, x := y;\ P;\ \mathbf{end}\, x)$

where y is free in `P` and ranges over values of the type T.

A.7.3 Value-result parameter

If the result of a process is written to the variable that was originally passed to it, we can use a `vres`.

This is written as

```
vres x : T @ P
```

The meaning is given by the lambda function

`val x : T @ P` $= \lambda\, y : var(T) \bullet (\mathbf{var}\, x := y;\ P;\ y := x;\ \mathbf{end}\, x)$

where y is free in `P` and ranges over variables of type T.

A.7.4 Block statements

The block statement enables the use of locally defined variables within actions.

It is written as

```
dcl x : T @ A
```

and defined as

`dcl x : T @ A` $= \lambda\, y : T \bullet (\mathbf{var}\, x := y;\ A;\ \mathbf{end}\, x)$

Alternatively, the value of the variable `x` may be assigned at its declaration. This is written as

```
dcl x : T := e @ A
```

and defined as

`dcl x : T := e @ A` $= (\mathbf{var}\, x := e;\ A;\ \mathbf{end}\, x)$

A.8 Summary

This Appendix provides a guide to understanding the meaning of the operators which were not covered in Chapter 4. In some cases this has been done by showing how they are derived from the core operators, and in some cases by giving them a direct semantic interpretation. A portion 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 object-orientation in order to be interpreted. Examples include method calls, parametrised and instantiated actions.