CML Definition 2 — Part 0: Overview

Deliverable Number: D23.3-0
Version: 0.2
Date: 7 March 2013
Public Document

http://www.compass-research.eu
Contributors:

Jim Woodcock, York

Editors:

Jim Woodcock, York

Reviewers:

Zoe Andrews, Newcastle
Joey W. Coleman, Aarhus
Marcel Oliveira, UFPE
Richard Payne, Newcastle
Document History

<table>
<thead>
<tr>
<th>Ver</th>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>07-03-2013</td>
<td>Jim Woodcock</td>
<td>Initial document version</td>
</tr>
<tr>
<td>0.2</td>
<td>27-03-2013</td>
<td>Jim Woodcock</td>
<td>Update following internal review</td>
</tr>
</tbody>
</table>
Overview

This is the third deliverable for the CML definition; the previous deliverables are:

- **D23.1—CML Definition 0.** This contains the initial syntactic design for CML. This is the first step in providing the sound underpinnings and effective tool support for the COMPASS Modelling Language, CML. It was produced to allow tool building for the language to get underway as soon as possible, and to get early feedback from the project’s case studies. The language definition is inherited from the following baseline languages: VDM, CSP, and Circus.

- **D23.2—CML Definition 1.** This contains the first version of the semantics of CML in Hoare & He’s Unifying Theories of Programming (UTP). This deliverable was intended for discussion and is now updated as part of the current deliverable (D23.3). It has the following sections:
  1. A summary of the relevant theories from UTP: the alphabetised relational calculus and the theory of designs.
  2. A semantics for a restricted subset of CML based on Lowe & Ouaknine’s timed testing traces semantics for CSP.
  3. An extension of the language subset to include stateful reactive processes with a novel timed reactive design semantics that covers the core CML language.
  4. Additional language features treated either as a shallow embedding of CML expression language in UTP or as derived operators.
  5. A unifying theory for monotonic partial logics with undefined constructs.
  6. A first version of the operational semantics for the kernel language.
  7. An overview of the semantics for the object-oriented features of CML.

Deliverable D23.3 contains the description of CML Definition 2. It consists of five parts:

**Revised Denotational Semantics (D23.3-1).** This part contains a revision of the semantics for CML Definition 1 that appeared in deliverable D23.2. This comprises the following chapters:

1. An introduction to Unifying Theories of Programming.
2. The UTP semantics for CML Definition 1.
3. Derived CML constructs.
4. Undefinedness.
5. Appendices containing proofs.

The language described in this chapter contains imperative constructs from VDM, concurrency and communication constructs from CSP, and time. The major updates include further proofs of the consistency of the semantics in the appendices and a revision of the derived constructs.

**Hoare Logic (D23.3-2).** This part contains a new contribution describing a formal system with a set of logical rules for reasoning about the correctness of CML processes. It provides an axiomatic semantics for CML, with a model demonstrating soundness provided by the denotational semantics. In carrying out the proof of soundness, the denotational semantics has been further revised in anticipation of CML Definition 3 (deliverable D23.4), and this new semantics is contained in an appendix to this deliverable.

**Object-Orientation Semantics (D23.3-3).** Deliverable D23.2 contained an outline of the proposed semantics for the object-oriented features of CML; here we give a complete account of the object-oriented semantics.

**Operational Semantics (D23.3-4).** Deliverable D23.2 contained an outline of the proposed operational semantics for CML; here we give a complete account of the operational semantics for the untimed features of CML Definition 1. A distinguishing feature of the semantics is that it is symbolic. This underpins some of the testing work in Theme 3. Of course, a symbolic operational semantics can always be used with constants instead of the symbols, so the symbolic operational semantics subsumes the non-symbolic one. As well as describing the transitional rules for the operational semantics, this part contains a partial proof of correctness. It also contains a substantial number of laws of the relational, design, and reactive theories of UTP. It is anticipated that this collection of laws will form part of a reference manual for reasoning about the semantics of CML within UTP.

**Appendix: Denotational Semantics for CML Definition 3 (D23.3-5)** This appendix contains a draft of the denotational semantics to be used in deliverable D23.4—CML Definition 3. This is a planned revision of the denotational semantics for CML Definition 2 contained in Part 1 (D23.3-1) of this deliverable. The Hoare logic described in Part 2 (D23.3-2) has been proved sound by hand with respect to this new semantics.
Grant Agreement: 287829
Comprehensive Modelling for Advanced Systems of Systems

CML Definition 2
Deliverable Number: D23.3-1
Version: 0.3
Date: Submission date
Public Document

http://www.compass-research.eu
Contributors:

Jeremy Bryans, Newcastle
Jim Woodcock, York

Editors:

Jeremy Bryans, Newcastle
Jim Woodcock, York

Reviewers:

Joey Coleman, Aarhus
Marcel Oliveira, UFPE
Richard Payne, Newcastle
Document History

<table>
<thead>
<tr>
<th>Ver</th>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>19-11-2012</td>
<td>Jeremy Bryans</td>
<td>Initial document version drawn from D23.2</td>
</tr>
<tr>
<td>0.2</td>
<td>26-02-2013</td>
<td>Jeremy Bryans</td>
<td>Updates to discussion of non-core constructs and proofs</td>
</tr>
<tr>
<td>0.3</td>
<td>07-03-2013</td>
<td>Jim Woodcock</td>
<td>Sundry updates</td>
</tr>
</tbody>
</table>
Abstract

This report contains the second version of the semantics of the COMPASS Modelling Language (CML) in Hoare & He’s Unifying Theories of Programming (UTP). This language has been constructed as a modelling language for systems of systems. An introduction to the syntax was given in D23.1 [33], and an initial version of the semantics in D23.2 [4]. This document extends D23.2, and is meant to be read in conjunction with D23.1, the language definition. The reader is assumed to have read that document. The language described in that document is the one considered here. We start with a summary of the relevant theories from UTP: the alphabetised relational calculus and the theory of designs. Next, we give a semantics to a restricted subset of CML. This is based on Lowe & Ouaknine’s timed testing traces semantics for CSP. Following this, we extend the language subset to include stateful reactive processes and give it a timed reactive design semantics. This is novel work. Our semantics covers the core CML language, so the correspondence between additional language features are treated either as a shallow embedding of CML expression language in UTP, or as derived operators. We then describe a unifying theory for monotonic partial logics with undefined constructs to give a sound basis for choosing a treatment of undefinedness in CML. This theory also justifies using different verification tools for proving facts about CML specifications.

A version of the operational semantics for the kernel language is contained in a separate document. The semantics for the object-oriented features of CML, is given a formal treatment in a separate document, as is a Hoare-logic for CML.
### Contents

1 Preface 8

2 Introduction 10

3 Unifying Theories of Programming 12
   3.1 Background 12
   3.2 Introduction 13
   3.3 The alphabetised relational calculus 15
   3.4 The complete lattice 19
   3.5 Designs 22
   3.6 Healthiness conditions 26
      3.6.1 \(H_1\): unpredictability 26
      3.6.2 \(H_2\): possible termination 27
      3.6.3 \(H_3\): dischargeable assumptions 28
      3.6.4 \(H_4\): feasibility 28

4 UTP Semantics for \(CML\) 29
   4.1 Timed Testing Traces 30
      4.1.1 STOP 32
      4.1.2 Prefix 33
      4.1.3 Internal Choice 34
      4.1.4 External Choice 34
      4.1.5 Parallel Composition 34
      4.1.6 Hiding 36
      4.1.7 Timeout 36
      4.1.8 Recursion 37
   4.2 Lowe & Ouaknine’s Axioms 37
      4.2.1 Well Foundedness 37
      4.2.2 Prefix Closure 37
      4.2.3 Refusals 38
      4.2.4 Timelock Freedom 38
      4.2.5 Zeno Freedom 38
      4.2.6 Time-guardedness 38
   4.3 Timed Imperative Sequential Reactive Processes 39
      4.3.1 Healthiness Conditions 40
      4.3.2 Sequential Composition 41
      4.3.3 Assignment 42
      4.3.4 STOP 42
5 Derived CML Language Constructs

5.1 Expressions ........................................ 45
5.1.1 Maps ........................................ 46
5.2 Specification and Assignment .......................... 48
5.3 Non-parallel Action Constructors ....................... 48
5.3.1 Termination, Deadlock, Chaos and Divergence ... 48
5.3.2 Delay ........................................ 49
5.3.3 Replicated Prefix ................................. 49
5.3.4 Guarded action .................................. 49
5.3.5 Untimed interrupt ................................. 49
5.3.6 Untimed timeout ................................ 50
5.3.7 Timed interrupt ................................ 50
5.3.8 The startsby operator ............................ 50
5.3.9 The endsby operator ............................. 50
5.3.10 Channel renaming ................................ 50
5.3.11 Mutual recursion ................................ 51
5.4 Parallel Action Constructors .......................... 51
5.4.1 Interleaving with state ............................ 51
5.4.2 Interleaving without state ......................... 51
5.4.3 Synchronous parallelism .......................... 52
5.4.4 Synchronous parallelism without state ........... 52
5.4.5 Alphabetised parallelism with state .............. 52
5.4.6 Alphabetised parallelism without state .......... 53
5.4.7 Generalised parallelism without state .......... 53
5.5 Replicated Action Constructors ....................... 53
5.5.1 Replicated sequential composition ............... 53
5.5.2 Replicated external choice ....................... 54
5.5.3 Replicated internal choice ....................... 54
5.5.4 Replicated interleaving ........................... 54
5.5.5 Replicated generalised parallelism ............... 55
5.5.6 Replicated alphabetised parallelism .............. 55
5.5.7 Replicated synchronous parallelism .............. 55
5.6 Control Statements ................................ 56
5.6.1 Nondeterministic if statement ................... 56
5.6.2 Nondeterministic do statement .................. 56
5.6.3 Conditionals and case statements ............... 57
5.6.4 Loops ........................................ 57
5.7 Processes .......................................... 59
5.7.1 Replicated generalised parallelism .............. 59
5.7.2 Replicated alphabetised parallelism ........................................ 59
5.7.3 Replicated synchronous parallelism ....................................... 60
5.7.4 Replicated interleaving ....................................................... 60
5.8 Parameters ............................................................................ 60
5.8.1 Result parameter ................................................................. 60
5.8.2 Value parameter ................................................................. 61
5.8.3 Value-result parameter ........................................................ 61
5.8.4 Block statements ................................................................. 61
5.9 Summary ................................................................................ 62

6 Undefinedness ........................................................................... 63
6.1 Introduction ........................................................................... 63
6.2 3-valued logic in UTP .............................................................. 64
6.2.1 Basic Sets and Constructors .................................................. 64
6.2.2 Conjunction ......................................................................... 66
6.2.3 Negation ............................................................................... 67
6.2.4 Disjunction ........................................................................... 67
6.2.5 Equality ............................................................................... 68
6.3 First-order Theories ................................................................. 69
6.3.1 Contexts for First-order Theories .......................................... 69
6.3.2 First-order Theory ................................................................. 70
6.3.3 Information-theoretic Ordering .............................................. 71
6.3.4 Strictness ............................................................................. 72
6.3.5 Definiteness .......................................................................... 73
6.3.6 Monotonicity ........................................................................ 74
6.3.7 Comparing FOTs .................................................................. 74
6.4 Specific First-order Theories ..................................................... 76
6.4.1 Strict Logic ............................................................................ 76
6.4.2 Kleene System ....................................................................... 77
6.4.3 McCarthy System ................................................................. 78
6.5 Guard Systems ........................................................................ 79
6.5.1 Validity ................................................................................ 79
6.5.2 Guards ................................................................................ 79
6.5.3 Definedness Guards .............................................................. 80
6.5.4 Guards for Definite McCarthy System ................................... 81
6.6 Summary ................................................................................ 81
6.6.1 Contribution ......................................................................... 81
6.6.2 Future work ......................................................................... 81

A Proofs ....................................................................................... 85
A.1 Well Foundedness .................................................................... 85
A.1.1 Prefix Closure ...................................................................... 89
A.1.2 Time can always pass .......................................................... 97
A.1.3 Zeno Freedom .................................................................... 120

B Proof of parallel precedence ....................................................... 127
B.1 Proof of Lemma 4.3.1 .............................................................. 138
Chapter 1

Preface

This document is COMPASS Deliverable 23.3, and produced as output to Task 2.3.1 within Work Package 23 [8]. 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 System of Systems (SoS) modelling and reasoning that will integrate existing notations and semantic foundations to cover contracts, concurrency, communication, object-orientation, time, and mobility.

This document updates Deliverable 23.2. It contains a behavioural semantic definition of the CML kernel, and an updated discussion of derived operators. It also discusses some of the issues arising around the question of undefinedness, and contains further proofs of the properties of the consistency of the semantics. A contribution on the operational semantics of CML, the semantics for object-orientation and a Hoare logic for CML are submitted separately.

In this deliverable, CML0 refers to the syntax definition for CML, found in [33, 7] CML1 refers to the semantic definitions found in Deliverable 23.2 [4]. The kernel refers to that restricted subset of CML which only contains language constructs that cannot be defined in terms of simpler constructs, and is discussed in Chapter 4. The core language further excludes imperative features – this subset is considered separately in order to explore the correspondence between CML and one of the established denotational models of CSP in the literature [19].

Inputs to this task include the work within T1.1.2 Requirements for Methods and Tools on the common requirements base. The work within T2.1.1 on Guidelines for Requirements Specification for SoS and work within T2.1.2 on Guidelines for System Architectures for SoS. This task will output to tasks within Theme 3 on tools work. Feedback from these tasks will be taken into account in subsequent deliverables.

The current document is intended to be read in conjunction with COMPASS Deliverable 23.1 “CML Definition 0”, that contains the complete initial syntactic design for CML [33], as well as Deliverable D31.2c, that contains a revised grammar. Deliverable 23.1 will be updated and re-issued once feedback has been gathered from the COMPASS work on tools underway in Theme 3.
Semantic approach  The semantic approach taken is that set out by Hoare & He in their book Unifying Theories of Programming [14]. They set out there a long-term research agenda, which has as its goal a comprehensive treatment of the relationships between all programming theories and pragmatic programming paradigms.

Review of Progress  Deliverable 23.1 contains the syntax definition for CML, referred to as $CML_0$. Deliverable 23.2 contains the initial semantics for a kernel of $CML$. It contains the contract language, explicit functionality, concurrency and communication, as well as a real-time semantics for $CML$. It also contains an initial operational semantics.

This deliverable (D23.3) is a revision of D23.2, and contains further proofs of properties and proofs of consistency of the semantics. It is submitted in conjunction with a revision of the operational semantics, as well as a semantics for object orientation and a Hoare logic for $CML$. 
Chapter 2

Introduction

CML is the COMPASS Modelling Language, the first language specifically designed for modelling and analysing systems of systems. It is based on the following baseline languages: VDM [20, 16, 9], CSP [26], and Circus [15]. The first version of the language, dubbed CML0, contains only the syntactic description of the language [33]. The main objective of Work Package 23 of the COMPASS project is to provide a complete design for CML, including integration of the baseline notation’s syntax and semantics. This will be used as the basis for the development of analysis techniques in Theme 2 and prototype tools development in Theme 3.

We describe the initial report on CML as a version for discussion [33]. The second version of the language is dubbed CML1, and was reported in D23.2 [4], which was described as being a living document. This current document (D23.3) is a further iteration, updating and extending the work of Work Package 23.

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

We describe the denotational semantics for a kernel of CML in Chapter 4. This restricted subset excludes language components that can be defined in terms of simpler constructs, which are dealt with in Chapter 5. The semantics is a combination of two complementary approaches. We give a shallow embedding of CML’s expression language in UTP; for example, sets and sequences are all part of UTP and are not further defined (issues of undefinedness are dealt with explicitly in Chapter 6.) Other constructs such as the process algebraic ones, are given as deep embeddings in UTP. In Section 4.1, we give a semantics to an even more restricted subset of CML: one in which there are no imperative features and no sequential composition. This impoverished subset is chosen as a vehicle to study the semantic domain and the meaning of the basic timed process algebraic features. It is based on Lowe & Ouaknine’s timed testing traces semantics for CSP [19], which also lacks state and sequential composition. Lowe & Ouaknine use a closed presentation for the semantics, following an established tradition. In Section 4.2, we discuss their axioms
and posit most of them as theorems of CML’s basic semantics. CML is strictly more powerful 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. An appendix contains some of the proofs of theorems corresponding to Lowe & Ouaknine’s axioms.

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

In Chapter 6 we address the issue of undefined expressions in CML. We describe a unifying theory for monotonic partial logics with undefined constructs as a sound basis for choosing a treatment of undefinedness in CML. This theory justifies using a variety of tools implementing various approaches to verification.
Chapter 3

Unifying Theories of Programming

3.1 Background

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

UTP can trace its origins back to the work on predicative programming, which was started by Hehner; see [13] 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, a 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 ([25, 35]), hardware/software co-design ([5]) and component-based systems ([34]). 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 [30].

3.2 Introduction

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

The semantic model is an alphabetised version of Tarski’s relational calculus, presented in a predicative style that is reminiscent of the schema calculus in the Z [31] 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 \( (\text{clock}) \); the number of available resources \( (\text{res}) \); a trace of the events in the life of the program \( (\text{tr}) \); a set of refused events \( (\text{ref}) \) or a flag that says whether the program is waiting for interaction with its environment \( (\text{wait}) \). The signature gives the rules for the syntax for denoting objects of the theory. Healthiness conditions identify properties that characterise the theory.

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

Example 3.2.1 (Healthiness conditions)

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

\[
B \triangleq \text{clock} \leq \text{clock'}
\]

If we add \( B \) to the description of some activity, then the variable clock describes the time observed immediately before the activity starts, whereas \( \text{clock'} \) 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 \( \text{ok} \) 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.

\[
P = (\text{ok} \Rightarrow P)
\]

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

Healthiness conditions can often be expressed in terms of a function \( \phi \) that makes a program healthy. There is no point in applying \( \phi \) twice, since we cannot make a healthy program even healthier. Therefore, \( \phi \) 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 \( \text{and}_B \triangleq \lambda X \bullet P \land B \) 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 [14]. Here, we present the general relational setting, and the transition to the theory of designs.

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

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

A **condition** \((b, c, d, \ldots, \text{true})\) 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: \(\alpha (P \land Q) = \alpha P \cup \alpha Q\). Of course, if a variable is mentioned in the alphabet of both \(P\) and \(Q\), then they are both constraining the same variable.

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

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

\[
S \subseteq P \text{ iff } [P \Rightarrow S]
\]

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

\[
\begin{align*}
x' > x \land y' = y \subseteq x' &= x + 1 \land y' = y \quad \text{[definition of \(\subseteq\)]} \\
&= [x' = x + 1 \land y' = y \Rightarrow x' > x \land y' = y] \quad \text{[universal one-point rule, twice]} \\
&= [x + 1 > x \land y = y] \quad \text{[arithmetic and reflection]} \\
&= \text{true}
\end{align*}
\]

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.

\[
P \circ b \triangleright Q \equiv (b \land P) \lor (\neg b \land Q) \quad \text{if } \alpha b \subseteq \alpha P = \alpha Q
\]

\[
\alpha (P \circ b \triangleright Q) \equiv \alpha P
\]
Informally, \( P \triangleleft b \triangleleft Q \) means \( P \) if \( b \) else \( Q \).

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

\[
\begin{align*}
L_1 & \quad P \triangleleft b \triangleleft P = P & \text{idempotence} \\
L_2 & \quad P \triangleleft b \triangleleft Q = Q \triangleleft \neg b \triangleleft P & \text{symmetry} \\
L_3 & \quad (P \triangleleft b \triangleleft Q) \triangleleft c \triangleleft R = P \triangleleft b \land c \triangleleft (Q \triangleleft c \triangleleft R) & \text{associativity} \\
L_4 & \quad P \triangleleft b \triangleleft (Q \triangleleft c \triangleleft R) = (P \triangleleft b \triangleleft Q) \triangleleft c \triangleleft (P \triangleleft b \triangleleft R) & \text{distributivity} \\
L_5 & \quad P \triangleleft \text{true} \triangleleft Q = P = Q \triangleleft \text{false} \triangleleft P & \text{unit} \\
L_6 & \quad P \triangleleft b \triangleleft (Q \triangleleft b \triangleleft R) = P \triangleleft b \triangleleft R & \text{unreachable branch} \\
L_7 & \quad P \triangleleft b \triangleleft (P \triangleleft c \triangleleft Q) = P \triangleleft b \lor c \triangleleft Q & \text{disjunction} \\
L_8 & \quad (P \triangleleft Q) \triangleleft b \triangleleft (R \triangleleft S) = (P \triangleleft b \triangleleft R) \triangleleft (Q \triangleleft b \triangleleft S) & \text{interchange}
\end{align*}
\]

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

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

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

\[
\begin{align*}
(P \triangleleft b \triangleleft (Q \triangleleft b \triangleleft R)) & \quad [L2] \\
= ((Q \triangleleft b \triangleleft R) \triangleleft \neg b \triangleleft P) & \quad [L3] \\
= (Q \triangleleft b \land \neg b \triangleleft (R \triangleleft \neg b \triangleleft P)) & \quad \text{propositional calculus} \\
= (Q \triangleleft \text{false} \triangleleft (R \triangleleft \neg b \triangleleft P)) & \quad [L5] \\
= (R \triangleleft \neg b \triangleleft P) & \quad [L2] \\
= (P \triangleleft b \triangleleft R) & \quad \square
\end{align*}
\]

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

**Law 3.3.1 (Refinement to conditional)**

\[
P \sqsubseteq (Q \triangleleft b \triangleleft R) = (P \sqsubseteq b \land Q) \land (P \sqsubseteq \neg b \land R)
\]

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

**Proof of Law 3.3.1**

\[
\begin{align*}
P & \sqsubseteq (Q \triangleleft b \triangleleft R) & \quad \text{[definition of \( \sqsubseteq \)]} \\
& = [(Q \triangleleft b \triangleleft R) \Rightarrow P] & \quad \text{[definition of conditional]} \\
& = [b \land Q \lor \neg b \land R \Rightarrow P] & \quad \text{[propositional calculus]} \\
& = [b \land Q \Rightarrow P] \land [\neg b \land R \Rightarrow P] & \quad \text{[definition of \( \sqsubseteq \), twice]}
\end{align*}
\]
A compositional argument is also available for conjunctions.

**Law 3.3.2 (Separation of requirements)**

\[(P \land Q) \subseteq R = (P \subseteq R) \land (Q \subseteq R)\]

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.

\[
P(v') ; Q(v) \equiv \exists v_0 \cdot P(v_0) \land Q(v_0) \quad \text{if } outP = inQ' = \{v'\}
\]

\[
\text{in}(P(v') ; Q(v)) \equiv \text{in}P
\]

\[
\text{out}(P(v') ; Q(v)) \equiv \text{out}Q
\]

Composition is associative and distributes backwards through the conditional.

\[
L1 \quad P ; (Q ; R) = (P ; Q) ; R \quad \text{associativity}
\]

\[
L2 \quad (P \bowtie b \bowtie Q) ; R = ((P ; R) \bowtie b \bowtie (Q ; R)) \quad \text{left distribution}
\]

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 \(\alpha e\) 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.

\[
x :=_A e \equiv (x' = e \land y = y' \land \cdots \land z' = z)
\]

\[
\alpha(x :=_A e) \equiv A \cup A'
\]

There is a degenerate form of assignment that changes no variable; it’s called “skip”, and has the following definition.

\[
\Pi_A \equiv (v' = v) \quad \text{if } A = \{v\}
\]

\[
\alpha \Pi_A \equiv A \cup A'
\]

Skip is the identity of sequence.

\[
L5 \quad P ; \Pi_{\alpha P} = P = \Pi_{\alpha P} ; P \quad \text{unit}
\]

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

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

\[ P \sqcap Q \equiv P \lor Q \quad \text{if } \alpha P = \alpha Q \]

\[ \alpha(P \sqcap Q) \equiv \alpha P \]

The alphabet must be the same for both arguments.

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

**Law 3.3.3 (Refinement and nondeterminism)**

\[ P \sqsubseteq Q = (P \sqcap Q = P) \]

**Proof**

\[
\begin{align*}
P \sqcap Q &= P \\
&= (P \sqcap Q \sqsubseteq P) \land (P \sqsubseteq P \sqcap Q) \quad \text{[antisymmetry]} \\
&= [P \Rightarrow P \sqcap Q] \land [P \sqcap Q \Rightarrow P] \quad \text{[definition of \( \sqsubseteq \), twice]} \\
&= [P \Rightarrow P \lor Q] \land [P \lor Q \Rightarrow P] \quad \text{[definition of \( \lor \), twice]} \\
&= \text{true} \land [P \lor Q \Rightarrow P] \quad \text{[propositional calculus]} \\
&= [Q \Rightarrow P] \quad \text{[propositional calculus]} \\
&= P \sqsubseteq Q \quad \text{[definition of \( \sqsubseteq \)]}
\end{align*}
\]

Another fundamental result is that reducing nondeterminism leads to refinement.

**Law 3.3.4 (Thin nondeterminism)**

\[ P \sqcap Q \sqsubseteq P \]

The proof is immediate from properties of the propositional calculus.

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

\[
\begin{align*}
\texttt{var} \ x &\triangleq (\exists x \cdot \Pi_A) \quad \alpha(\texttt{var} \ x) \triangleq A \setminus \{x\} \\
\texttt{end} \ x &\triangleq (\exists x' \cdot \Pi_A) \quad \alpha(\texttt{end} \ x) \triangleq A \setminus \{x'\}
\end{align*}
\]

The relation \texttt{var} \( x \) is not homogeneous, since it does not include \( x \) in its alphabet, but it does include \( x' \); similarly, \texttt{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.

\[
\begin{align*}
(\texttt{var} \ x ; Q) &= (\exists x \cdot Q) \\
(Q ; \texttt{end} \ x) &= (\exists x' \cdot Q)
\end{align*}
\]
More interestingly, we can use \texttt{var} $x$ and \texttt{end} $x$ to specify a variable block.

$$(\texttt{var} \ x; \ Q; \ \texttt{end} \ x) = (\exists x, x' \cdot Q)$$

In programs, we use \texttt{var} $x$ and \texttt{end} $x$ paired in this way, but the separation is useful for reasoning.

The following laws are representative.

$$L6 \quad (\texttt{var} \ x; \ \texttt{end} \ x) = \mathbb{I}$$

$$L8 \quad (x := e; \ \texttt{end} \ x) = (\texttt{end} \ x)$$

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

$$(\texttt{var} \ x; \ x := 2 \ast y; \ w := 0; \ \texttt{end} \ x)$$

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: \textit{alphabet extension}. If the right-hand assignment is $P \equiv w :=_A 0$, then the left-hand assignment is denoted by $P_{+x}$.

$$P_{+x} \equiv P \land x' = x$$

$$\alpha(P_{+x}) \equiv \alpha P \cup \{x, x'\}$$

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

$$L1 \quad \texttt{var} \ x; \ Q_{+x} ; \ \texttt{end} \ x = Q ; \ \texttt{var} \ x ; \ P ; \ \texttt{end} \ x$$

$$L2 \quad \texttt{var} \ x; \ P; \ Q_{+x} ; \ \texttt{end} \ x = \texttt{var} \ x; \ P; \ \texttt{end} \ x; \ Q$$

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.

3.4 The complete lattice

The refinement ordering is a partial order: reflexive, anti-symmetric, and transitive. Moreover, the set of alphabetised predicates with a particular alphabet $A$ is a complete lattice under the refinement ordering. Its bottom element is denoted $\bot_A$, and is the weakest predicate \textit{true}; this is the program that aborts, and behaves quite arbitrarily. The top element is denoted $\top_A$, and is the strongest predicate \textit{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 $A$.

$$L1 \quad \bot_A \sqsubseteq P$$

\textit{bottom 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.

\[
\begin{align*}
L1 & \quad P \subseteq (\bigcap S) \iff (P \subseteq X \text{ for all } X \in S) \\
L1A & \quad (\bigcap S) \subseteq X \text{ for all } X \in S \quad \text{lower bound} \\
L1B & \quad \text{if } P \subseteq X \text{ for all } X \in S, \text{ then } P \subseteq (\bigcap S) \quad \text{greatest lower bound}
\end{align*}
\]

These laws characterise basic properties of least upper bounds.

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

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

\[
\begin{align*}
P \odot R &= (P \cap Q) \odot R \\
&= (P \odot R) \cap (Q \odot R) \\
&\subseteq Q \odot R
\end{align*}
\]

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

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

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

\[
\mu F \triangleq \bigcap \{ X \mid F(X) \subseteq X \}
\]

The strongest fixed-point \( \nu F \) is the dual of the weakest fixed-point.

Hoare & He use weakest fixed-points to define recursion. They write a recursive program as \( \mu X \cdot C(X) \), where \( C(X) \) 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 \( C \) stand for recursive calls to \( C \) itself. The definition of recursion is as follows.

\[
\mu X \cdot C(X) \triangleq \mu F \quad \text{where } F \triangleq \lambda X \cdot C(X)
\]

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

\[
L1 \quad \mu F \subseteq Y \text{ if } F(Y) \subseteq Y \quad \text{weakest fixed-point}
\]
$L2 \quad [F(\mu F) = \mu F]$  

$L1$ 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 $L1$**

\[
F(Y) \subseteq Y \\
= Y \in \{ X \mid F(X) \subseteq X \} \\
\Rightarrow \bigcap \{ X \mid F(X) \subseteq X \} \subseteq Y \\
= \mu F \subseteq Y
\]

**Proof of $L2$**

\[
\mu F = F(\mu F) \\
= \mu F \subseteq F(\mu F) \land F(\mu F) \subseteq \mu F \\
\iff F(F(\mu F)) \subseteq F(\mu F) \land F(\mu F) \subseteq \mu F \\
\iff F(\mu F) \subseteq \mu F \\
= F(\mu F) \subseteq \bigcap \{ X \mid F(X) \subseteq X \} \\
\iff \forall X \in \{ X \mid F(X) \subseteq X \} \cdot F(\mu f) \subseteq X \\
= \forall X \cdot F(X) \subseteq X \Rightarrow F(\mu F) \subseteq X \\
\iff \forall X \cdot F(X) \subseteq X \Rightarrow F(\mu F) \subseteq F(X) \\
\iff \forall X \cdot F(X) \subseteq X \Rightarrow \mu F \subseteq X \\
= true
\]

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

\[
b \ast P \triangleq \mu X \bullet ( (P ; X) \ast b \ast \bot )
\]

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

\[
\mu X \bullet X \\
= \bigcap \{ Y \mid (\lambda X \bullet X)(Y) \subseteq Y \} \\
= \bigcap \{ Y \mid Y \subseteq Y \} \\
= \bigcap \{ Y \mid true \} \\
= \bot
\]

A surprising, but simple, consequence of Example 3.4.2 is that a program can recover from a non-terminating loop!
Example 3.4.3 (Aborting loop) Suppose that the sole state variable is \( x \) and that \( c \) is a constant.

\[
(b \ast P); \ x := c \quad \begin{array}{l} \text{[Example 3.4.2]} \\
= \bot; \ x := c \quad \begin{array}{l} \text{[definition of } \bot \text{]} \\
= \text{true}; \ x := c \quad \begin{array}{l} \text{[definition of assignment]} \\
= \exists x_0 \bullet \text{true} \land x' = c \quad \begin{array}{l} \text{[definition of composition]} \\
= x' = c \quad \begin{array}{l} \text{[definition of assignment]} \\
= x := c \quad \begin{array}{l} \text{[predicate calculus]} \\
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\]

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

3.5 Designs

The problem pointed out in Section 3.4 can be explained as the failure of general alphabetised predicates \( P \) to satisfy the equation below.

\[
\text{true} \ ; \ P = \text{true}
\]

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

The solution is to consider a subset of the alphabetised predicates in which a particular observational variable, called \( \text{ok} \), 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 \( \text{false} \) cannot possibly belong to this set, since \( \text{false} = \text{false} \ ; \text{true} \).

The predicates in this set are called designs. They can be split into precondition-postcondition pairs, and are in the same spirit as specification statements used in refinement calculi. As such, they are a basis for unifying languages and methods like B [1], VDM [16], Z [31], and refinement calculi [21, 3, 22].

In designs, \( \text{ok} \) records that the program has started, and \( \text{ok}' \) 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 \( \text{ok} \) or \( \text{ok}' \), is written \( (P \vdash Q) \). It is defined as follows.

\[
(P \vdash Q) \equiv (\text{ok} \land P \Rightarrow \text{ok}' \land Q)
\]
If the program starts in a state satisfying $P$, then it will terminate, and on termination $Q$ will be true.

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

Example 3.5.1 (Abort)

\[
false \vdash false \quad \text{[definition of design]}
\]

\[
= ok \land false \Rightarrow ok' \land false \quad \text{[false zero for conjunction]}
\]

\[
= false \Rightarrow ok' \land false \quad \text{[vacuous implication]}
\]

\[
= true \quad \text{[vacuous implication]}
\]

\[
= false \Rightarrow ok' \land true \quad \text{[false zero for conjunction]}
\]

\[
= ok \land false \Rightarrow ok' \land true \quad \text{[definition of design]}
\]

\[
= false \vdash true \quad \square
\]

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

Example 3.5.2 (Miracle)

\[
true \vdash false \quad \text{[definition of design]}
\]

\[
= ok \land true \Rightarrow ok' \land false \quad \text{[true unit for conjunction]}
\]

\[
= ok \Rightarrow false \quad \text{[contradiction]}
\]

\[
= \neg ok \quad \square
\]

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

Law 3.5.1 Refinement of designs

\[
P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2 = [P_1 \land Q_2 \Rightarrow Q_1] \land [P_1 \Rightarrow P_2] \quad \square
\]

Proof

\[
P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2 \quad \text{[definition of $\sqsubseteq$]}
\]

\[
= [ (P_2 \vdash Q_2) \Rightarrow (P_1 \vdash Q_1) ] \quad \text{[definition of design, twice]}
\]

\[
= [ (ok \land P_2 \Rightarrow ok' \land Q_2) \Rightarrow (ok \land P_1 \Rightarrow ok' \land Q_1) ] \quad \text{[case analysis on $ok$]}
\]

\[
= [ (P_2 \Rightarrow ok' \land Q_2) \Rightarrow (P_1 \Rightarrow ok' \land Q_1) ] \quad \text{[case analysis on $ok'$]}
\]

\[
= [ (P_2 \Rightarrow Q_2) \Rightarrow (P_1 \Rightarrow Q_1) ] \land (\neg P_2 \Rightarrow \neg P_1) \quad \text{[propositional calculus]}
\]

\[
= [ (P_2 \Rightarrow Q_2) \Rightarrow (P_1 \Rightarrow Q_1) ] \land (P_1 \Rightarrow P_2) \quad \text{[predicate calculus]}
\]

\[
= [P_1 \land Q_2 \Rightarrow Q_1] \land [P_1 \Rightarrow P_2] \quad \square
\]

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.

\[
L1 \quad true ; (P \vdash Q) = true \quad \text{left-zero}
\]
Proof

\[ \text{true} : (P \vdash Q) \]  \hspace{1cm} \text{[property of sequential composition]}  
\[ = \exists \, \text{ok}_0 \cdot \text{true} : (P \vdash Q)[\text{ok}_0/\text{ok}] \]  \hspace{1cm} \text{[case analysis]}  
\[ = (\text{true} ; (P \vdash Q) [\text{true}/\text{ok}]) \lor (\text{true} ; (P \vdash Q) [\text{false}/\text{ok}]) \]  \hspace{1cm} \text{[property of design]}  
\[ = (\text{true} ; (P \vdash Q) [\text{true}/\text{ok}]) \lor (\text{true} ; \text{true}) \]  \hspace{1cm} \text{[relational calculus]}  
\[ = (\text{true} ; (P \vdash Q) [\text{true}/\text{ok}]) \lor \text{true} \]  \hspace{1cm} \text{[propositional calculus]}  
\[ = \text{true} \]  \hspace{1cm} \square \]

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

\[ (x := e) \triangleq (\text{true} \vdash x = e \land y = y \land \cdots \land z = z) \]
\[ \Pi_d \triangleq (\text{true} \vdash \Pi) \]

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

\[ L2 \quad (v := e ; v := f(v)) = (v := f(e)) \]
\[ L3 \quad (v := e ; (P \triangleleft b(v) \triangleright Q)) = ((v := e ; P) \triangleleft b(e) \triangleright (v := e ; Q)) \]
\[ L4 \quad (\Pi_d ; (P \vdash Q)) = (P \vdash Q) \]

As as an example, we present the proof of \( L2 \).

Proof of \( L2 \)

\[ v := e ; v := f(v) \]  \hspace{1cm} \text{[definition of assignment, twice]}  
\[ = (\text{true} \vdash v' = e) ; (\text{true} \vdash v' = f(v)) \]  \hspace{1cm} \text{[case analysis on ok_0]}  
\[ = ((\text{true} \vdash v' = e)[\text{true}/\text{ok}'] ; (\text{true} \vdash v' = f(v))[\text{true}/\text{ok}]) \lor \neg \text{ok} ; \text{true} \]  \hspace{1cm} \text{[definition of design]}  
\[ = ((\text{ok} \Rightarrow v' = e) ; (\text{ok}' \land v' = f(v))) \lor \neg \text{ok} \]  \hspace{1cm} \text{[relational calculus]}  
\[ = \text{ok} \Rightarrow (v' = e ; (\text{ok}' \land v' = f(v))) \]  \hspace{1cm} \text{[assignment composition]}  
\[ = \text{ok} \Rightarrow \text{ok}' \land v' = f(e) \]  \hspace{1cm} \text{[definition of design]}  
\[ = (\text{true} \vdash v' = f(e)) \]  \hspace{1cm} \text{[definition of assignment]}  
\[ = v := f(e) \]  \hspace{1cm} \square \]

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.

\[ T1 \quad ((P_1 \vdash Q_1) \cap (P_2 \vdash Q_2)) = (P_1 \land P_2 \vdash Q_1 \lor Q_2) \]
If the choice between two designs depends on a condition \( b \), then so do the precondition and the postcondition of the resulting design.

\[
T2 \quad ((P_1 \vdash Q_1 \prec b 
\succ (P_2 \vdash Q_2)) = ((P_1 \prec b \succ P_2) \vdash (Q_1 \prec b 
\succ Q_2))
\]

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.

\[
T3 \quad ((P_1 \vdash Q_1) ; (P_2 \vdash Q_2)) = ((\neg (\neg P_1 ; \mathbf{true}) \land (Q_1 \mathbf{wp} P_2)) \vdash (Q_1 ; Q_2))
\]

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

\[
Q \mathbf{wp} P = \neg (Q ; \neg P)
\]

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.

\[
T3' \quad ((p_1 \vdash Q_1) ; (p_2 \vdash Q_2)) = (p_1 \land (Q_1 \mathbf{wp} P_2)) \vdash (Q_1 ; Q_2))
\]

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.

\[
T4 \quad (\mu X, Y \cdot (F(X, Y) \vdash G(X, Y))) = (P(Q) \vdash Q)
\]

where \( P(Y) = (\nu X \cdot F(X, Y)) \) and \( Q = (\mu Y \cdot P(Y) \Rightarrow G(P(Y), Y)) \)

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).

\[
\top_D = (\mathbf{true} \vdash \mathbf{false}) = \neg \mathbf{ok}
\]

\[
\bot_D = (\mathbf{false} \vdash \mathbf{true}) = \mathbf{true}
\]

The least upper bound and the greatest lower bound are established in the following theorem.
Theorem 3.5.1  *Meets and joins*

\[
\bigcap_i (P_i \vdash Q_i) = (\bigwedge_i P_i) \vdash (\bigvee_i Q_i) \\
\bigcup_i (P_i \vdash Q_i) = (\bigvee_i P_i) \vdash (\bigwedge_i P_i \Rightarrow Q_i)
\]

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

### 3.6 Healthiness conditions

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

#### 3.6.1 \( H1 \): unpredictability

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

\[
\text{true} ; R = \text{true} \quad \text{and} \quad \Pi_0 ; R = R
\]

We now present a proof of these results.

**Designs with left-units and left-zeros are \( H1 \)**

\[
R \\
= \Pi_0 ; R \\
= (\text{true} \vdash \Pi_0) ; R \\
= (ok \Rightarrow ok' \land \Pi_0) ; R \\
= (\neg ok ; R) \lor (\Pi ; R) \\
= (\neg ok ; \text{true} ; R) \lor (\Pi ; R) \\
= \neg ok \lor (\Pi ; R) \\
= \neg ok \lor R \\
= ok \Rightarrow R
\]

[assumption (\( \Pi_0 \) is left-unit)]

[\( \Pi_0 \) definition]

[design definition]

[relational calculus]

[relational calculus]

[assumption (\( \text{true} \) is left-zero)]

[assumption (\( \Pi \) is left-unit)]

[relational calculus]
**H1** designs have a left-zero

\[
\begin{align*}
\text{true } & ; R \\
= \text{true } & ( ok \Rightarrow R ) \quad \text{[assumption (R is H1)]} \\
= ( \text{true } & ; \neg ok ) \lor ( \text{true } & ; R ) \quad \text{[relational calculus]} \\
= \text{true } & \lor ( \text{true } & ; R ) \quad \text{[relational calculus]} \\
= \text{true } & \\
\end{align*}
\]

**H1** designs have a left-unit

\[
\begin{align*}
\sqsubseteq_o & ; R \\
= ( \text{true } & \sqsubseteq_o ) ; R \quad \text{[definition of } \sqsubseteq_o \text{]} \\
= ( ok \Rightarrow ok' \land \sqsubseteq_o ) ; R \quad \text{[definition of design]} \\
= ( \neg ok \lor ( ok \land R ) \quad \text{[relational calculus]} \\
= ( \neg ok ; \text{true } & ; R ) \lor ( ok \land R ) \quad \text{[true is left-zero]} \\
= \neg ok \lor ( ok \land R ) \quad \text{[relational calculus]} \\
= ok \Rightarrow R \quad \text{[R is H1]} \\
= R \quad \text{[design definition]}
\end{align*}
\]

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

### 3.6.2 **H2**: possible termination

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

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

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

\[
\begin{align*}
R & \\
= ok \Rightarrow R \quad \text{[assumption (R is H1)]} \\
= ok \Rightarrow ( \neg ok' \land R^f ) \lor ( ok' \land R^t ) \quad \text{[assumption (R is H2)]} \\
= ok \Rightarrow ( \neg ok' \land R^f \land R^t ) \lor ( ok' \land R^t ) \quad \text{[propositional calculus]} \\
= ok \Rightarrow ( ( ( \neg ok' \land R^f \lor ok' ) \land R^t ) \quad \text{[propositional calculus]} \\
= ok \Rightarrow ( ( R^f \lor ok' ) \land R^t ) \quad \text{[propositional calculus]} \\
= ok \Rightarrow ( R^f \land ok' \land R^t ) \quad \text{[assumption (R is H2)]} \\
= ok \Rightarrow R^t \lor ( ok' \land R^t ) \quad \text{[propositional calculus]} \\
= ok \land \neg R^f \Rightarrow ok' \land R^t \quad \text{[design definition]}
\end{align*}
\]
It is very simple to prove that designs are **H1** healthy; we present the proof that designs are **H2** healthy.

**Designs are H2**

\[
(P \vdash Q)[false/ok'] \quad \text{[definition of design]}
\]
\[
= (ok \land P \Rightarrow false) \quad \text{[propositional calculus]}
\]
\[
\Rightarrow (ok \land P \Rightarrow Q) \quad \text{[definition of design]}
\]
\[
= (P \vdash Q)[true/ok'] \quad \square
\]

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

### 3.6.3 **H3**: dischargeable assumptions

The healthiness condition **H3** is specified as an algebraic law: \( R = R ; \quad II \). 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

\[
((P \vdash Q) = (P \vdash Q) ; \quad II) \quad \text{[definition of design-skip]}
\]
\[
= ((P \vdash Q) = ((P \vdash Q) ; (true \vdash II))) \quad \text{[sequence of designs]}
\]
\[
= ((P \vdash Q) = (\neg (\neg P ; \quad true) \land \neg (Q ; \quad true) \vdash Q ; \quad II)) \quad \text{[skip unit]}
\]
\[
= ((P \vdash Q) = (\neg (\neg P ; \quad true) \vdash Q)) \quad \text{[design equality]}
\]
\[
= (\neg P = \neg P ; \quad true) \quad \text{[propositional calculus]}
\]
\[
= (P = P ; \quad true) \quad \square
\]

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

### 3.6.4 **H4**: feasibility

The final healthiness condition is also algebraic: \( R ; \quad true = true \). Using the definition of sequence, we can establish that this is equivalent to \( \exists v' \cdot R \), 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 \( \top_o \) is not **H4** healthy, since miracles are not feasible.
Chapter 4

UTP Semantics for **CML**

We give in this Chapter the semantics for **CML2**, the language of timed imperative reactive processes that combines VDM with discrete-time CSP. We focus on the kernel subset of the language that describes actions rather than process-level combinators; the latter are closely related to a subset of the former. Imperative features of **CML** are represented by assignment and specification statements. Other programming-language features are derived from more basic control structures. For example, the **while** loop is derived from a combination of recursion, conditional, and sequential composition. In this semantics, we use the notation of UTP rather than the syntax of VDM; Chapter 5 gives the correspondence between the two.

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

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

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

Consider the following CSP process: \( a \rightarrow b \rightarrow STOP \). 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, \( a \rightarrow b \rightarrow STOP \), we can conduct this kind of experiment at different points in the evolution of the process. We could, for instance, conduct it before anything has happened at all. Suppose that the set of possible events is \( \{ a, b, c \} \). If we were to offer the entire set to the process, then it could not refuse to engage in \( a \), but it could refuse both \( b \) and \( c \). If we were to make a meaner offer (that is, a subset of our original offer), say only \( \{ b, c \} \), then it would still
refuse. Here are all the refusals:

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

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

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

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

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

### 4.1 Timed Testing Traces

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

Let \( \Sigma \) be the universe of events and let the clock event be \( tock \notin \Sigma \). Our semantic domain consists of traces with embedded refusal sets. Refusal sets may only be observed at the end of each time interval. For example, the trace

\[
\langle a, b, \{b, c\}, tock, \emptyset, tock, c \rangle
\]

represents the observation:
• The trace \( \langle a, b \rangle \) occurred in the first time interval.
• At the end of this trace, the process refused the set of events \( \{ b, c \} \).
• No events were observed during the second time interval.
• The third time interval is incomplete, but the trace \( \langle c \rangle \) was observed so far.

Traces bearing this structure are drawn from the following set:

**Definition 4.1.1**

\[
\text{timedTrace} \equiv (\Sigma + \mathbb{P}(\Sigma).\text{tock})^*
\]

This definition uses a variation on the standard notation for regular expressions, where the dot is to be understood as concatenation, and + is to be understood as a choice, and \( x^* \) is the expression that describes all the finite sequences containing only \( a \). Refusal sets immediately precede \text{tock} events and these pairs are separated by sequences of events; just what we need.

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

\[
\langle \{ a, b \}, \text{tock}, \{ b \}, \text{tock}, \{ a, b \}, \text{tock}, \{ b \}, \text{tock}, \{ a, b \}, \text{tock} \rangle
\]

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

**Definition 4.1.2**

\[
\mathbf{T}_0(P) = P \land tt' \in \text{timedTrace}
\]

As a conjunctive function, \( \mathbf{T}_0 \) is a monotonic idempotent. Applying \( \mathbf{T}_0 \) gives us a way of type-checking the variable \( tt' \).

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

**Definition 4.1.3**

\[
\text{squash}(\emptyset) = \langle \rangle
\]
\[\text{squash}(f) = \langle f(\min(\text{dom } f)) \rangle \triangleleft \text{squash}(\{\min(\text{dom } f)\} \triangleleft f)\]
\[t \uparrow S = \text{squash}(t \triangleright S)\]

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

**Definition 4.1.4**

\[
\begin{align*}
\text{trace}(t) &= t \uparrow \Sigma^\triangleright \\
\text{refsduring}(t) &= \text{ran}(t \triangleright P(\Sigma)) \\
\text{refusals}(t) &= \bigcup \text{refsduring}(t)
\end{align*}
\]

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

**Lemma 4.1.1 (Trace extraction)**

\[
\begin{align*}
\text{trace}(\langle \rangle) &= \langle \rangle \\
\text{trace}(\langle a \rangle \triangleright t) &= \langle a \rangle \triangleright \text{trace}(t) \quad \text{if } a \in \Sigma^\triangleright \\
\text{trace}(\langle X \rangle \triangleright t) &= \text{trace}(t) \quad \text{if } X \in P(\Sigma)
\end{align*}
\]

We define an order relation on traces: \(s \preceq t\) holds when \(s\) contains less information than \(t\).

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

\[
\begin{align*}
\langle \rangle &\preceq u \\
\langle a \rangle \triangleright t &\preceq \langle a \rangle \triangleright u \quad \text{if } t \preceq u \\
\langle X, \triangleright \rangle \triangleright t &\preceq \langle Y, \triangleright \rangle \triangleright u \quad \text{if } t \preceq u
\end{align*}
\]

This is a stronger relation than the usual prefix relation on traces, \(\preceq\).

**Lemma 4.1.2 (Precedence traces)**

\[t \preceq u \Rightarrow \text{trace}(t) \preceq \text{trace}(u)\]

*Proof* by induction on \(t\).

A similar result holds for the refusals over testing traces:

**Lemma 4.1.3 (Precedence refusals)**

\[t \preceq u \land a \in \text{refusals}(t) \Rightarrow a \in \text{refusals}(u)\]

### 4.1.1 STOP

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

**Definition 4.1.6 (STOP testing trace)**

\[
\text{STOP} \equiv \text{T0}(\text{trace}(tt') \in \text{tock}^*)
\]

### 4.1.2 Prefix

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

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

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

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

\[
\langle \emptyset, \text{tock}, \{a\}, \text{tock}, b, c, \{a, c\}, \text{tock} \rangle
\]

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

**Definition 4.1.7 (idleprefix and idlesuffix)**

\[
\begin{align*}
\text{idleprefix}(t) &\leq t \\
\text{trace}(\text{idleprefix}(t)) &\in \text{tock}^* \\
\forall t : \text{TimedTrace} &
\begin{align*}
\text{trace}(u) &\in \text{tock}^* \land \text{trace}(u) \leq \text{trace}(t) \\
\Rightarrow &\text{trace}(u) \leq \text{idleprefix}(\text{trace}(t)) \\
\text{idlesuffix}(t) &\equiv t - \text{idleprefix}(t)
\end{align*}
\end{align*}
\]

Continuing with our second case, the \( a \)-event must be the first non-tock event in the trace: \( \text{head}((\text{idlesuffix}(tt'))) \). The event \( a \) must not be refused during the idle prefix: \( a \not\in \text{refusals}(\text{idleprefix}(tt')) \). In the example above, this does not hold.

Finally, the action will continue as \( P \) behaves: \( P[\text{tail}(\text{idlesuffix}(tt'))/tt'] \). Note that the use of \( \text{head} \) and \( \text{tail} \) are both defined, since \( \text{trace}(tt') \not\in \text{tock}^* \). All this is formalised as follows:
Definition 4.1.8 (Prefix)

\[
\begin{align*}
a \rightarrow P & \equiv \\
& \left( \begin{array}{l}
a \notin \text{refusals}(tt') \\
\langle \text{trace}(tt') \in \text{tock}^{*} \rangle \\
a = \text{head} \left( \text{trace} \left( \text{idlesuffix}(tt') \right) \right) \\
\land a \notin \text{refusals} \left( \text{idleprefix}(tt') \right) \\
\land P \left[ \text{tail} \left( \text{idlesuffix}(tt') \right) / tt' \right]
\end{array} \right)
\end{align*}
\]

4.1.3 Internal Choice

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

Definition 4.1.9 (Internal choice)

\[
P \cap Q \equiv P \lor Q
\]

4.1.4 External Choice

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

Definition 4.1.10 (External choice)

\[
P \boxdot Q \equiv (P \land Q)[\text{idleprefix}(tt')/tt'] \land (P \lor Q)
\]

The difference between internal and external choice can be seen by comparing the following two processes:

\[
\begin{align*}
a \rightarrow \text{STOP} \cap b \rightarrow \text{STOP}
\end{align*}
\]

and

\[
\begin{align*}
a \rightarrow \text{STOP} \boxdot b \rightarrow \text{STOP}
\end{align*}
\]

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

4.1.5 Parallel Composition

The parallel composition \( P \parallel Q \) specifies the set of events that require synchronisation between the two actions \( P \) and \( Q \); outside this set events happen independently, without needing the participation of the other action. Parallel composition is then a form of restricted conjunction, where each action’s behaviour is seen as a projection of the overall trace.
Definition 4.1.11 (Parallel composition)

\[ P \parallel_A Q \overset{\text{def}}{=} \exists t, u \cdot P[t/t'] \land Q[u/t'] \land tt' \in t \parallel_A u \]

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

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

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

Definition 4.1.12

\[ X \cap_A Y \overset{\text{def}}{=} \mathcal{P}((X \cap A) \cup (Y \cap A) \cup (X \cap Y)) \]

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

Definition 4.1.13 (Trace interleaving)

\( t, u \in \text{timedTrace}; \ a, b \in A; \ c, d \notin A; \ S, T \in \mathbb{P} \Sigma \)

\[
\begin{align*}
{t ||_A u} & = u ||_A t \\
{\langle \rangle ||_A \langle \rangle} & = \{\langle \rangle\} \\
{\langle \rangle ||_A \langle b \rangle} & = \{\langle \rangle\} \\
{\langle \rangle ||_A \langle d \rangle} & = \{\langle d \rangle \mid v \in t ||_A u\} \\
{\langle \rangle ||_A \langle T, tock \rangle} & = \{\langle \rangle\} \\
{\langle a \rangle \cap t ||_A \langle a \rangle} & = \{\langle a \rangle \cap v \mid v \in t ||_A u\} \\
{\langle a \rangle \cap t ||_A \langle b \rangle} & = \{\langle \rangle\} \\
{\langle a \rangle \cap t ||_A \langle d \rangle} & = \{\langle d \rangle \cap v \mid v \in \langle a \rangle \cap t ||_A u\} \\
{\langle a \rangle \cap t ||_A \langle T, tock \rangle} & = \{\langle \rangle\} \\
{\langle c \rangle \cap t ||_A \langle d \rangle} & = \{\langle c \rangle \cap v \mid v \in t ||_A \langle d \rangle \cap u\} \cup \{\langle d \rangle \cap v \mid v \in \langle c \rangle \cap t ||_A u\} \\
{\langle c \rangle \cap t ||_A \langle T, tock \rangle} & = \{\langle c \rangle \cap v \mid v \in t ||_A \langle T, tock \rangle \cap u\} \\
{S, tock \cap t ||_A \langle T, tock \rangle} & = \{\langle U, tock \rangle \cap v \mid U = S \cap_A T \land v \in t ||_A u\}
\end{align*}
\]

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

Lemma 4.1.4 (Minimality of trace composition)

\[ r \in t ||_A u \Rightarrow \exists s, w \cdot ((s \prec t \lor w \prec u) \land r \in s ||_A w) \]

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

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

Definition 4.1.14 (Hiding)

\[ P \setminus A \equiv \exists t \cdot P[t/t'] \land A \text{ urgent } t \land (tt' = t \setminus A) \]

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

Definition 4.1.15 (Urgency)

\[ A \text{ urgent } t \equiv \forall s, X \cdot s \not\sim (X, \text{ tock}) \leq t \Rightarrow A \subset X \]

The semantic hiding operator is then defined inductively:

Definition 4.1.16 (Trace hiding)

\[
\begin{align*}
\langle \rangle \setminus A & = \langle \rangle \\
(\langle S, \text{ tock} \rangle \circ tt) \setminus A & = (S \setminus A, \text{ tock}) \circ (tt \setminus A) \\
(\langle a \rangle \circ tt) \setminus A & = tt \setminus A \\
(\langle b \rangle \circ tt) \setminus A & = \langle b \rangle \circ (tt \setminus A)
\end{align*}
\]

4.1.7 Timeout

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

The operator is defined in two cases. In one case, at least \( n \) time units have passed without a visible event: \( \text{tock}^n \leq \text{trace}(tt') \). To account for this behaviour, \( P \) must have been able to wait for this period without engaging in any external events; the subsequent trace is then a behaviour of \( Q \). It is given by \( tt' - u \), where trace subtraction is defined in the usual way. The other case is the complement: fewer than \( n \) time units have passed, say \( m \), without a visible event: \( \neg \text{tock}^n \leq \text{trace}(tt') \). Now, if the idle suffix is empty, then it must be possible for \( P \) to wait for \( m \) time units. On the other hand, if the idlesuffix is non-empty, then it must also have been possible for \( P \) to wait \( m \) time units and then perform the idle suffix. Either way, the trace is a behaviour of \( P \).
Definition 4.1.17  *Timeout*

\[
P \xrightarrow{n} Q \equiv \\
(\exists u \bullet u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[u/\text{tt'}] \land Q[tt' - u/\text{tt'}]) \\
\leq \text{tock}^n \leq \text{trace}(tt') \\
P
\]

4.1.8  *Recursion*

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

Definition 4.1.18 (Recursion)

\[
\mu F = \bigcap \{ P \mid F(P) \subseteq P \}
\]

4.2  *Lowe & Ouaknine’s Axioms*

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

4.2.1  *Well Foundedness*

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

Definition 4.2.1 (T1: Well foundedness)

\[
T1(P) = P[\emptyset/\text{tt'}]
\]

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

Proof 4.2.1 See Appendix.

4.2.2  *Prefix Closure*

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

Definition 4.2.2 (T2: Prefix closure)

\[
T2 \quad [ P \land t \preceq tt' \Rightarrow P[t/\text{tt'}]]
\]

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

Proof 4.2.2 See Appendix.
4.2.3 Refusals

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

**Definition 4.2.3 (T3: Refusals)**

\[
T_3(P) = P \land (P[tt' \leftarrow (A, tock)/tt'] \land \neg P[tt' \leftarrow \langle a \rangle/\langle tt' \rangle]) \Rightarrow P[tt' \leftarrow (A \cup \{a\}, tock)/\langle tt' \rangle])
\]

**Theorem 4.2.3 (Refusals)** Every CML operator preserves \( T_3 \)-healthiness.

4.2.4 Timelock Freedom

A process can always allow time to pass.

**Definition 4.2.4 (T4: Timelock freedom)**

\[
P \Rightarrow P[tt' \leftarrow (\emptyset, tock)/\langle tt' \rangle]
\]

**Theorem 4.2.4 (Timelock freedom)** Every CML operator preserves \( T_4 \)-healthiness.

4.2.5 Zeno Freedom

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

**Definition 4.2.5 (T5: Zeno freedom)**

\[
T_5(P) = \forall k. \exists n. \forall tt'. P \Rightarrow (\#(tt' \uparrow tock) \leq k \Rightarrow \#(\text{trace}(tt')) \leq n)
\]

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

**Theorem 4.2.5 (Zeno freedom)** Suppose that \( P \) is a time-guarded process, then for every \( k \) there is an \( n \), such that \( P \) is \( T_5 \)-healthy.

**Proof 4.2.3** See Appendix.

4.2.6 Time-guardedness

A syntactic check is available to ensure that a CML process is time-guarded. The following is adapted from [24].
Definition 4.2.6 (Time-guardedness) If \( X, Y \) are \( \text{CML} \) process variables, and \( A, B \subseteq \Sigma \), then a \( \text{CML} \) term is time-guarded for \( X \), provided it has been constructed from terms satisfying the following rules:

- \( \text{STOP} \)
- \( Y \neq X \)
- \( a \to P, \text{provided } P \text{ is time-guarded for } X \)
- \( P \setminus A \)
- \( \mu \ Y . F(X) \)
- \( P \triangleright Q, \text{provided } n \geq 1 \text{ or } P \text{ is time-guarded for } X \)
- \( P \sqcap Q, P \sqcap Q, P \parallel_A Q, \text{provided } P \text{ and } Q \text{ are time-guarded for } X \)

If \( P \) is time-guarded for \( X \), then we may safely construct the recursive process \( \mu X . P(X) \).

### 4.3 Timed Imperative Sequential Reactive Processes

In this section, we extend our treatment of \( \text{CML} \) by including sequential composition and imperative state. We introduce four new observations:

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

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

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

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

\[
\begin{array}{ccc}
\text{tt} & Q & P \\
\text{tt'} & & \\
\text{rt} & & \\
\text{rt'} & & \\
\end{array}
\]
The behaviour of \( P \) is represented by the trace \( tt' \). The behaviour of the predecessors of \( P \) (\( Q \) in the diagram) is represented by the trace \( rt \), which in this diagram is equal to the trace \( tt \) (although, we shall not need to refer to \( tt \) again, instead referring to \( rt \)). Finally, the trace of the entire system, including the behaviour of \( P \) and its predecessors, is given by \( rt' \).

### 4.3.1 Healthiness Conditions

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

**Definition 4.3.1 \( \text{RT0} \)**

\[
\text{RT0}(P) = T0(P) \land (rt \in \text{timedTrace}) \land (rt' \in \text{timedTrace})
\]

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

**Definition 4.3.2 \( \text{RT1} \)**

\[
\text{RT1}(P) = P \land (rt' = rt \land tt')
\]

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

**Definition 4.3.3 \( \text{RT2} \)**

\[
\text{RT2}(P(rt, rt')) = P[(), tt'/rt, rt']
\]

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

**Definition 4.3.4 \( \text{RT3} \)**

\[
\text{RT3}(P) = (\Pi_{rt} \triangleleft wait \triangleright P)
\]

where \( \Pi_{rt} = \text{RT}(true \triangleright \Pi) \)

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

**Definition 4.3.5 (RT4)**

$$RT4(P) = RT01 \circ T0(\neg ok) \lor P$$

where $RT01 = RT0 \circ RT1$.

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

**Definition 4.3.6 (RT5)**

$$RT5(P) = P ; J$$

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

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

**Lemma 4.3.1 (RT monotonicity)** $RT0$–$RT5$ are all monotonic idempotents.

**Lemma 4.3.2 (RT commutativity properties)** $RT0$–$RT5$ all commute, with the exception of

1. $RT0(RT2(P)) \Rightarrow RT2(RT0(P))$, but $RT2(RT0(P)) \not\Rightarrow RT0(RT2(P))$.
2. $RT4(RT2(P)) \Rightarrow RT2(RT4(P))$, but $RT2(RT4(P)) \not\Rightarrow RT4(RT2(P))$.

**Definition 4.3.7 (RT)**

$$RT \equiv RT0 \circ RT1 \circ RT2 \circ RT3 \circ RT4 \circ RT5$$

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

### 4.3.2 Sequential Composition

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

**Definition 4.3.8 (Sequential composition)**

$$P ;_{RT} Q = P ; Q$$
4.3.3 Assignment

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

Definition 4.3.9 (Assignment)

\[
(x :=_{rt} e) = \text{RT013}(\text{true} \vdash (\text{trace}(tt') \in \text{lock}^*) \land \neg \text{wait} \land (x' = e))
\]

4.3.4 STOP

The definition of \( \text{STOP} \) in Definition 4.1.6 is almost what we need, but we need to make it aware of our new observational variables and turn it into a design. First, it is a design with precondition \( \text{true} \); second, it is perpetually waiting; third, it does not modify the state variables; and finally, it is \( \text{RT013} \)-healthy.

Definition 4.3.10 (Deadlock)

\[
\text{STOP}_{rt} = \text{RT013}(\text{true} \vdash \text{STOP}_{tt} \land \text{wait} \land x' = x)
\]

4.3.5 SKIP

We define \( \text{SKIP} \) to be the vacuous assignment.

Definition 4.3.11 (Termination)

\[
\text{SKIP}_{rt} = (v' :=_{rt} v)
\]

4.3.6 Prefix

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

Definition 4.3.12

\[
P^c_b = P[b, c/\text{wait}, \text{ok}']
\]

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

An event-prefixed process \( a \rightarrow P \) is able to diverge if \( P \) diverges, but we know that this can happen only after an \( a \) event. So, \( P'f[(a) \sim tt'/tt] \) is \( P \)'s divergent behaviour on any trace starting with the event \( a \). The precondition is the negation of this: \( \neg P'f[(a) \sim tt'/tt] \). The postcondition is given by the testing traces prefix operator.
Definition 4.3.13 (Prefix)
\[ a \rightarrow_{RT} P = RT013(\neg P_f^t((a) \sim tt' / tt) \vdash a \rightarrow_{RT} P_f^t) \]

4.3.7 Internal Choice

Internal choice is simply disjunction, as usual.

Definition 4.3.14 (Internal choice)
\[ P \cap_{RT} Q = P \lor Q \]

4.3.8 External Choice

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

Definition 4.3.15 (External choice)
\[ P \square_{RT} Q = RT013(\neg (P \lor Q)_f^t \vdash P_f^t \square_{RT} Q_f^t) \]

4.3.9 Timeout

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

\[ ((\text{trace}(tt') \leq \text{tock}^n) \Rightarrow P_f^t) ; RT01(\text{true}) \]

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

\[ (P_f^t \land (\text{trace}(tt') = \text{tock}^n)) \ wp \neg Q_f^t \]

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

Definition 4.3.16 (Timeout)
\[ P \triangleright_{RT}^n Q = RT013\left(\neg (\neg ((\text{trace}(tt') \leq \text{tock}^n) \Rightarrow P_f^t) ; RT01(\text{true})) \land (P_f^t \land (\text{trace}(tt') = \text{tock}^n)) \ wp \neg Q_f^t \right) \]

\[ P_f^t \triangleright_{RT}^n Q_f^t \]
4.3.10 Parallel Composition

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

The precondition of the parallel composition of $P$ and $Q$ is the conjunction of the preconditions of $P$ and $Q$. The postcondition merges the intermediate or final states of the two processes. It does this by running the two postconditions in parallel using the testing traces parallel operator. Since the program variables are partitioned, the equation $(v' = v)$ takes care of the appropriate merging of these programming variables, and we need worry only about merging the observational variables. The testing traces parallel operator has already taken care of the $tt_0$ trace, which then determines the value of $rt'$. The parallel composition reaches a stable state providing the two operands both reach a stable state, and this is taken care of by taking the conjunction of their individual results for their $ok_0$ variables. Similarly, the composition is in a waiting state if either of the processes end up in a waiting state. This is taken care of by taking the disjunction of their waiting states.

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

$$P \|_{\text{RT}} Q = RT013 \left( \neg (P \lor Q)_f \vdash \exists \ ok_0', ok_1', wait0', wait1' \bullet (P_f[ok0', wait0'/ok', wait'] \|_{\text{RT}} Q_f[ok1', wait1'/ok', wait']) \land (rt' = rt) \land (v' = v) \land (ok' = ok0 \land ok1) \land (wait' = wait0 \lor wait1) \right)$$

4.3.11 Hiding

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

**Definition 4.3.18 (Hiding)**

$$P \setminus_{\text{RT}} A = RT013(\neg (P_f \setminus \text{RT} A) \vdash P_f \setminus \text{RT} A)$$

4.3.12 Recursion

If $F$ is $\text{RT}$-healthy, then the least fixed point of $F$ is just the $\text{RT}$-healthy least fixed point of the $\text{TT}$-healthy fixed-point of $F$.

**Definition 4.3.19**

$$(\mu X \bullet F(X)) = RT(\mu X \bullet F(X))$$
Chapter 5

Derived CML Language Constructs

Chapter 4 has given a semantic interpretation of the reactive subset of CML, including imperative and sequential processes (in Section 4.3.) This chapter shows how that treatment may be extended to cover (i) expressions (including type operators), (ii) the remaining process operators (either in terms of definitions in operators we have already defined, or directly in terms of the timed trace semantics), and (iii) replication of actions.

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 dealt with there, and to illustrate the meaning of operators not treated in Chapter 4.

The remainder of the chapter is structured as follows: Section 5.1 addresses expressions, and Section 5.2 the specification and assignment statements. Actions (Section 5.3), parallel actions (Section 5.4) and replicated actions (Section 5.5) are next, followed by control statements in Section 5.6. Section 5.9 concludes.

5.1 Expressions

Expressions are the basic building blocks of the language. Their syntax is given in the CML0 language definition [33, Section 17] and D31.2c [7].

UTP is essentially generic with respect to expression notation. For the CML semantics, some expression operators are required to manipulate the auxiliary observable variables needed to capture the meaning of the language. However, expressions tend to be limited to Boolean operations on Boolean variables, as well as operations on sequences (e.g., head, append) to manipulate traces. These are expressed using the notation preferred in [14].

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

The CML semantics under development exploits this genericity. It does not need to refer explicitly to expression notation in order to give meaning to the higher level operators of CML (process composition, flow of control etc) that are based on it. However, in practice, the expression syntax can be thought of 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 [18, 23], giving rise to the alphabetised relations required by the CML semantics.

In CML, arithmetic connectives are given the obvious meaning. Issues of undefineness are dealt with in Chapter 6. Propositional connectives and quantifiers are synonyms for corresponding UTP operators: CML’s not, and, or, => and <=> correspond to UTP’s operators ¬, ∧, ∨, ⇒, and ⇔. CML’s forall and exists are equivalent to the use of the quantifiers ∀ and ∃ in UTP.

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

5.1.1 Maps

A map type records a relation between elements of two types. A map type from a type X to a type Y is a type that associates with each element of X (or a subset of X) an element of Y. A map can be thought of as an unordered collection of pairs. The first elements in each pair must be unique, and the set of all first elements is called the domain of the map. The set of the second elements in the pairs is called the range of the map.

The CML expression

\[ m = \text{map } X \text{ to } Y \]

describes a map from type X to type Y, labelled m. The map m is therefore some subset of the set

\[ \{(x, y) \mid x \in X, y \in Y\} \]

The CML operator \text{dom} returns the domain of a mapping:

\[ \text{dom } m = \{x \mid \exists y \bullet (x, y) \in m\} \]

The CML operator \text{rng} returns the range of a mapping. The range is made of members of Y which are mapped to by an element of X.

\[ \text{rng } m = \{y \mid \exists x \bullet (x, y) \in m\} \]

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

\[ m = \text{inmap } X \text{ to } Y \]
then
\[
m = \text{map } X \to Y \land \forall x_1, x_2 \in \text{dom } m \cdot x_1 \neq x_2 \Rightarrow m(x_1) \neq m(x_2)
\]

\textit{CML} defines mapping operators to manipulate these constructs. In the following definitions, \( m = \text{map } X \to Y \).

The \textit{CML} operator \texttt{munion}, or \textit{mapping union}, combines two mappings. It is only defined for mappings with distinct domains.

\[
m \texttt{munion} n = \{(x, y) | (x, y) \in m \cup n\}
\]

If the domains of the mappings overlap, then \( m \texttt{munion} n \) is undefined. (The issue of undefinedness is addressed in Chapter 6.) In general, a map may be \textit{overwritten} with another map using the operator \texttt{++}. 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.

\[
m ++ n = \{(x, y) | (x, y) \in n\} \cup \{(x, y) | x \in \text{dom } m \setminus \text{dom } n \land (x, y) \in m\}
\]

A \textit{domain subtraction} operator is defined within \textit{CML}. Given a set of elements \( s \), and a map \( m \) of type \texttt{map X to Y}, the mapping obtained by removing the set \( s \) from the domain of \( m \) is written as

\[
s <\neg:\ m
\]

and defined as

\[
s <\neg:\ m = \{(x, y) | x \in (\text{dom } m) \setminus s \land (x, y) \in m\}
\]

The \textit{mapping restriction} operator that \textit{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

\[
s <: m
\]

and this is defined as

\[
s <: m = \{(x, y) | x \in s \cap \text{dom } m \land (x, y) \in m\}
\]

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

\[
m :-> s
\]

and defined as

\[
m :-> s = \{(x, y) | y \in (\text{rng } m) \setminus s \land (x, y) \in m\}
\]

The \textit{range restriction} operator limits the mapping to the subset of the mapping where the range intersects with the identified set. It is written

\[
m :-> s
\]

and defined as

\[
m :-> s = \{(x, y) | y \in \text{rng } m \cap s \land (x, y) \in m\}
\]
5.2 Specification and Assignment

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

The notion of design needs to be reinterpreted within the timed-traces semantic framework via the healthiness function \( \text{RT} \). The assignment \( x := e \) is based on the redefined assignment operator for timed traces. In practice this is based on the redefinition of design for timed traces (\( \Rightarrow_{\text{RT}} \)). Table 5.1 provides the correspondence between specification constructs in CML and UTP, based on \( \text{RT} \).

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
<th>Notes</th>
<th>CML0 Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{pre}\ P \text{post}\ Q])</td>
<td>( P \Rightarrow_{\text{RT}} Q )</td>
<td>Specification Statement</td>
<td>Section 13 and 14</td>
</tr>
<tr>
<td>([\text{post}\ Q])</td>
<td>( \text{true} \Rightarrow_{\text{RT}} Q )</td>
<td>As above, no precondition supplied</td>
<td>As above</td>
</tr>
<tr>
<td>( x := e )</td>
<td>( x := e )</td>
<td>Assignment (similarly for multiple assignment)</td>
<td>Section 15.2</td>
</tr>
</tbody>
</table>

Table 5.1: Specification and Assignment

5.3 Non-parallel Action Constructors

An action in CML defines the body of operations and processes. This section deals with the action constructors from [33, Section 15]. It includes flow-of-control operators, process operators and time operators. These are discussed in detail in the following subsections.

5.3.1 Termination, Deadlock, Chaos and Divergence

The CML actions Skip and Stop are basic actions, and are given a UTP semantics directly in chapter 4.

The CML action Chaos can preform or refuse any event, but its behaviour is not completely arbitrary – it must respect the behavioural healthiness conditions \( \text{RT} \). The definition of Chaos is given in terms of the design notation.

\[
\text{Chaos} = \text{false} \Rightarrow_{\text{RT}} \text{true}
\]

The divergent process Div is the bottom of the lattice (written \( \bot \) in mathematical notation.) It does not respect the healthiness conditions \( \text{RT} \).
5.3.2 Delay

The CML process \( \text{Wait } n \) is the process that waits for \( n \) time units before terminating successfully. It is defined as

\[
\text{Wait } n = \text{Stop} \triangleright n
\]

5.3.3 Replicated Prefix

The complex form of action prefix

\[
e!x : P(x) \rightarrow A(x)
\]

can be translated into a choice of simple action prefix actions

\[
\square_{x \in P(x)} (e!x \rightarrow A(x))
\]

5.3.4 Guarded action

The process \( \langle g \rangle \& A \) behaves like action \( A \) if \( g \) is true, otherwise it deadlocks. It can be defined in terms of the conditional operator.

\[
\langle g \rangle \& A = A \circ g \triangleright \text{Stop}
\]

5.3.5 Untimed interrupt

The untimed interrupt construct \( A \setminus B \) behaves as \( A \), until the occurrence of one of the initial events of \( B \), after which \( A \) is disabled and \( B \) takes over. If \( A \) terminates before \( B \) interrupts, then \( B \) is disabled. The initial events of \( B \) are given by \( \text{initials}(B) \). It is written as \( A \triangle B \) in mathematical notation.

For simplicity, we impose the requirements that the first observation of any of the initial events of \( B \) is the point at which control transfers from \( A \) to \( B \), and that the composed process will not initially refuse to perform any of the initial events of \( B \).

In the first case, therefore, if no initial events of \( B \) have been observed in the trace \( tt' \), the observed trace is wholly due to \( A \), and no initial events of \( B \) have been refused.

If, on the other hand, we have seen one of \( \text{initials}(B) \), we know that at that point control has been passed to process \( B \). The first part of the trace is behaviour of \( A \) (which must not lead to a terminated state of \( A \)) and the second part of the trace is a behaviour of \( B \).

\[
A \triangle B = \begin{cases} 
A \land \text{initials}(B) \cap \text{refusals}(tt') = \emptyset \\
\langle \text{initials}(B) \cap \text{events}(tt') = \emptyset \rangle \\
\exists u, v \cdot tt' = u \land v \land \\
\text{events}(u) \cap \text{initials}(B) = \emptyset \\
\text{refusals}(u) \cap \text{initials}(B) = \emptyset \\
A[u, \text{false}/tt', \text{wait}] \land B[v/tt']
\end{cases}
\]
5.3.6 Untimed timeout

The untimed variant of the timeout operator may be defined in terms of the basic operator, which has been given a semantics in Chapter 4. Recall that once the action on the left of the timeout operator starts, the action on the right is disabled.

The untimed timeout operator is represented by the notation $A \triangleright B$, written in CML as $A[\triangleright B]$. The interrupting action may non-deterministically choose to cut in at any time, and is defined as a non-deterministic choice over all possible timeout values.

$$A \triangleright B = \bigcap_{i \in N} A \triangleright B$$

5.3.7 Timed interrupt

The timed interrupt operator allows one action to interrupt another after a fixed delay. It is written as $A \triangleleft_n B$, or $A/\{n\} \backslash B$ in CML notation. It is given a direct definition in terms of the timed trace semantics. Either the specified time delay has not passed, in which case the action $A \triangleleft_n B$ behaves as $A$, or the time delay has passed $(\#(trace(tt')) \{tock\}) > n$, in which case the first $n$ time units of the observed trace must be a (non-terminating) observation of $A$, and the remainder must be an observation of the interrupting action $B$.

$$A \triangleleft_n B = \left\{ A \begin{array}{l} \varnothing (\#(trace(tt')) \{tock\}) \leq n \triangleright \exists u \bullet \#(trace(u) \{tock\}) = n \land A[u, false/\tt', wait] \land B[tt' - u/\tt'] \end{array} \right\}$$

5.3.8 The startsby operator

In the CML construct $A$ startsby $n$, the action $A$ must begin its interaction behaviour before the deadline $n$, otherwise it becomes infeasible. It is defined using the timed timeout operator.

$$A \text{ startsby } n = A \triangleright n \top$$

5.3.9 The endsby operator

This operator insists that a process either finishes by a deadline, or becomes infeasible. It is defined in terms of the timed interrupt operator.

$$A \text{ endsby } n = A \triangleleft_n \top$$

5.3.10 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 $A[nc/c]$.
5.3.11 Mutual recursion

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

\[ \mu X_1 \ldots X_n \bullet F_1(X_1 \ldots X_n) \ldots F_n(X_1 \ldots X_n) \geq \cap \{ (X_1 \ldots X_n) \mid F_1(X_1 \ldots X_n) \ldots F_n(X_1 \ldots X_n) \} \]

5.4 Parallel Action Constructors

The parallel operators are shown in Table 4, Section 15 of [33]. There are eight variants in all. The denotational semantics defines the semantics for a single general operator and the remainder are expressed in terms of this one operator.

The general operator \( (A \parallel_{\text{cs}} B) \) is defined semantically in Section 4.3.10. Its syntactic synonym is called generalised parallel and has the form: \( A [ \mid \text{ns}_1 \mid \text{cs} \mid \text{ns}_2 \mid ] B \). It behaves as \( A \) and \( B \) executed in parallel and synchronising on the set of channels in \( \text{cs} \). \( A \) (resp., \( B \)) can modify only the state components in \( \text{ns}_1 \) (resp., \( \text{ns}_2 \)).

If \( \alpha(A) = \{ \text{ns}_1, \text{wait}, \text{ok}, \text{tt} \} \) and \( \alpha(B) = \{ \text{ns}_2, \text{wait}, \text{ok}, \text{tt} \} \), then \( A [ \mid \text{ns}_1 \mid \text{cs} \mid \text{ns}_2 \mid ] B \) is defined as \( A \parallel_{\text{cs}} B \). Note that the definition allows \( A \) and \( B \) to write to all variables in their alphabets.

Following this, the definitions of the derived operators are straightforward, and are shown below.

5.4.1 Interleaving with state

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

\[ P [ \mid \mid \text{ns}_1 \mid \text{ns}_2 \mid ] Q \]

and defined to be

\[ P [ \mid \mid \text{ns}_1 \mid \text{ns}_2 \mid ] Q = P [ \mid \text{ns}_1 \mid \{ \mid \mid \} \mid \text{ns}_2 \mid ] Q \]

Channel names are introduced within channel brackets

\( \{ \mid \mid \} \)

and so the empty set of channel names is written \( \{ \mid \mid \} \).

5.4.2 Interleaving without state

The interleaving operator which does not allow actions \( A \) and \( B \) is written as

\[ A [ \mid \mid ] B \]

It is defined as

\[ A [ \mid \mid ] B = A [ \mid \{ \mid \} \mid \{ \mid \mid \} \mid \{ \mid \} \mid ] B \]
5.4.3 Synchronous parallelism

Synchronous parallelism is written as

\[ A [ | ns_1 | ns_2 | ] B \]

Actions \( A \) and \( B \) are executed in parallel synchronising on all channels (\( \Sigma \)). \( A \) can modify the state in \( ns_1 \) and \( B \) and modify the state in \( ns_2 \). It is defined as

\[ A [ | ns_1 | ns_2 | ] B = A [ | ns_1 | \Sigma | ns_2 | ] B \]

5.4.4 Synchronous parallelism without state

Synchronous parallelism without state is written as

\[ A || B \]

and is a shorthand for the case where actions \( A \) and \( B \) are executed in parallel synchronising on all channels (\( \Sigma \)), and neither \( A \) nor \( B \) can modify state. It is defined as

\[ A || B = A [ | {} | \Sigma | {} | ] B \]

5.4.5 Alphabetised parallelism with state

Alphabetised parallelism is written as

\[ A [ | ns_1 | X || Y | ns_2 | ] B \]

and defines an action combination in which \( A \) can only communicate on channels in \( X \) and \( B \) is restricted to communicating on channels in \( Y \). Note \( A \) will act like \texttt{Stop} if it tries to synchronise on a channel not in \( X \), and that it will synchronise independently of \( B \) if it tries to communicate on a channel in \( X \) but not in \( Y \).

To define alphabetised parallelism, we first define the following construct to restrict an action \( A \) to a set of channels \( X \):

\[ Res(A, ns1, X) = A [ | ns1 | \Sigma \setminus X | {} | ] STOP \]

\( A \) may write to the variables in the namespace \( ns_1 \).

Using this, the definition of alphabetised parallel is

\[ A [ | ns_1 | X || Y | ns_2 | ] B = \]
\[ Res(A, ns1, X) [ | ns_1 | X \cap Y | ns_2 | ] Res(B, ns2, Y) \]
5.4.6 Alphabetised parallelism without state

Alphabetised parallelism without state changes is written as

\[ A[ X \parallel Y ] B \]

\( A \) and \( B \) may only communicate on channels in \( X \) and \( Y \) respectively, and must synchronise on the channels in \( X \cap Y \). Neither is allowed to write to the state.

\[ A[ X \parallel Y ] B = \text{Res}(A, \{\}, X) \parallel \{\} \parallel X \cap Y \parallel \{\} \parallel \text{Res}(B, \{\}, Y) \]

5.4.7 Generalised parallelism without state

In generalised parallelism without state, written as

\[ A \parallel cs \parallel B \]

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

\[ A \parallel cs \parallel B = A \parallel \{\} \parallel cs \parallel \{\} \parallel B \]

5.5 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 (section 15) of the CML0 language definition [33].

5.5.1 Replicated sequential composition

Replicated sequential composition is written in CML as

\[ ; i : e @ A(i) \]

The variable \( e \) must be a sequence, and for each \( i \) in the sequence \( A \), the elements \( A(i) \) are executed in order.

The meaning of replicated sequential composition is defined using recursion over the sequence \( e \) using a local variable \( c \).

\[
\text{var } c, \text{seq} := e; \mu X \bullet \left( \begin{array}{c}
c := \text{head}(\text{seq}); A(c); \text{seq} := \text{tail}(\text{seq}); X \\
\end{array} \right. \begin{array}{c}
\land \text{seq} \neq () \\
\text{end } c, \text{seq}
\end{array}
\]
5.5.2 Replicated external choice

Replicated external choice is written as

\[ [] \,: \,\, e @ A(i) \]

and its meaning is a generalisation of the formula for external choice. Either no observable event has occurred, in which case all components \( A(i) \) must be capable of delaying their activity by the elapsed time, or one of the \( A(i) \) components must have been chosen. If \( e \) is empty, the construct is equivalent to \( STOP \).

\[ \land_{i \in e} A(i)[idleprefix(tt')/tt'] \land \lor_{i \in e} A(i) \]

5.5.3 Replicated internal choice

Replicated internal choice is only defined over nonempty sets. Replicated internal choice over a set \( e \) is written as

\[ |\sim| \,: \, e @ A(i) \]

and its meaning is a generalisation of the meaning of internal choice, which is disjunction. The set \( e \) must not be empty.

\[ \lor_{i \in e} A(i) \]

5.5.4 Replicated interleaving

Replicated interleaving is written as

\[ ||| \,: \, e @ \ns(i) A(i) \]

where \( \ns(i) \) is the name set of variables in action \( A(i) \)'s state.

When the set \( e \) has precisely two elements, say 1 and 2, the operator is defined as the binary stateful form of the interleaving operator defined in Section 5.4.1. For simplicity, we assume here that the set \( e \) is a set of contiguous natural numbers starting from 1.

\[ A(1) \, ||| \, \ns(1) \land \ns(2) \, ||| \, B(2) \]

and if

\[ ||| \,: \, \{1\ldots j-1\} @ \ns(i) A(i) \]

is defined for the first \( j - 1 \) elements, then the definition for \( j \) elements is given as

\[ A(j) \, ||| \, \ns(j) \land \ns_{j-1} \, ||| \, \{1\ldots j-1\} @ \ns(i) A(i) \]

where \( \ns_j \) is defined as \( \bigcup_{i=1\ldots j} \ns(i) \).
5.5.5 Replicated generalised parallelism

Replicated generalised parallelism is written as

\[ \begin{array}{l}
[| cs |] i: e @ [ns(i)] A(i)
\end{array} \]

In the case binary case, this operator reduces to the operator defined in Section 4.3.10. When combining a larger number of processes the definition may be recursively constructed as follows.

If

\[ \begin{array}{l}
[| cs |] i: \{1...j-1\}@ [ns(i)] A(i)
\end{array} \]

is defined for the first \( j - 1 \) elements, then we extend the definition to element \( j \) as

\[ \begin{array}{l}
A(j) [| ns(j) | cs | ns_{j-1} |] ([| cs |] i: \{1...j-1\}@ [ns(i)] A(i))
\end{array} \]

where \( ns_j \) is again defined as \( \bigcup_{i \in 1...j} ns(i) \).

5.5.6 Replicated alphabetised parallelism

Replicated alphabetised parallelism is written as

\[ \begin{array}{l}
|| i: e @ [ns(i)|cs(i)] A(i)
\end{array} \]

In the binary case, the operator is defined as in Section 5.4.5.

To extend a definition for \( j - 1 \) actions

\[ \begin{array}{l}
|| i: \{1...j-1\}@ [ns(i)|cs(i)] A(i)
\end{array} \]

to action \( A(j) \), we write

\[ \begin{array}{l}
A(j)
\end{array} \]

\[ \begin{array}{l}
[| ns(j) | cs(j) | cs_{j-1} | ns_{j-1} |]
\end{array} \]

\[ \begin{array}{l}
( i: \{1...j-1\}@ [ns(i)|cs(i)] A(i))
\end{array} \]

where \( cs_j \) is defined as \( \bigcap_{i \in 1...j} cs(i) \) and \( ns_j \) is again defined as \( \bigcup_{i \in 1...j} ns(i) \).

5.5.7 Replicated synchronous parallelism

Replicated synchronous parallelism is written as

\[ \begin{array}{l}
|| i: e @ [ns(i)] A(i)
\end{array} \]

All processes \( A(i) \) synchronise on all events. Each \( A(i) \) may only modify state components in \( ns(i) \). This can be defined in terms of the replicated generalised parallel operator, where the synchronisation namespace is that of all events.
\[ \text{i.e.} \; \forall i \in \text{ns}(i) A(i) \]
is thus defined as
\[ \text{i.e.} \; \forall i \in \text{ns}(i) A(i) \]
where \( a_j \) is defined as \( \bigcup_{i \in 1..j} \alpha(A_i) \).

5.6 Control Statements

Control statements (Table 7 of [33]) 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 [33, 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.

\( e^T \) is defined as \( \bigwedge e \bowtie e \bowtie T_D \). Each construct may be generalised in the obvious way.

5.6.1 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 nondeterministically and the corresponding action is executed.

The binary form of the nondeterministic if statement is written

\[
\begin{align*}
\text{if } e_1 \rightarrow a_1 \\
| e_2 \rightarrow a_2 \\
\text{end}
\end{align*}
\]

which means
\[
(e_1^T; a_1 \cap e_2^T; a_2) \bowtie e_1 \lor e_2 \nabla \perp
\]

The generalised form, with \( n \) guards, is written
\[
\bigwedge_{i \in \{1..n\}} (e_i^T; a_i) \bowtie \bigvee_{i \in \{1..n\}} e_i \nabla \perp
\]

5.6.2 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
\begin{verbatim}
  do e_1 -> a_1
      | e_2 -> a_2
  end
\end{verbatim}

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

\[ \mu X \bullet (e^1_T; a_1 \cap e^2_T; a_2); X \left( e_1 \lor e_2 \right) \sqcup \Pi \]

The generalised form is given as

\[ \mu X \bullet \cap_{i \in \{1..n\}} (e_i_T; a_i); X \left( \lor_{i \in \{1..n\}} e_i \right) \sqcup \Pi \]

### 5.6.3 Conditionals and case statements

Next are the conditional and cases statements, which feature in [33, Section 15.8]. The definitions in the semantic notation for each construct are given in Table 5.2. 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.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantic equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{if} \textit{e} \textbf{then} \textit{a}</td>
<td>\textit{a} \left&lt; e \right&gt; \sqcup \Pi</td>
<td>no elseif/else</td>
</tr>
<tr>
<td>\textbf{if} \textit{e} \textbf{then} \textit{a}_1  \textbf{else} \textit{a}_2</td>
<td>\textit{a}_1 \left&lt; e \right&gt; \sqcup \textit{a}_2</td>
<td>without elseif</td>
</tr>
<tr>
<td>\textbf{if} \textit{e}_1 \textbf{then} \textit{a}_1  \textbf{elseif} \textit{e}_2 \textbf{then} \textit{a}_2 \textbf{else} \textit{a}_3</td>
<td>\textit{a}_1 \left&lt; e_1 \right&gt; \lor (\textit{a}_2 \left&lt; e_2 \right&gt; \lor \textit{a}_3)</td>
<td>with else/elseif</td>
</tr>
<tr>
<td>\textbf{cases} \textit{e}_1: \textit{(e}_2\textbf{)} \textbf{then} \textit{a}_1  \textbf{then} \textit{a}_2 \textbf{...} \textbf{then} \textit{a}<em>n \textbf{others} \textbf{then} \textit{a}</em>{n+1} \textbf{end}</td>
<td>\textit{a}<em>1 \left&lt; (e_1 = e_2) \lor \ldots \lor (e_n = e</em>{n+1} \lor \ldots) \right&gt;</td>
<td>Cases statement</td>
</tr>
</tbody>
</table>

Table 5.2: Conditionals

### 5.6.4 Loops

The final set of control statements concern loops, which are presented in [33, Section 15.9]. These comprise the sequence for loop, the set for loop, the index for loop, and the while loop.
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 in the sequence.

$$\text{for } x \text{ in } s \text{ do } a$$

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.

$$\text{var } x, v := s; \mu X \bullet ((x, v := \text{head}(v), \text{tail}(v); a; X) \triangleleft v \neq () \triangleright I);$$

$\text{end } v, x$

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.

$$\text{forall } x \text{ in } s \text{ do } a$$

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.

$$\text{var } x, T := s; \mu X \bullet ((\{x \in T; a; T := T \setminus \{x\}; X) \triangleleft T \neq \emptyset \triangleright I);$$

$\text{end } T, x$

Index for loop

The index for loop steps through a range in user-defined increments. It is written as below.

$$\text{for } i = e_1 \text{ to } e_2 \text{ by } e_3 \text{ do } a$$

The variable $c$ takes the value $e_1$ initially, and performs the action $a$, increments $c$ and recursively calls the conditional. If $c$ is greater than $e_2$ the construct terminates gracefully. If $c$ is less than or equal to $e_2$ the conditional is again recursively called.

$$\text{var } c := e_1; f := e_2; \text{ step := } e_3; a; \mu X \bullet ((a; c := c + \text{step}; X) \triangleleft c \leq f \triangleright I);$$

$\text{end } c, f, \text{ step}$
While loop

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

\[
\text{while } e \text{ do } a
\]

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

\[
\mu X \cdot ((a; X) \triangleleft e \triangleright \Pi)
\]

5.7 Processes

A \textit{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, and therefore follow the definitions in Section 5.5.

5.7.1 Replicated generalised parallelism

Replicated generalised parallelism is written as

\[
[| cs |] \triangleleft e \triangleright A(i)
\]

When the set \( e \) has two elements, say \( x \) and \( y \), the above is defined as generalised parallelism:

\[
A(x) [| cs |] B(y)
\]

and when the cardinality of \( e \) is greater than two, it is defined as

\[
\prod_j (A(j) [| cs |] (|| cs || i:e \setminus \{j\}@A(i)))
\]

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

If \( e \) has cardinality one then it is defined as \( A(i) \), where \( \{i\} = e \). It is undefined if \( e \) is empty.

5.7.2 Replicated alphabetised parallelism

Replicated alphabetised parallelism is written as

\[
|| i:e \triangleleft [cs(i)] A(i)
\]
In this case all actions synchronise on the intersection of the sets \( cs(i) \). It is defined in a similar way to the generalised parallelism operator. When the set \( e \) has two elements, say \( x \) and \( y \), the definition is:

\[
A(x) [x | y] B(y)
\]

and when the cardinality of \( e \) is greater than two, it is defined as

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

### 5.7.3 Replicated synchronous parallelism

Actions combined using replicated synchronous parallelism must all synchronise on all events. Replicated synchronous parallelism is written as

\[
|| i : e @ A(i)
\]

It is equivalent to replicated alphabetised parallelism, where the synchronisation alphabet is the alphabet of one of the operands, \( \alpha(A(j)) \) for some \( j \in e \). Since the operator is homogeneous, the operands all must have the same alphabet.

\[
|| i : e @ [\alpha(A(j))] A(i)
\]

### 5.7.4 Replicated interleaving

Replicated interleaving is written as

\[
||| i : e @ A(i)
\]

All actions proceed independently in parallel, and there is no synchronisation. It can be defined in terms of alphabetised parallelism, where the synchronisation alphabet is the empty set.

\[
|| i : e @ [\emptyset] A(j)
\]

### 5.8 Parameters

#### 5.8.1 Result parameter

A result is written from a process using the parameter \( \text{res} \). This is written as

\[
\text{res} \ x : T @ P
\]

The meaning is given by the lambda function

\[
\text{res} \ x : T @ P = \lambda y : \text{var}(T) \bullet (\text{var} x; P; y := x; \text{end} x)
\]

where the fresh variable \( y \) ranges over variables of type \( T \), and stores the result of \( P \) on its completion.
5.8.2 Value parameter

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

\[ \text{val } x : T @ P \]

The meaning is given by the lambda function

\[ \text{val } x : T @ P = \lambda y : T \bullet (\text{var } x := y; P; \text{end } x) \]

where \( y \) is free in \( P \) and ranges over values of the type \( T \).

5.8.3 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

\[ \text{vres } x : T @ P \]

The meaning is given by the lambda function

\[ \text{val } x : T @ P = \lambda y : \text{var}(T) \bullet (\text{var } x := y; P; y := x; \text{end } x) \]

where \( y \) is free in \( P \) and ranges over variables of type \( T \).

5.8.4 Block statements

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

It is written as

\[ \text{dcl } x : T @ A \]

and defined as

\[ \text{dcl } x : T @ A = \lambda y : T \bullet (\text{var } x := y; A; \text{end } x) \]

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

\[ \text{dcl } x : T := e @ A \]

and defined as

\[ \text{dcl } x : T := e @ A = (\text{var } x := e; A; \text{end } x) \]
5.9 Summary

This Chapter 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 proportion of the CML syntax is considered out of scope because it pertains to the object-oriented features of the language, or to constructs that require the semantic extensions of object-orientation in order to be interpreted. Examples include method calls, parametrised and instantiated actions. These are discussed in part three of this deliverable.
Chapter 6

Undefinedness

6.1 Introduction

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

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

More generally, if we choose a specific treatment of undefined expressions, then is it possible to use verification tools with different treatments? For example, there are two different treatments of undefined expressions for VDM: Jones’s VDM uses the Logic of Partial Functions (LPF), which has been implemented in Isabelle [2], whilst Larsen’s VDM in Overture uses McCarthy Conditionals [17]. What is the relationship between these? How does the treatment in the VDM part interact with the other tools that we might use? For example, in the FDR implementation of CSPM [10], undefinedness is handled by a combination of arithmetic overflow and boolean short-circuit expressions. In the Circus tools, undefinedness is handled through the use of classical logic and arbitrary undefined values. Does any of this matter? And what if CML is used for a system of systems with heterogeneous constituents using different formalisms with different solutions to the undefined problem?

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

The work presented in this Chapter forms the basis of Victor Bandur’s PhD work and is
based on original ideas due to Mark Saaltink in his underpinnings for the Z/Eves theorem prover [28]. They have published joint papers with the authors at Marktoberdorf and ICECCS 2007 [32].

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

6.2 3-valued logic in UTP

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

6.2.1 Basic Sets and Constructors

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

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

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

We take inspiration from Rose’s standard encoding of three-valued logic [27], which is reminiscent of Hoare & He’s UTP designs [14, Chapter 3], in modelling three logical values using just a pair of predicates: $(P, Q)$. The intuitive meaning is that $P$ describes the region where $(P, Q)$ is true and $Q$ describes the region where $(P, Q)$ is defined. Just like Hoare & He designs, we can combine the pair of predicates into a single predicate by introducing an observational variable, in this case $def$: the observation that the predicate is defined. This gives us a model for the pair.

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

$$(P, Q) \models (def \Rightarrow P) \land (Q = def)$$
The next example demonstrates that this definition accounts for all three logical values.

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

\[
\begin{align*}
\text{true} &= (\text{true, true}) &= \text{def} \\
\text{false} &= (\text{false, true}) &= \text{false} \\
\bot &= \{ (\text{true, false}), (\text{false, false}) \} &= \neg \text{def}
\end{align*}
\]

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

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

\[
(P \land Q, Q) = (P, Q)
\]

Proof 6.2.1

\[
\begin{align*}
(P \land Q, Q) \\
\{ \text{Definition 6.2.1} \} \\
= (\text{def } \Rightarrow P \land Q) \land (Q = \text{def}) \\
\{ \text{Propositional calculus} \} \\
= (\text{def } \Rightarrow P) \land (Q = \text{def}) \\
\{ \text{Definition 6.2.1} \} \\
= (P, Q)
\end{align*}
\]

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

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

\[
R = (R^t, \neg R^f), \quad \text{where } R^b = R[b/\text{def}]
\]

Proof 6.2.2

\[
\begin{align*}
((P, Q)^t, \neg (P, Q)^f) \\
\{ \text{Definition 6.2.1, twice} \} \\
= (((\text{def } \Rightarrow P) \land (Q = \text{def}))^t, \neg ((\text{def } \Rightarrow P) \land (Q = \text{def}))^f) \\
\{ \text{Definition of } R^b. \} \\
= ((\text{true } \Rightarrow P) \land (Q = \text{true}), \neg ((\text{false } \Rightarrow P) \land (Q = \text{false})))
\end{align*}
\]
\[
\{ \text{Propositional calculus} \} = (P \land Q, \neg (\text{true} \land \neg Q))
\]
\[
\{ \text{Propositional calculus} \} = (P \land Q, Q)
\]

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

\((x = y \ast z), (y \neq 0)\)

We can consider three examples with specific values for \(x\), \(y\), and \(z\).

<table>
<thead>
<tr>
<th>((x = y \ast z), (y \neq 0))</th>
</tr>
</thead>
<tbody>
<tr>
<td>((3 = 6/2), (2 \neq 0))</td>
</tr>
<tr>
<td>((2 = 6/2))</td>
</tr>
<tr>
<td>((2 = 6/0))</td>
</tr>
<tr>
<td>= ((6 = 2 \ast 3), (2 \neq 0))</td>
</tr>
<tr>
<td>= ((6 = 2 \ast 2), (2 \neq 0))</td>
</tr>
<tr>
<td>= ((6 = 0 \ast 2), (0 \neq 0))</td>
</tr>
<tr>
<td>= ((\text{true}, \text{true}))</td>
</tr>
<tr>
<td>= (\text{false}, \text{true})</td>
</tr>
<tr>
<td>= (\text{false}, \text{false})</td>
</tr>
</tbody>
</table>

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

6.2.2 Conjunction

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

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

\[(P, Q) \land_T (R, S) \equiv (P \land R, Q \land S)\]

It is useful to see the truth table for conjunction:

<table>
<thead>
<tr>
<th>(\land_T)</th>
<th>def</th>
<th>(\neg) def</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>def</td>
<td>def</td>
<td>(\neg) def</td>
<td>false</td>
</tr>
<tr>
<td>(\neg) def</td>
<td>(\neg) def</td>
<td>(\neg) def</td>
<td>(\neg) def</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>(\neg) def</td>
<td>false</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>(\land_T)</th>
<th>(\text{true}_T)</th>
<th>(\bot_T)</th>
<th>(\text{false}_T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{true}_T)</td>
<td>(\text{true}_T)</td>
<td>(\bot_T)</td>
<td>(\text{false}_T)</td>
</tr>
<tr>
<td>(\bot_T)</td>
<td>(\bot_T)</td>
<td>(\bot_T)</td>
<td>(\bot_T)</td>
</tr>
<tr>
<td>(\text{false}_T)</td>
<td>(\text{false}_T)</td>
<td>(\bot_T)</td>
<td>(\text{false}_T)</td>
</tr>
</tbody>
</table>
Example 6.2.3 (Partial conjunction) Partial conjunction gives a meaning to the case where one operand is undefined.

\[(y = 3) \land_T (z = x/y)\]
\[= ((y = 3), \text{true}) \land_T ((x = y * z), (y \neq 0))\]
\[= ((y = 3) \land (x = y * z), \text{true} \land (y \neq 0))\]

6.2.3 Negation

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

\[\neg_T (P, Q) = (\neg P, Q)\]

The truth table is:

<table>
<thead>
<tr>
<th>(\neg)</th>
<th>def</th>
<th>false</th>
<th>(\neg)</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>def</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>(\neg)</td>
<td>def</td>
<td>false</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example 6.2.4 (Partial negation)

\[\neg_T (z = x/y)\]
\[= \neg_T ((x = y * z), (y \neq 0))\]
\[= ((x \neq y * z), (y \neq 0))\]

6.2.4 Disjunction

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

\[(P, Q) \lor_T (R, S) \equiv (P \lor Q, R \land S)\]

The truth tables are:

<table>
<thead>
<tr>
<th>(\lor)</th>
<th>def</th>
<th>(\neg)</th>
<th>def</th>
<th>false</th>
<th>(\lor)</th>
<th>true</th>
<th>(\neg)</th>
<th>def</th>
<th>false</th>
<th>(\lor)</th>
<th>true</th>
<th>(\neg)</th>
<th>def</th>
<th>false</th>
<th>(\lor)</th>
<th>true</th>
<th>(\neg)</th>
<th>def</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>def</td>
<td>def</td>
<td>(\neg)</td>
<td>def</td>
<td>false</td>
<td>true</td>
<td>(\neg)</td>
<td>true</td>
<td>(\neg)</td>
<td>def</td>
<td>false</td>
<td>true</td>
<td>(\neg)</td>
<td>true</td>
<td>def</td>
<td>false</td>
<td>true</td>
<td>(\neg)</td>
<td>true</td>
<td>def</td>
</tr>
</tbody>
</table>
Example 6.2.5 (Partial disjunction) Define \( P \Rightarrow T Q \) as \( \neg_T P \lor_T Q \). Now suppose that \( f \) is a partial function symbol, such that

\[
(y = f(x)) = ((y = f(x)), x \in \text{dom } f)
\]

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

\[
\begin{align*}
x \in \text{dom } f & \Rightarrow_T (y = f(x)) \\
&= \neg_T (x \in \text{dom } f) \lor_T (y = f(x)) \\
&= \neg_T (x \in \text{dom } f, \text{true}) \lor_T (y = f(x)) \\
&= (x \notin \text{dom } f, \text{true}) \lor_T (y = f(x)) \\
&= (x \notin \text{dom } f, \text{true}) \lor_T ((y = f(x)), x \in \text{dom } f) \\
&= (x \in \text{dom } f \Rightarrow (y = f(x)), x \in \text{dom } f) \\
&= ((y = f(x)), x \in \text{dom } f)
\end{align*}
\]

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

6.2.5 Equality

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

\[
\begin{align*}
(\text{def} =_T \neg \text{def}) &= \text{false} \\
(\text{def} =_T \text{false}) &= \text{false} \\
(\neg \text{def} =_T \text{false}) &= \text{false}
\end{align*}
\]

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

\[
(f(x, y) = \bot) \Leftrightarrow (x = \bot) \lor (y = \bot) \lor (f(x, y) = (x = y))
\]

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

A very simple lemma is a consequence of these definitions.

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

1. \( Q \Rightarrow (\neg_T (P, Q) = \neg P) \)
2. \( Q \land S \Rightarrow ((P, Q) \land_T (R, S) = P \land Q) \)

68
3. \( Q \land S \Rightarrow ((P, Q) \lor_T (R, S) = P \lor Q) \)

\[\]

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

### 6.3 First-order Theories

#### 6.3.1 Contexts for First-order Theories

We introduce a context theory \( \text{CXT} \) for our first-order theories, which will all be subtheories of \( \text{CXT} \). Its alphabet contains two observational variables:

\[
P \text{Shape} : \mathcal{P}((\mathbb{U}^\perp)^* \rightarrow \mathbb{B}^\perp) \\
F \text{Shape} : \mathcal{P}((\mathbb{U}^\perp)^* \rightarrow \mathbb{U}^\perp)
\]

and its signature is:

\[
\begin{align*}
= & : \mathbb{U}^\perp \times \mathbb{U}^\perp \rightarrow \mathbb{B}^\perp \\
\neg & : \mathbb{B}^\perp \rightarrow \mathbb{B}^\perp \\
\lor & : \mathbb{B}^\perp \times \mathbb{B}^\perp \rightarrow \mathbb{B}^\perp \\
\forall & : (\mathbb{U} \rightarrow \mathbb{B}^\perp) \rightarrow \mathbb{B}^\perp \\
\forall & : (\mathbb{U} \rightarrow \mathbb{B}^\perp) \rightarrow \mathbb{U}^\perp \\
\end{align*}
\]

\( P \text{Shape} \) describes all the possible denotations for the predicate symbols of this theory. Every denotation is a partial function from some number of parameters, each of which could be drawn from \( \mathbb{U} \) or could be undefined, to a boolean result, which could also be undefined. The purpose of \( P \text{Shape} \) is to constrain all the theory’s predicate symbols in a uniform way. \( F \text{Shape} \) does the same job as \( P \text{Shape} \), except that it describes the possible denotations of function symbols. The operators \( =, \neg, \lor \) give the syntax for equality, negation, and disjunction, respectively.

The \( \forall \) function takes as its argument a function \( \mathbb{U} \rightarrow \mathbb{B}^\perp \) that describes a binding for the universal quantifier that characterises the predicate that must be universally true. The function considers each element of its domain in turn and assigns to it one of the three logical values. The \( \forall \) function takes this binding function and decides whether the universally quantified predicate is true, false, or undefined. Notice that the binding function ranges only over defined values. This means that we are excluding logics where bound variables may be undefined, as is the case in LCF [11].

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

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

\[
\text{CTX}(P) = \\
P \land (\forall f : \mathbb{U} \rightarrow \mathbb{B}^\perp \bullet f \neq \emptyset \Rightarrow \forall_T(f) \in \text{dom } f^\perp)
\]
This requires that the definite description of a non-empty binding function returns either an undefined value or an element from the domain of the binding. We require this result in Lemma 6.3.3, where we prove that theories are closed under constructs over their signature.

**Example 6.3.1 (Context)** Consider a context $X_1$ with no predicate symbols and only monadic and dyadic function symbols. This is defined as:

$$X_1(P) = P \land (P\text{Shape} = \emptyset) \land (F\text{Shape} = (\mathbb{U}_1 \cup (\mathbb{U}_1)^2 \rightarrow \mathbb{U}_1))$$

$P$\text{Shape} and $F$\text{Shape} are used to add type information: we use them to restrict how predicate and function symbols behave, particularly, as we shall see later, with respect to undefinedness. □

### 6.3.2 First-order Theory

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

A partition $(\text{Var}, \text{Pred}, \text{Fun})$

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

The first healthiness condition requires that every variable is defined and has a value drawn from $\text{Dom}$:

$$DV(P) = P \land (\forall v : \text{Var} \bullet v \in \text{Dom})$$

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

$$DP(P) = P \land (\forall p : \text{Pred} \bullet p \in ((\text{Dom}^\perp)^{\rho(p)} \rightarrow \mathbb{B}^\perp) \cap \text{PShape})$$

$$DF(P) = P \land (\forall f : \text{Fun} \bullet f \in ((\text{Dom}^\perp)^{\rho(f)} \rightarrow \text{Dom}^\perp) \cap \text{FShape})$$

**Example 6.3.2 (First-order theory)** Consider a theory $T_1$ with context $X_1$ that has just a single function symbol for integer division. This is defined as:

$$T_1(P) = X_1(P)$$

$$\land \text{Var} = \emptyset$$

$$\land \text{Pred} = \emptyset$$

$$\land \text{Fun} = \{-/\}$$

$$\land \text{Dom} = \mathbb{N}$$

$$\land \rho = \{-/\, \mapsto \, 2\}$$

$$\land -/\in (\mathbb{N}^\perp \times \mathbb{N}^\perp \rightarrow \mathbb{N}^\perp) \cap \text{FShape}$$

□
6.3.3 Information-theoretic Ordering

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

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

$$a \sqsubseteq b \equiv (a \neq \bot) \Rightarrow (a = b)$$

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

$$x \sqsubseteq y \equiv \forall i : 1 \ldots k \bullet x_i \sqsubseteq y_i$$

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

$$f \sqsubseteq g \equiv (\text{dom } f = \text{dom } g) \land (\forall x : \text{dom } f \bullet f(x) \sqsubseteq g(x))$$

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

$$A \sqsubseteq_H B \equiv \forall a : A \bullet \exists b : B \bullet a \sqsubseteq b$$

These definitions are illustrated in the following set of examples.

**Example 6.3.3 (Ordering)**

1. Elements:

$$\bot \sqsubseteq 1$$
$$1 \sqsubseteq 1$$
$$\neg (1 \sqsubseteq 2)$$

2. Tuples:

$$(0, \bot, 2) \sqsubseteq (0, 1, 2)$$
$$() \sqsubseteq ()$$
$$\neg ((1, 2) \sqsubseteq (1, 2))$$
$$\neg ((1, 2) \sqsubseteq (1, 2))$$

3. Functions:

$$\lambda x, y : \mathbb{N} \bullet \bot \sqsubseteq (y = 0) \triangleright x/y$$

$$\not\sqsubseteq (\lambda x, y : \mathbb{N} \bullet 0 \sqsubseteq (y = 0) \triangleright x/y)$$

$$\neg ((\lambda x, y : \mathbb{N} \bullet \bot \sqsubseteq (y = 0) \triangleright x/y) \sqsubseteq (\lambda x, y : \mathbb{N} \bullet \bot \sqsubseteq (y = 0) \triangleright x/y))$$

$$\lambda n : \mathbb{N} \bullet \bot \sqsubseteq (n \mod 2 = 0) \triangleright n \not\sqsubseteq (\lambda n : \mathbb{N} \bullet \bot \sqsubseteq (n \mod 2 = 0) \triangleright n)$$
4. Sets of functions:

\[
\{ (\lambda x, y : \mathbb{N} \cdot \bot \triangleleft (y = 0) \triangleright x/y), \\
(\lambda n : \mathbb{N} \cdot \bot \triangleleft (n \mod 2 = 0) \triangleright n), \\
(\lambda n : \mathbb{N} \cdot n) \}
\]

\[\subseteq_{H}\]

\[
\{ (\lambda x, y : \mathbb{N} \cdot 0 \triangleleft (y = 0) \triangleright x/y), \\
(\lambda n : \mathbb{N} \cdot n) \}
\]

\[\Box\]

We further generalise the ordering by lifting it to contexts.

**Definition 6.3.2 (Ordering on contexts)**

\[
S \subseteq_{H} T = \forall P : S, Q : T \cdot P \subseteq_{H} Q
\]

where

\[
P \subseteq_{H} Q = \\
P_{\text{Shape}} \subseteq_{H} P_{\text{Shape}} \\
\wedge F_{\text{Shape}} \subseteq_{H} F_{\text{Shape}} \\
\wedge (=_{S}) \subseteq (=_{T}) \\
\wedge (\neg_{S}) \subseteq (\neg_{T}) \\
\wedge (\forall_{S}) \subseteq (\forall_{T}) \\
\wedge (\exists_{S}) \subseteq (\exists_{T}) \\
\wedge (\Box_{S}) \subseteq (\Box_{T})
\]

**Example 6.3.4 (Subtheory)** Consider \(X_2\), a subtheory of \(X_1\), where the following holds:

\[
\forall f : F_{\text{Shape}_{X_1}} \cdot \text{zero} \circ f \in F_{\text{Shape}_{X_2}}
\]

and where the total function zero is defined:

\[
\text{zero}(x) \triangleq (0 \triangleleft (x = \bot) \triangleright x)
\]

All other components remain unchanged. Then \(P_{X_1} \subseteq_{H} P_{X_2}\), since

\[
f \subseteq \text{zero} \circ f = (\text{dom} f = \text{dom}(\text{zero} \circ f)) \wedge \forall x : \text{dom} f \cdot f(x) \subseteq \text{zero} \circ f(x)
\]

and so we have \(F_{\text{Shape}_{X_1}} \subseteq_{H} F_{\text{Shape}_{X_2}}\). \(\Box\)

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

### 6.3.4 Strictness

The notion of strictness is a familiar one from the definition of programming languages. A function \(f\) is strict if \(f(\bot) = \bot\), and it is usually used to denote that a function loops forever or performs an illegal operation, such as division by zero. We can interpret a strict function operationally as one that always evaluates all of its arguments. A restricted notion considers functions that are strict in one or more arguments.
Definition 6.3.3 (Strictness) Function \( f : (X^\perp)^{\rho(f)} \rightarrow Y^\perp \) is strict if, whenever at least one of its arguments is undefined, then the result is undefined:

\[
\text{strict}(f) = \forall x : (X^\perp)^{\rho(f)} \bullet (\exists i : 1 \ldots \rho(f) \bullet (x_i = \perp)) \Rightarrow (f(x) = \perp)
\]

\[\square\]

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

\[
\_ \times \_ : \mathbb{N}^\perp \times \mathbb{N}^\perp \rightarrow \mathbb{N}^\perp
\]

\[
x \times y = \perp \mathrel{\lor} (x = \perp) \mathrel{\lor} (y = \perp) \mathrel{\lor} x \times y
\]

\[\square\]

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

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

\[
\text{strict}(T) = \{ P : T \bullet \text{strict}(P) \}
\]

where \( \text{strict}(P) = \exists P\text{Shape}_0, F\text{Shape}_0 \bullet 
\]

\[
P\text{Shape} = \{ p : P\text{Shape}_0 \mid \text{strict}(p) \}
\]

\[
\land F\text{Shape} = \{ f : F\text{Shape}_0 \mid \text{strict}(f) \}
\]

\[
\land P[P\text{Shape}_0, F\text{Shape}_0/P\text{Shape}, F\text{Shape}]
\]

\[\square\]

6.3.5 Definiteness

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

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

\[
\text{definite}(f) = \forall x : (X^\perp)^{\rho(f)} \bullet (f(x) = \perp) \Rightarrow (\exists i : 1 \ldots \rho(f) \bullet (x_i = \perp))
\]

\[\square\]

Example 6.3.6 (Definite function) \( \_ \times \_ \) is definite. \[\square\]

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

Definition 6.3.6 (Definite Contexts) Making a context definite:

\[
\text{definite}(T) = \{ P : T \bullet \text{definite}(P) \}
\]

where \( \text{definite}(P) = 
\]

\[
P\text{Shape} = \{ p : P\text{Shape}_0 \mid \text{definite}(p) \}
\]

\[
\land F\text{Shape} = \{ f : F\text{Shape}_0 \mid \text{definite}(f) \}
\]

\[
\land P[P\text{Shape}_0, F\text{Shape}_0/P\text{Shape}, F\text{Shape}]
\]

\[\square\]
6.3.6 Monotonicity

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

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

\[
\text{monotonic}(f) = \forall x_1, x_2 : (X^\perp)^{\rho(f)} \bullet x_1 \sqsubseteq x_2 \Rightarrow f(x_1) \sqsubseteq f(x_2)
\]

\( \Box \)

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

<table>
<thead>
<tr>
<th>( \neg_T )</th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bot )</td>
<td>( \bot )</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

\( \Box \)

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

**Definition 6.3.8 (Monotonic Contexts)** \( T \) is a monotonic context:

\[
\text{monotonic}(T) = \forall P : T \bullet \text{monotonic}(P)
\]

where \( \text{monotonic}(P) =
\]

\[
(\forall p : \text{Pred}_T \bullet \text{monotonic}(p)) \\
\land (\forall f : \text{Fun}_T \bullet \text{monotonic}(f)) \\
\land \text{monotonic}(=_T) \\
\land \text{monotonic}(\neg_T) \\
\land \text{monotonic}(\lor_T) \\
\land \text{monotonic}(\forall_T) \\
\land \text{monotonic}(\exists_T)
\]

\( \Box \)

The following simple lemma is useful.

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

6.3.7 Comparing FOTs

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

**Definition 6.3.9 (Comparing FOTs)** Comparing FOTs \( U \) and \( V \): for \( P : U \) and \( Q : V \)

\[
P \sqsubseteq_H Q = \text{Dom}_q = \text{Dom}_v \land \text{Pred}_q \sqsubseteq_H \text{Pred}_v \land \text{Fun}_q \sqsubseteq_H \text{Fun}_v
\]

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

**Lemma 6.3.2 (Models)** Suppose that we have two CXTs \( S \) and \( T \), where \( S \subseteq_H T \). Suppose further that \( U \) is a FOT extending \( S \). Then there is a FOT \( V \) extending \( T \) such that \( U \subseteq V \).

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

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

<table>
<thead>
<tr>
<th></th>
<th>( S )</th>
<th>( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>PShape</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>FShape</td>
<td>strict(( U^\perp \rightarrow U^\perp ))</td>
<td>( U^\perp \rightarrow U )</td>
</tr>
<tr>
<td>Dom</td>
<td>{0, 1}</td>
<td>{0, 1}</td>
</tr>
<tr>
<td>( \rho )</td>
<td>{ inc \mapsto 1 }</td>
<td>{ inc \mapsto 1 }</td>
</tr>
<tr>
<td>( A )</td>
<td>( \text{inc}(\perp) = \perp )</td>
<td>( \text{inc}(\perp) = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \text{inc}(0) = 1 )</td>
<td>( \text{inc}(0) = 1 )</td>
</tr>
<tr>
<td></td>
<td>( \text{inc}(1) = \perp )</td>
<td>( \text{inc}(1) = 1 )</td>
</tr>
</tbody>
</table>

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

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

\[
P \Rightarrow e \in \text{Dom}_U^\perp
\]

This lemma is proved by syntactic induction.

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

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

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

The induction hypothesis is that $x_s \subseteq x_T$.

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

\[
(f(x))_u = f_u(x_u) \quad \{ \text{hypothesis } x_u \subseteq x_v + S \text{ monotonic, and so } f_u \text{ is monotonic} \} \\
\subseteq f_u(x_v) \quad \{ \text{assumption: } P_u \subseteq Q_v, \text{ and so } Fun_u \subseteq Fun_v \text{ and so } f_u \subseteq f_v \} \\
\subseteq f_v(x_v) \quad \{ \text{interpretation} \} \\
= (f(x))_v
\]

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

\[
(f(x))_u = f_u(x_u) \quad \{ \text{interpretation} \} \\
\subseteq f_u(x_v) \quad \{ \text{assumption: } P_s \subseteq P_T \} \\
\subseteq f_v(x_v) \quad \{ \text{hypothesis } + V \text{ monotonic} \} \\
\subseteq (f(x))_v \quad \{ \text{interpretation} \}
\]

\[
\square
\]

6.4 Specific First-order Theories

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

6.4.1 Strict Logic

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

\[
S1(P) = \text{strict}(P)
\]

Next, equality is strict:

\[
(=_s(x, y) = \bot) \land (x = \bot) \lor (y = \bot) \lor (=_s(x, y) = (x = y))
\]

76
Recall Example 6.2.6 for an explanation of the definedness of this definition. If either argument is undefined, then the equality is undefined; otherwise, strict equality depends on the underlying UTP equality.

Definite description is strict:

$$(\forall s(f) = x) \iff \bot \notin \text{ran } f \land (\text{dom}(f \triangleright \{\text{true}\}) = \{x\}) \triangleright (\forall s(f) = \bot)$$

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

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

$$(\forall s(f) = \bot) \iff \bot \in \text{ran } f \triangleright (\forall s(f) = (\text{ran } f = \{\text{true}\}))$$

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

$$\neg s(P) = \neg P$$

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

$$\lor s (P, Q) = P \lor Q$$

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

<table>
<thead>
<tr>
<th>$\neg_s$</th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\lor_s$</th>
<th>true</th>
<th>$\bot$</th>
<th>false</th>
<th>$\bot$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>$\bot$</td>
<td>true</td>
<td>$\bot$</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$\bot$</td>
<td>true</td>
<td>false</td>
<td>$\bot$</td>
</tr>
</tbody>
</table>

### 6.4.2 Kleene System

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

$$K1(P) = P \land (\forall f : PShape_{k} \cup FShape_{k} \cdot \text{monotonic}(f))$$

Equality and definite are both strict:

$$(=_{k}) = (=_{s})$$
$$s_{k} = (s_{s})$$

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

$$(\forall s(f) = \text{false}) \iff \text{false} \in \text{ran } f \triangleright (\forall s(f) = \bot)$$

$$(\forall s(f) = \text{true}) \iff (\text{ran } f = \{\text{true}\}) \triangleright (\forall s(f) = \bot))$$

77
Negation is strict:
\[ \neg_k = \neg_s \]

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

\[
\begin{align*}
((\forall_k (P, Q) = true) &\lor (P = true) \land (Q = true)) \\
((\forall_k (P, Q) = false) &\lor (P = false) \land (Q = false)) \\
((\forall_k (P, Q) = \perp))
\end{align*}
\]

<table>
<thead>
<tr>
<th>( \forall_k )</th>
<th>\text{true}</th>
<th>\perp</th>
<th>\text{false}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{true}</td>
<td>\text{true}</td>
<td>\perp</td>
<td>\text{true}</td>
</tr>
<tr>
<td>\perp</td>
<td>\text{true}</td>
<td>\perp</td>
<td>\perp</td>
</tr>
<tr>
<td>\text{false}</td>
<td>\text{true}</td>
<td>\perp</td>
<td>\text{false}</td>
</tr>
</tbody>
</table>

### 6.4.3 McCarthy System

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

\[ M1 = K1 \]

Equality and definite description are both strict.

\[
(=_m) = (=_k) \\
\iota_m = \iota_k
\]

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

\[ \forall_m = \forall_k \]

Negation is the same as Kleene.

\[ \neg_m = \neg_k \]

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

\[
\begin{align*}
((\forall_m (P, Q) = true) &\lor (P = true) \land (Q = true)) \\
((\forall_m (P, Q) = \perp)) &\lor (P = \perp) \land (Q = \perp) \lor ((\forall_m (P, Q) = false))
\end{align*}
\]

<table>
<thead>
<tr>
<th>( \forall_m )</th>
<th>\text{true}</th>
<th>\perp</th>
<th>\text{false}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{true}</td>
<td>\text{true}</td>
<td>\perp</td>
<td>\text{true}</td>
</tr>
<tr>
<td>\perp</td>
<td>\perp</td>
<td>\perp</td>
<td>\perp</td>
</tr>
<tr>
<td>\text{false}</td>
<td>\text{true}</td>
<td>\perp</td>
<td>\text{false}</td>
</tr>
</tbody>
</table>

All three systems are monotonic.
Lemma 6.4.1 (strict-Kleene-McCarthy monotonicity)

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

There is an interesting definedness order between the three systems:

Lemma 6.4.2 (Strict-McCarthy-Kleene ordering) for $\rho_s = \rho_m = \rho_k$ and $\text{Dom}_s = \text{Dom}_m = \text{Dom}_k$

$\text{FOT}_s \subseteq \text{FOT}_m \subseteq \text{FOT}_k$

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

6.5 Guard Systems

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

6.5.1 Validity

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

$T \models P$ if $P$ is valid for all $U$, $T \subseteq U$ implies $P_U = \text{true}$

6.5.2 Guards

Suppose that $c$ is a construct. Then predicate $G$ is a guard for $c$ in $\mathbf{CXT}_T$ (denoted by $G \bowtie_T P$) iff for every $\text{FOT}_v$ that extends $\mathbf{CXT}_T$ we have

1. $(G_v \neq \bot)$
2. $(G_v = \text{true}) \Rightarrow (c_v \neq \bot)$

$G$ is a tight guard if we also have

3. $(G_v = \text{false}) \Rightarrow (c_v = \bot)$

Now we are ready to state and prove our main result, which is due originally to Saaltink.
Theorem 6.5.1 (Main Theorem (Saaltink)) Suppose that $\text{CXT}_T \subseteq \text{CXT}_S$, that either one is monotonic, and that $G$ is a guard for $P$ in $\text{CXT}_S$. Then, if $(T \models G)$ and $(T \models P)$, we have that $(S \models P)$. □

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

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

Proof 6.5.1 (Main Theorem)

1. From the Models Lemma 6.3.2, since $\text{CXT}_T \subseteq \text{CXT}_S$ and $\text{FOT}_S$ extends $\text{CXT}_S$, then there exists $\text{FOT}_S$ that extends $\text{CXT}_T$ and for which we have $\text{FOT}_S \subseteq \text{FOT}_T$.

2. Since $G \sim_S P$, know that $(G_\uparrow \neq \bot) \land ((G_\uparrow = \text{true}) \Rightarrow (P_\uparrow \neq \bot))$ from the definition of a guard.

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

4. We now repeat this argument for $P$. By construct monotonicity, $(S$ monotonic), we have $P_\uparrow \subseteq P_\downarrow$, therefore $(P_\uparrow = P_\downarrow)$. But $T \upmodels P$, so $(P_\downarrow = \text{true})$ and therefore $(P_\uparrow = \text{true})$.

6.5.3 Definedness Guards

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

Example 6.5.2 (Definedness guard)

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

□

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

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

$$\mathcal{D} \Phi \equiv \Phi \land \mathcal{D}(\Phi)$$
If we define a system of guards for every construct in our language, then we can use this system inductively to generate verification conditions for the definedness of all constructs. In the next section we demonstrate this for the case of the definite McCarthy system.

### 6.5.4 Guards for Definite McCarthy System

\[
D_m x = \text{true} \\
D_m(p(e)) = \forall i : 1..\rho(P) \bullet D_m e_i \\
D_m(f(e)) = \forall i : 1..\rho(f) \bullet D_m e_i \\
D_m(e_1 = e_2) = D_m e_1 \land D_m e_2 \\
D_m(\neg P) = D_m P \\
D_m(P \lor Q) = D_m P \land (P \lor D_m Q) \\
D_m(\forall x \bullet P) = \forall x \bullet D_m P \\
D_m(\exists x \bullet P) = (\forall x \bullet D_m P) \land (\exists_1 x \bullet P)
\]

**Theorem 6.5.2 (McCarthy guards)** If \( c \) is a construct, then \( D_m(c) \) is a guard for \( c \) in \( \text{definite}(T) \), and a tight guard for \( c \) in \( \text{strict \ definite}(T) \). \( \square \)

### 6.6 Summary

#### 6.6.1 Contribution

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

#### 6.6.2 Future work

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


82


Appendix A

Proofs

A.1 Well Foundedness

Theorem 4.2.1 (Well foundedness) Every CML operator preserves $T_1$-healthiness.

Proof A.1.1 By induction on syntax.

1. Deadlock

\[
\begin{align*}
STOP[\langle \rangle/\text{tt}\rangle] \\
= \{ STOP \} \\
(T_0(\text{trace}(\text{tt}\rangle) \in \text{tock}^*))[\langle \rangle/\text{tt}\rangle] \\
= \{ T_0, \text{substitution} \} \\
\text{trace}(\langle \rangle) \in \text{tock}^* \land \langle \rangle \in \text{timedTrace} \\
= \{ \langle \rangle \in \text{timedTrace} \} \\
\text{trace}(\langle \rangle) \in \text{tock}^* \\
= \{ \text{trace}(\langle \rangle) = \langle \rangle \} \\
\langle \rangle \in \text{tock}^* \\
= \{ \text{Kleene closure} \} \\
\text{true}
\end{align*}
\]
2. **Prefix**

\[(a \rightarrow P)[\{/tt']
\]

\[
= \{ \text{prefix} \}
\]

\[
(\begin{array}{l}
\text{a} \notin \text{refusals}(tt') \\
\text{\langle trace(tt') \in tock \rangle} \\
\text{\langle a = \text{head(trace(idlesuffix(tt'))) \land} \\
\text{\langle a} \notin \text{refusals(idleprefix(tt')) \land} \\
\text{\langle P[\text{tail(idlesuffix(tt'))/tt']} \rangle}
\end{array}
\)
\]

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

\[
T0(\text{a} \notin \text{refusals}(tt') \land \text{trace(tt') \in tock})[{\langle/ tt'}]
\]

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

\[
T0(\text{a} \notin \text{refusals}()) \land () \in tock
\]

\[
= \{ T0 \}
\]

\[
a \notin \text{refusals}() \land \text{trace}() \in tock \land () \in \text{TimedTrace}
\]

\[
= \{ () \in \text{TimedTrace} \}
\]

\[
\text{trace}() \in tock \land a \notin \text{refusals}()
\]

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

\[
\text{trace}() \in tock \land a \notin \text{refusals}()
\]

\[
= \{ \text{trace}() = () \}
\]

\[
() \in tock \land a \notin \text{refusals}()
\]

\[
= \{ \text{refusals}() = \emptyset \}
\]

\[
() \in tock \land a \notin \emptyset
\]

\[
= \{ \text{empty set} \}
\]

\[
() \in tock \land \text{true}
\]

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

\[
() \in tock
\]

\[
= \{ \text{Kleene closure} \}
\]

\[
\text{true}
\]

3. **Internal choice** Suppose \(P[\{/ tt']\)

\[
(P \cap Q)[\{/ tt']
\]

\[
= \{ \text{nondeterministic choice} \}
\]

\[
(P \lor Q)[\{/ tt']
\]

\[
= \{ \text{substitution} \}
\]

\[
P[\{/ tt'] \lor Q[\{/ tt']
\]

\[
= \{ \text{assumption: } P[\{/ tt'] \}
\]

\[
\text{true}
\]

4. **External choice** Suppose both \(P[\{/ tt']\) and \(Q[\{/ tt']\)

\[
(P \Box Q)[\{/ tt']
\]
D23.3-1 - CML Definition 2 (Public)

\[
\begin{align*}
\text{Definition 2 (Public)} &= \{ \text{external choice} \} \\
\&= \{ \text{idleprefix}(\text{tt}) \} \wedge (P \vee Q) \}
\]

\[
\begin{align*}
(P \wedge Q)[\text{idleprefix}(\text{tt})/\text{tt}'] &\wedge (P \vee Q)[\langle \rangle/\text{tt}'] \\
&= \{ \text{propositional calculus} \} \\
\end{align*}
\]

\[
\begin{align*}
P[\langle \rangle/\text{tt}'] &\wedge Q[\langle \rangle/\text{tt}'] \vee Q[\langle \rangle/\text{tt}'] \\
&= \{ \text{propositional calculus} \} \\
P[\langle \rangle/\text{tt}'] &\wedge Q[\langle \rangle/\text{tt}'] \\
&= \{ \text{assumptions: } P[\langle \rangle/\text{tt}'] \text{ and } Q[\langle \rangle/\text{tt}'] \} \\
\text{true} &= \{ \text{true} \}
\end{align*}
\]

5. **Parallel composition**

\[
(P \parallel_A Q)[\langle \rangle/\text{tt}'] \\
= \{ \text{parallel composition} \} \\
(\exists t, u \bullet P[t/\text{tt}'] \wedge Q[u/\text{tt}'] \wedge \text{tt'} \in (t \parallel_A u))[\langle \rangle/\text{tt}'] \\
= \{ \text{substitution} \} \\
\exists t, u \bullet P[t/\text{tt}'] \wedge Q[u/\text{tt}'] \wedge \langle \rangle \in (t \parallel_A u) \\
\quad \iff \{ \text{predicate calculus} \} \\
P[\langle \rangle/\text{tt}'] \wedge Q[\langle \rangle/\text{tt}'] \wedge \langle \rangle \in (t \parallel_A \langle \rangle) \\
= \{ \text{parallel traces: } \langle \rangle \in (t \parallel_A \langle \rangle) \} \\
P[\langle \rangle/\text{tt}'] &\wedge Q[\langle \rangle/\text{tt}'] \\
= \{ \text{assumptions: } P[\langle \rangle/\text{tt}'] \text{ and } Q[\langle \rangle/\text{tt}'] \} \\
\text{true} &= \{ \text{true} \}
\]

6. **Hiding**

\[
(P \setminus A)[\langle \rangle/\text{tt}'] \\
= \{ \text{hiding} \} \\
(\exists t \bullet P[t/\text{tt}'] \wedge \text{urgent } t \wedge (\text{tt'} = (t \setminus A)))[\langle \rangle/\text{tt}'] \\
= \{ \text{substitution} \} \\
\exists t \bullet P[t/\text{tt}'] \wedge \text{urgent } t \wedge (\langle \rangle = (t \setminus A)) \\
\quad \iff \{ \text{predicate calculus} \} \\
P[\langle \rangle/\text{tt}'] &\wedge \text{urgent } \langle \rangle \wedge (\langle \rangle = (t \setminus A)) \\
= \{ \text{assumption: } P[\langle \rangle/\text{tt}'] \} \\
A \text{ urgent } \langle \rangle \wedge (\langle \rangle = (t \setminus A)) \\
= \{ \text{urgency: } A \text{ urgent } \langle \rangle \} \\
(\langle \rangle = (t \setminus A)) \\
= \{ \text{trace hiding: } (\langle \rangle \setminus A) = \langle \rangle \} \\
(\langle \rangle = \langle \rangle)
\]

87
= \{ \textit{equality} \}
true

7. Timing W.T.P. $[P[\emptyset/\tau t'] \land Q[\emptyset/\tau t'] \Rightarrow (P \overset{n}{\Rightarrow} Q)[\emptyset/\tau t']]$

$(P \overset{n}{\Rightarrow} Q)[\emptyset/\tau t']$

= \{ \textit{timeout semantics} \}

\begin{align*}
(\exists u \cdot u \leq \emptyset) \land (\text{trace}(u) = \text{tock}^n) \land P[u/\tau t'] \land Q[\emptyset - u/\tau t'] \\
\text{\textit{tock}^n} \leq \text{trace}(\emptyset) \Rightarrow P[\emptyset/\tau t']
\end{align*}

= \{ \textit{assumption: } P[\emptyset/\tau t'] \}

\begin{align*}
(\exists u \cdot u \leq \emptyset) \land (\text{trace}(u) = \text{tock}^n) \land P[u/\tau t'] \land Q[\emptyset - u/\tau t'] \\
\text{\textit{tock}^n} \leq \text{trace}(\emptyset) \Rightarrow \textit{true}
\end{align*}

= \{ \textit{conditional: } (P \circ b \overset{\textit{true}}{\Rightarrow}) = (b \Rightarrow P) \}

\text{\textit{tock}^n} \leq \text{trace}(\emptyset) \Rightarrow (\exists u \cdot u \leq \emptyset) \land (\text{trace}(u) = \text{tock}^n) \land P[u/\tau t'] \land Q[\emptyset - u/\tau t']

= \{ \textit{trace extraction: } \text{trace}(\emptyset) = \emptyset \}

\textit{tock}^n \leq \emptyset \Rightarrow (\exists u \cdot u \leq \emptyset) \land (\text{trace}(u) = \text{tock}^n) \land P[u/\tau t'] \land Q[\emptyset - u/\tau t']

= \{ \textit{event repetition: } \text{tock}^n \leq \emptyset = (n = 0) \}

n = 0 \Rightarrow (\exists u \cdot u \leq \emptyset) \land (\text{trace}(u) = \text{tock}^n) \land P[u/\tau t'] \land Q[\emptyset - u/\tau t']

= \{ \textit{precedence: } u \leq \emptyset = (u = \emptyset) \}

n = 0 \Rightarrow (\exists u \cdot (u = \emptyset) \land (\text{trace}(u) = \text{tock}^n) \land P[u/\tau t'] \land Q[\emptyset - u/\tau t']

= \{ \textit{predicate calculus: one-point rule} \}

n = 0 \Rightarrow (\text{trace}(\emptyset) = \text{tock}^n) \land P[\emptyset/\tau t'] \land Q[\emptyset - \emptyset]

= \{ \textit{assumption: } P[\emptyset/\tau t'] \}

n = 0 \Rightarrow (\text{trace}(\emptyset) = \text{tock}^n) \land Q[\emptyset - \emptyset]

= \{ \textit{sequence difference: } s - \emptyset = s \}

n = 0 \Rightarrow (\text{trace}(\emptyset) = \text{tock}^n) \land Q[\emptyset/\tau t']

= \{ \textit{assumption: } Q[\emptyset/\tau t'] \}

n = 0 \Rightarrow (\text{trace}(\emptyset) = \text{tock}^n)

= \{ \textit{trace extraction: } \text{trace}(\emptyset) = \emptyset \}

n = 0 \Rightarrow (\emptyset = \text{tock}^n)

= \{ \textit{Leibniz} \}

n = 0 \Rightarrow (\emptyset = \text{tock}^0)

= \{ \textit{event repetition: } a^0 = \emptyset \}

n = 0 \Rightarrow (\emptyset = \emptyset)

= \{ \textit{equality} \}

n = 0 \Rightarrow \textit{true}

= \{ \textit{propositional calculus} \}
true

8. **Recursion** $T_1$ can be written as the conjunctive idempotent $T_1(P) = P \land P[\langle \rangle / tt']$. Recursion therefore satisfies $T_1$, as demonstrated in [12].

### A.1.1 Prefix Closure

**Theorem 4.2.2 (Prefix closure)** Every CML operator preserves $T_2$-healthiness.

**Proof A.1.2** By induction on program syntax.

1. **Deadlock**
   
   Assume $STOP \land t \leq tt'$
   
   $t \leq tt' \Rightarrow trace(t) \leq trace(tt')$
   
   $STOP[t/\langle tt' \rangle]$
   
   $= \{ STOP \}$
   
   $(T_0(trace(tt') \in tock^*)[t/\langle tt' \rangle]$
   
   $= \{ substitution \}$
   
   $trace(t) \in tock^* \land t \in timedTrace$
   
   $= \{ assumption: t \leq tt'; (s \leq t) \land t \in timedTrace \Rightarrow s \in timedTrace \}$
   
   $trace(t) \in tock^* \land tt' \in timedTrace$
   
   $= \{ STOP \}$
   
   $STOP$

2. **Prefix W.T.P.**
   
   $(a \rightarrow P) \land t \leq tt' \Rightarrow (a \rightarrow P)[t/\langle tt' \rangle]$

   given
   
   $[P \land t \leq tt' \Rightarrow P[t/\langle tt' \rangle]]$

   $(a \rightarrow P)[t/\langle tt' \rangle]$

   $= \{ prefix \}$

   $(a \notin refusals(tt')$

   $< trace(tt') \in tock^*>$

   $T_0$

   $a = head(trace(idlesuffix(tt'))) \land$

   $a \notin refusals(idleprefix(tt')) \land$

   $P[tail(idlesuffix(tt'))/tt']$

   $= \{ T_0 \}$
\[
\begin{align*}
\begin{cases}
    a \notin \text{refusals}(tt') \land tt' \in \text{TimedTrace} \\
    \sim \text{trace}(tt') \in \text{tock}^* \\
    a = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \land \\
    a \notin \text{refusals}(\text{idleprefix}(tt')) \land \\
    P[\text{tail}(\text{idlesuffix}(tt'))/tt'] \land \\
    tt' \in \text{TimedTrace}
\end{cases} & [t/t'] \\
= \{ \text{substitution} \}
\end{align*}
\]
\[
\begin{align*}
\begin{cases}
    a \notin \text{refusals}(t) \land t \in \text{TimedTrace} \\
    \sim \text{trace}(t) \in \text{tock}^* \\
    a = \text{head}(\text{trace}(\text{idlesuffix}(t))) \\
    a \notin \text{refusals}(\text{idleprefix}(t)) \land \\
    P[\text{tail}(\text{idlesuffix}(t))/tt'] \land \\
    t \in \text{TimedTrace}
\end{cases} & [t/t'] \\
= \{ \text{conditional} \}
\end{align*}
\]
\[
\begin{align*}
\begin{cases}
    \text{trace}(t) \in \text{tock}^* \land a \notin \text{refusals}(t) \land t \in \text{TimedTrace} \\
\end{cases} & \lor \\
\begin{cases}
    \neg (\text{trace}(t) \in \text{tock}^*) \land a = \text{head}(\text{trace}(\text{idlesuffix}(t))) \\
    \land a \notin \text{refusals}(\text{idleprefix}(t)) \land \\
    P[\text{tail}(\text{idlesuffix}(t))/tt'] \land t \in \text{TimedTrace} \\
\end{cases} & \neg (\text{trace}(t) \in \text{tock}^*) \\
= \{ \text{T1}(P) \} \\
\begin{cases}
    \text{trace}(t) \in \text{tock}^* \land a \notin \text{refusals}(t) \land t \in \text{TimedTrace} \land P[\emptyset/tt'] \\
\end{cases} & \lor \\
\begin{cases}
    \neg (\text{trace}(t) \in \text{tock}^*) \land a = \text{head}(\text{trace}(\text{idlesuffix}(t))) \\
    \land a \notin \text{refusals}(\text{idleprefix}(t)) \land \\
    P[\text{tail}(\text{idlesuffix}(t))/tt'] \land t \in \text{TimedTrace} \\
\end{cases} & \neg (\text{trace}(t) \in \text{tock}^*) \\
= \{ \text{T1}(P) \} \\
\begin{cases}
    \text{trace}(t) \in \text{tock}^* \land a \notin \text{refusals}(t) \land t \in \text{TimedTrace} \land P[\emptyset/tt'] \\
\end{cases} & \lor \\
\begin{cases}
    a = \text{head}(\text{trace}(\text{idlesuffix}(t))) \land a \notin \text{refusals}(\text{idleprefix}(t)) \land \\
    P[\text{tail}(\text{idlesuffix}(t))/tt'] \land t \in \text{TimedTrace} \\
\end{cases} & \neg (\text{trace}(tt') \in \text{tock}^*) \\
= \{ \text{propositional calculus} \}
\end{align*}
\]
\[
\begin{align*}
\begin{cases}
    \text{trace}(t) \in \text{tock}^* \land a \notin \text{refusals}(t) \land t \in \text{TimedTrace} \land P[\emptyset/tt'] \\
\end{cases} & \lor \\
\begin{cases}
    \neg (\text{trace}(tt') \in \text{tock}^*) \\
\end{cases} & \lor
\end{align*}
\]
\(\neg (\text{trace}(tt) \in \text{tock}^*) \land a = \text{head}(\text{trace}(\text{idlesuffix}(t))) \land a \notin \text{refusals}(\text{idleprefix}(t)) \land P[\text{tail}(\text{idlesuffix}(t))/tt] \land t \in \text{TimedTrace} \)

\(= \{ \text{trace}(t) \in \text{tock}^* \land a \notin \text{refusals}(t) \Rightarrow \} \)

\(\{ a \notin \text{refusals}(\text{idleprefix}(t)) \land \text{tail}(\text{idlesuffix}(t)) = () \} \)

\(\text{trace}(t) \in \text{tock}^* \land a \notin \text{refusals}(t) \land t \in \text{TimedTrace} \)

\(\vee\)

\(\neg (\text{trace}(tt) \in \text{tock}^*) \land \text{trace}(t) \in \text{tock}^* \land a \notin \text{refusals}(\text{idleprefix}(t)) \land P[\text{tail}(\text{idlesuffix}(t))/tt] \land t \in \text{TimedTrace} \)

\(\neg (\text{trace}(tt) \in \text{tock}^*) \land a = \text{head}(\text{trace}(\text{idlesuffix}(t))) \land a \notin \text{refusals}(\text{idleprefix}(t)) \land P[\text{tail}(\text{idlesuffix}(t))/tt] \land t \in \text{TimedTrace} \)

\(\iff \{ t \leq tt \land a = \text{head}(\text{trace}(\text{idlesuffix}(tt))) \Rightarrow \} \)

\(\{ \text{trace}(t) \in \text{tock}^* \land a = \text{head}(\text{trace}(\text{idlesuffix}(t))) \} \)

\(\text{trace}(t) \in \text{tock}^* \land a \notin \text{refusals}(t) \land t \in \text{TimedTrace} \)

\(\vee\)

\(\neg (\text{trace}(tt) \in \text{tock}^*) \land a = \text{head}(\text{trace}(\text{idlesuffix}(tt))) \land a \notin \text{refusals}(\text{idleprefix}(t)) \land P[\text{tail}(\text{idlesuffix}(t))/tt'] \land t \in \text{TimedTrace} \)

\(\{ \text{T2}(P); \text{tail}(\text{idlesuffix}(t)) \leq \text{tail}(\text{idlesuffix}(tt')) \} \)

\(\text{trace}(t) \in \text{tock}^* \land a \notin \text{refusals}(t) \land t \in \text{TimedTrace} \)

\(\vee\)

\(\neg (\text{trace}(tt) \in \text{tock}^*) \land a = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \land a \notin \text{refusals}(\text{idleprefix}(tt')) \land P[\text{tail}(\text{idlesuffix}(tt))/tt'] \land t \in \text{TimedTrace} \)

\(\{ t \leq tt \land a \notin \text{refusals}(\text{idleprefix}(tt')) \Rightarrow a \notin \text{refusals}(\text{idleprefix}(t)) \} \)

\(\text{trace}(t) \in \text{tock}^* \land a \notin \text{refusals}(t) \land t \in \text{TimedTrace} \)

\(\vee\)

\(\neg (\text{trace}(tt) \in \text{tock}^*) \land a = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \land a \notin \text{refusals}(\text{idleprefix}(tt')) \land P[\text{tail}(\text{idlesuffix}(tt))/tt'] \land t \in \text{TimedTrace} \)

\(\{ t \leq tt \land tt' \in \text{TimedTrace} \Rightarrow t \in \text{timedTrace} \} \)

\(\text{trace}(t) \in \text{tock}^* \land a \notin \text{refusals}(t) \land tt' \in \text{TimedTrace} \)
\[ \forall \\
¬ (\text{trace}(tt') \in \text{tock}^*) \land a = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \\
\land a \notin \text{refusals}(\text{idleprefix}(tt')) \land P[\text{tail}(\text{idlesuffix}(tt'))/tt'] \\
\land tt' \in \text{TimedTrace} \\
= \{ t \leq tt \land a \notin \text{refusals}(tt') \Rightarrow a \notin \text{refusals}(t) \} \\
\text{trace}(t) \in \text{tock}^* \land a \notin \text{refusals}(tt') \land tt' \in \text{TimedTrace} \\
\forall \\
¬ (\text{trace}(tt') \in \text{tock}^*) \land a = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \\
\land a \notin \text{refusals}(\text{idleprefix}(tt')) \land P[\text{tail}(\text{idlesuffix}(tt'))/tt'] \\
\land tt' \in \text{TimedTrace} \\
= \{ \text{conditional} \} \\
a \notin \text{refusals}(tt') \land tt' \in \text{TimedTrace} \\
\lhd \text{trace}(tt') \in \text{tock}^* \rhd \\
a = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \land \\
a \notin \text{refusals}(\text{idleprefix}(tt')) \land \\
P[\text{tail}(\text{idlesuffix}(tt'))/tt'] \land \\
tt' \in \text{TimedTrace} \\
= \{ \text{T0} \} \\
\begin{align*}
\text{T0} & \left( a \notin \text{refusals}(tt') \\
& \lhd \text{trace}(tt') \in \text{tock}^* \rhd \\
& a = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \land \\
& a \notin \text{refusals}(\text{idleprefix}(tt')) \land \\
& P[\text{tail}(\text{idlesuffix}(tt'))/tt'] \land \\
& tt' \in \text{TimedTrace} \\
\right)
\end{align*} \\
= \{ \text{prefix} \} \\
a \rightarrow P \\
= \{ \text{assumption: } a \rightarrow P \} \\
\text{true}

3. **Nondeterminism** Assume

\[ t \leq tt' \land P \Rightarrow P[\text{tr} \cap t/\text{tt}'] \]
\[ t \leq tt' \land Q \Rightarrow Q[\text{tr} \cap t/\text{tt}'] \]

W.T.P.

\[ t \leq tt' \land P \cap Q \Rightarrow (P \cap Q)[\text{tr} \cap t/\text{tt}'] \]

Follows from properties of substitution.
4. External choice

\[ P \boxdot Q = (P \land Q)[\text{idleprefix}(t')/tt'] \land (P \lor Q) \]

Assume

\[
\begin{align*}
t \preceq t' & \land P \Rightarrow P[t/\text{tt}'] \\
t \preceq t' & \land Q \Rightarrow Q[t/\text{tt}']
\end{align*}
\]

W.T.P.

\[
t \preceq t' \land (P \boxdot Q) \Rightarrow (P \boxdot Q)[t/\text{tt}']
\]

\[
\begin{align*}
(P \boxdot Q)[t/\text{tt}'] & = \{ \text{external choice} \} \\
((P \land Q)[\text{idleprefix}(t')/tt'] \land (P \lor Q))[t/\text{tt}'] & = \{ \text{substitution} \} \\
(P[\text{idleprefix}(t')/tt'] \land Q[\text{idleprefix}(t')/tt'] \land (P \lor Q))[t/\text{tt}'] & = \{ \text{substitution} \} \\
(P[\text{idleprefix}(t)/\text{tt}'] \land Q[\text{idleprefix}(t)/\text{tt}'] \land (P \lor Q t'/tt')) & \iff \{ \text{assumptions: (i) } t \preceq t'; (ii) } t \preceq t' \land P \Rightarrow P[t'/tt']; \} \\
& \{ t \preceq t' \land Q \Rightarrow Q[t'/tt'] \} \\
(P[\text{idleprefix}(t)/\text{tt}'] \land Q[\text{idleprefix}(t)/\text{tt}'] \land (P \lor Q)) & \iff \{ \text{idleprefix}(t) \preceq \text{idleprefix}(t') \land P[\text{idleprefix}(t')/\text{idleprefix}(t')] \Rightarrow \} \\
& \{ P[\text{idleprefix}(t)/\text{idleprefix}(t')] \} \\
& \{ \text{idleprefix}(t) \preceq \text{idleprefix}(tt') \land Q[\text{idleprefix}(tt')/\text{idleprefix}(tt')] \Rightarrow \} \\
& \{ Q[\text{idleprefix}(t)/\text{idleprefix}(tt')] \} \\
(P[\text{idleprefix}(tt')/tt'] \land Q[\text{idleprefix}(tt')/tt'] \land (P \lor Q)) & = \{ \text{propositional calculus} \} \\
((P \land Q)[\text{idleprefix}(tt')/tt'] \land (P \lor Q)) & = \{ \text{definition external choice} \} \\
P \boxdot Q
\]
5. **parallel** Assume $P \parallel_A Q$ and $t \leq tt'$. W.T.P. $(P \parallel_A Q)[t/tt']$

\[
(P \parallel_A Q)[t/tt'] = \{ \text{definition of } \parallel_A \}
\]

\[
(\exists s, u \cdot P[s/tt'] \land Q[u/tt'] \land tt' \in s \parallel_A u)[t/tt'] = \{ \text{substitution} \}
\]

\[
\exists s, u \cdot P[s_1/tt'] \land Q[u_1/tt'] \land t \in s \parallel_A u
\]

\[
\leftarrow \{ P \text{ and } Q \text{ are } T2\text{-healthy} \}
\]

\[
\exists s, u \cdot P[s_1/tt'] \land s \leq s_1 \land Q[u_1/tt'] \land u \leq u_1 \land t \in s \parallel_A u
\]

\[
\leftarrow \{ \text{lemma: parallel-precedence} \}
\]

\[
P[s_1/tt'] \land Q[u_1/tt'] \land tt' \in s_1 \parallel_A u_1 \land t \leq tt'
\]

\[
= \{ \text{assumption: } t \leq tt' \}
\]

\[
P[s_1/tt'] \land Q[u_1/tt'] \land tt' \in s_1 \parallel_A u_1
\]

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

\[
P \parallel_A Q
\]

For proof of the parallel precedence lemma, see Appendix B.

6. **Hiding** Assume

\[
P \land t \leq tt' \Rightarrow P[t/tt']
\]

Would like to prove

\[
P \setminus A \land t \leq tt' \Rightarrow (P \setminus A)[t/tt']
\]

\[
P \setminus A
\]

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

\[
\exists u \cdot P[u/tt'] \land A \text{ urgent } u \land tt' = u \setminus A
\]

\[
= \{ t \leq tt' \land tt' = u \setminus A \Rightarrow \exists v \cdot v \leq u \land t = v \setminus A \}
\]

\[
\exists u, v \cdot P[u/tt'] \land A \text{ urgent } u \land tt' = u \setminus A \land v \leq u \land t = v \setminus A
\]

\[
= \{ P \text{ is } T2 \}
\]

\[
\exists u, v \cdot P[v/tt'] \land A \text{ urgent } u \land tt' = u \setminus A \land v \leq u \land t = v \setminus A
\]

\[
= \{ A \text{ urgent } u \land v \leq u \Rightarrow A \text{ urgent } v \}
\]

\[
\exists u, v \cdot P[v/tt'] \land A \text{ urgent } v \land tt' = u \setminus A \land v \leq u \land t = v \setminus A
\]

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

\[
\exists v \cdot P[v/tt'] \land A \text{ urgent } v \land t = v \setminus A
\]

\[
= \{ \text{substitution} \}
\]

\[
(\exists v \cdot P[v/tt'] \land A \text{ urgent } v \land tt' = v \setminus A)[t/tt']
\]

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

\[
(P \setminus A)[t/tt']
\]

7. **Timing**

\[
[t \leq tt' \land (P \Rightarrow P[t/tt']) \land (Q \Rightarrow Q[t/tt']) \land (P \triangleright^n Q) \Rightarrow (P \triangleright^n Q)[t/tt']]
\]

**Case 1:** $\text{tok}^n \leq \text{trace}(tt')$

\[
P \triangleright^n Q
\]

94
\[
\{ \text{timeout} \}
\]
\[
(\exists u \bullet u \leq tt' \land \text{trace}(u) = \text{tock}^n \land P[tt'] \land Q[tt' - u/\text{tt'}])
\]
\[
\Rightarrow \text{trace}(tt') \supset P
\]
\[
\{ \text{assumption: tock}^n \leq \text{trace}(tt') \}
\]
\[
(\exists u \bullet u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[tt'] \land Q[tt' - u/\text{tt'}])
\]
\[
\Rightarrow \{ \text{predicate calculus: for arbitrary } u \}
\]
\[
u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[tt'] \land Q[tt' - u/\text{tt'}]
\]
\[
\Rightarrow \{ \text{precedence: } u \leq tt' \land t \leq tt' \Rightarrow t < u \lor u \leq t \}
\]
\[
u \leq tt' \land (t < u \lor u \leq t) \land (\text{trace}(u) = \text{tock}^n) \land P[tt'] \land Q[tt' - u/\text{tt'}]
\]
\[
\{ \text{propositional calculus} \}
\]
\[
(u \leq tt' \land t < u \land (\text{trace}(u) = \text{tock}^n) \land P[tt'] \land Q[tt' - u/\text{tt'}])
\]
\[
\lor (u \leq tt' \land u < t \land (\text{trace}(u) = \text{tock}^n) \land P[tt'] \land Q[tt' - u/\text{tt'}])
\]
\[
\Rightarrow \{ \text{precedence: } t < u \land (\text{trace}(u) = \text{tock}^n) \Rightarrow \neg \text{tock}^n \leq \text{trace}(t) \}
\]
\[
(\neg \text{tock}^n \leq \text{trace}(t) \land u \leq tt' \land t < u \land (\text{trace}(u) = \text{tock}^n)
\]
\[
\land P[tt'] \land Q[tt' - u/\text{tt'}])
\]
\[
\lor (u \leq tt' \land u \leq t \land (\text{trace}(u) = \text{tock}^n) \land P[tt'] \land Q[tt' - u/\text{tt'}])
\]
\[
\Rightarrow \{ \text{assumption: } t \leq u \land P[tt'] \Rightarrow P[t/\text{tt'}] \}
\]
\[
(\neg \text{tock}^n \leq \text{trace}(t) \land u \leq tt' \land t < u \land (\text{trace}(u) = \text{tock}^n)
\]
\[
\land P[t/\text{tt'}] \land Q[tt' - u/\text{tt'}])
\]
\[
\lor (u \leq tt' \land u \leq t \land (\text{trace}(u) = \text{tock}^n) \land P[tt'] \land Q[tt' - u/\text{tt'}])
\]
\[
\Rightarrow \{ \text{propositional calculus} \}
\]
\[
(\neg \text{tock}^n \leq \text{trace}(t) \land P[t/\text{tt'}])
\]
\[
\lor (u \leq tt' \land u \leq t \land (\text{trace}(u) = \text{tock}^n) \land P[tt'] \land Q[tt' - u/\text{tt'}])
\]
\[
\Rightarrow \{ \text{precedence: } u \leq tt' \Rightarrow (t - u) \leq (tt' - u) \}
\]
\[
(\neg \text{tock}^n \leq \text{trace}(t) \land P[t/\text{tt'}])
\]
\[
\lor (u \leq tt' \land u \leq t \land (t - u) \leq (tt' - u) \land (\text{trace}(u) = \text{tock}^n)
\]
\[
\land P[t/\text{tt'}] \land Q[tt' - u/\text{tt'}])
\]
\[
\Rightarrow \{ \text{assumption: } (t - u) \leq (tt' - u) \land Q[tt' - u/\text{tt'}] \Rightarrow Q[t - u/\text{tt'}] \}
\]
\[
(\neg \text{tock}^n \leq \text{trace}(t) \land P[t/\text{tt'}])
\]
\[
\lor (u \leq tt' \land u \leq t \land (\text{trace}(u) = \text{tock}^n) \land P[t/\text{tt'}] \land Q[t - u/\text{tt'}])
\]
\[
\Rightarrow \{ \text{propositional calculus} \}
\]
\[
(\neg \text{tock}^n \leq \text{trace}(t) \land P[t/\text{tt'}])
\]
\[
\lor (u \leq t \land (\text{trace}(u) = \text{tock}^n) \land P[t/\text{tt'}] \land Q[t - u/\text{tt'}])
\]
\[
\Rightarrow \{ \text{precedence: } u \leq tt' \Rightarrow \text{trace}(u) \leq \text{trace}(t) \}
\]
\[
(\neg \text{tock}^n \leq \text{trace}(t) \land P[t/\text{tt'}])
\]
\[
\lor (\text{trace}(u) \leq \text{trace}(t) \land u \leq t \land (\text{trace}(u) = \text{tock}^n)
\]
\[
\land P[t/\text{tt'}] \land Q[t - u/\text{tt'}])
\]
\[
= \{ \text{Leibniz} \}
Case 2: \( \nexists \) tock\( ^n \leq \) trace\( (tt') \). Note that sequence prefix is transitive; in particular,

\[
\begin{align*}
&\nexists \ u \bullet u \leq t \land (trace(u) = \text{tock\( ^n \))} \\
&\quad \land P[u/\text{tock\( ^n \))] \land Q[t - u/\text{tock\( ^n \))]
\end{align*}
\]

\[
\Rightarrow \{ \ \text{assumption: } \nexists \text{tock\( ^n \)) \leq \text{trace\( (tt') \))}\}
\]

\[
P \Rightarrow P[t/\text{tock\( ^n \))]
\]

\[
\Rightarrow \{ \ \text{assumption: } P \Rightarrow P[t/\text{tock\( ^n \))]}\}
\]

\[
P[t/\text{tock\( ^n \))]
\]

\[
\Rightarrow \{ \ \text{assumption: } \nexists \text{tock\( ^n \)) \leq \text{trace\( (tt') \))}\}
\]

\[
\nexists \ u \bullet u \leq tt' \land (trace(u) = \text{tock\( ^n \))} \\
\quad \land P[tt'/u/\text{tock\( ^n \))] \land Q[tt' - u/\text{tock\( ^n \))]
\]

\[
\Rightarrow \{ \ \text{assumption: } P \Rightarrow P[t/\text{tock\( ^n \))]}\}
\]

\[
P \Rightarrow \{ \ \text{transitivity}\}
\]

\[
\nexists \ u \bullet u \leq t \land (trace(u) = \text{tock\( ^n \))} \\
\quad \land P[u/\text{tock\( ^n \))] \land Q[t - u/\text{tock\( ^n \))]
\]

\[
\Rightarrow \{ \ \text{conditional}\}
\]

\[
(P \Rightarrow P)[t/\text{tock\( ^n \))]
\]
8. **Recursion** $T_2$ can be written as the conjunctive idempotent

$$T_2(P) = P \land (P[t/\ell']) \lor \neg t \leq \ell'$$

Recursion therefore satisfies $T_2$, by [12].

### A.1.2 Time can always pass

**Theorem 4.2.3 (Refusals)** Every CML operator preserves $T_3$-healthiness.

**Definition A.1.1** Lowe & Ouaknine Axiom

$$T_3 \quad [P[s \triangleright (A, \text{tock}) \triangleright v/\ell'] \land \neg P[s \triangleright \langle a \rangle/\ell'] \Rightarrow P[s \triangleright (A \cup \{a\}, \text{tock}) \triangleright v/\ell']]$$

**Theorem A.1.1**

1. **Deadlock**
   
   Assume $STOP$ and that $[STOP[s \triangleright (A, \text{tock}) \triangleright v/\ell'] \land \neg STOP[s \triangleright \langle a \rangle/\ell']$

   We show that $STOP[s \triangleright (A \cup \{a\}, \text{tock}) \triangleright v/\ell']$

   $$STOP[s \triangleright (A, \text{tock}) \triangleright v/\ell']$$
   = \{ definition \}

   $$T_0[\text{trace}(\ell') \in \text{tock}^*][s \triangleright (A, \text{tock}) \triangleright v/\ell']$$
   = \{ substitution \}

   $$\text{trace}(s \triangleright (A, \text{tock}) \triangleright v) \in \text{tock}^* \land s \triangleright (A, \text{tock}) \triangleright v \in \text{TimedTrace}$$
   = \{ (s \triangleright (A, \text{tock}) \triangleright v \in \text{TimedTrace}) = (s \triangleright (A \cup \{a\}, \text{tock}) \triangleright v \in \text{TimedTrace}) \}

   $$\text{trace}(s \triangleright (A \cup \{a\}, \text{tock}) \triangleright v) \in \text{tock}^* \land s \triangleright (A \cup \{a\}, \text{tock}) \triangleright v \in \text{TimedTrace}$$
   = \{ substitution \}

   $$T_0[\text{trace}(\ell') \in \text{tock}^*][s \triangleright (A \cup \{a\}, \text{tock}) \triangleright v/\ell']$$
   = \{ definition \}

   $$STOP[s \triangleright (A \cup \{a\}, \text{tock}) \triangleright v/\ell']$$

2. **prefix**

   Assume $T_3(P)$ and that $[(b \rightarrow P)[s \triangleright (A, \text{tock}) \triangleright v/\ell'] \land \neg (b \rightarrow P)[s \triangleright \langle a \rangle/\ell']]$

   We show $(b \rightarrow P)[s \triangleright (A \cup \{a\}, \text{tock}) \triangleright v/\ell']]$

   Observe first that if $\neg (a \rightarrow P)[s \triangleright \langle a \rangle/\ell']$ then a must have occurred, and so (i) trace(s) $\neq \emptyset$ (ii) hd(trace(s)) = a; (iii) a $\notin$ refusals(idleprefix(s))
Also, note (observation (iv))

\[\neg (a \to P)[s \sim \langle a \rangle/tt']\]
\[= \{\text{definition prefix}\}\]
\[\neg T0\]
\[\left(\begin{array}{l}
\neg \text{refusals}(tt') \\
\text{trace}(tt') \in \text{lock}^* \not\supset\\
\neg \text{refusals}(\text{idleprefix}(tt')) \land \\
\neg P[\text{tail}(\text{idleprefix}(tt'))/tt']
\end{array}\right)
\]
\[\neg (s \sim \langle a \rangle) \in \text{lock}^*\]
\[a \notin \text{refusals}(s \sim \langle a \rangle) \land \\
\neg P[\text{tail}(s \sim \langle a \rangle)/tt']\]
\[= \{\text{observation (i)}:\neg a \notin \text{refusals}(s \sim \langle a \rangle)\}\]
\[\neg (\neg s \sim \langle a \rangle) \in \text{TimedTrace}\]
\[\neg \text{refusals}(\text{idleprefix}(s \sim \langle a \rangle)) \land \\
\neg P[\text{tail}(\text{idleprefix}(s \sim \langle a \rangle))/tt']\]
\[= \{\text{observation:}\neg a \notin \text{refusals}(\text{idleprefix}(s)) \land a \lor \neg \text{false} = a\}\]
\[\neg s \sim \langle a \rangle \in \text{TimedTrace}\]
\[\neg \text{refusals}(\text{idleprefix}(s)) \land \\
\neg P[\text{tail}(\text{idleprefix}(s))/tt']\]
\[= \{\text{observation (ii):}\neg (a \to P)[s \sim \langle a \rangle/\text{tt}']\}\]
Case i: $b = a$

$$(b \rightarrow P)[s \circlearrowright (A, \text{tock}) \circlearrowright v/tt']$$
$$= \{ \text{case } b = a \}$$
$$(a \rightarrow P)[s \circlearrowright (A, \text{tock}) \circlearrowright v/tt']$$

{definition of prefix}

$$\begin{align*}
T_0 \quad &\left\{ \begin{array}{l}
\text{a} \notin \text{refusals}(tt') \\
\text{a} = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \land \\
\text{a} \notin \text{refusals}(\text{idleprefix}(tt')) \land \\
\text{P}[\text{tail}(\text{idlesuffix}(tt'))/tt']
\end{array} \right.
\ \{ s \circlearrowright (A, \text{tock}) \circlearrowright v/tt' \}
\end{align*}$$

= \{ \text{substitution} \}

$$\begin{align*}
\text{a} \notin \text{refusals}(s \circlearrowright (A, \text{tock}) \circlearrowright v) \land \\
\text{s} \circlearrowright (A, \text{tock}) \circlearrowright v \in \text{TimedTrace}
\end{align*}$$

$$\begin{align*}
< \text{trace}(s \circlearrowright (A, \text{tock}) \circlearrowright v) \in \text{tock}^* >
\quad &\left\{ \begin{array}{l}
\text{a} = \text{head}(\text{trace}(\text{idlesuffix}(s \circlearrowright (A, \text{tock}) \circlearrowright v))) \land \\
\notin \text{refusals}(\text{idleprefix}(s \circlearrowright (A, \text{tock}) \circlearrowright v)) \land \\
\text{P}[\text{tail}(\text{idlesuffix}(s \circlearrowright (A, \text{tock}) \circlearrowright v))/tt'] \land \\
\text{s} \circlearrowright (A, \text{tock}) \circlearrowright v \in \text{TimedTrace}
\end{array} \right.
\ \{ s \circlearrowright (A, \text{tock}) \circlearrowright v \in \text{TimedTrace} \}
\end{align*}$$

= \{ \text{observation: } \text{trace}(s) \neq \langle \rangle, \text{ and } a \text{ if } \text{false } b = b \}$$

$$\begin{align*}
\text{a} = \text{head}(\text{trace}(\text{idlesuffix}(s \circlearrowright (A, \text{tock}) \circlearrowright v))) \land \\
\text{s} \circlearrowright (A, \text{tock}) \circlearrowright v \in \text{TimedTrace}
\end{align*}$$

$$\begin{align*}
\text{a} \notin \text{refusals}(\text{idleprefix}(s \circlearrowright (A, \text{tock}) \circlearrowright v)) \land \\
\text{s} \circlearrowright (A, \text{tock}) \circlearrowright v \in \text{TimedTrace}
\end{align*}$$

$$\begin{align*}
\Rightarrow \{ s \circlearrowright (A, \text{tock}) \circlearrowright v \in \text{TimedTrace} \Rightarrow s \circlearrowright (a) \in \text{TimedTrace} \}$$

$$\begin{align*}
\text{a} = \text{head}(\text{trace}(\text{idlesuffix}(s \circlearrowright (A \cup \{a\}, \text{tock}) \circlearrowright v))) \land \\
\text{a} \notin \text{refusals}(\text{idleprefix}(s \circlearrowright (A \cup \{a\}, \text{tock}) \circlearrowright v)) \land \\
\text{P}[\text{tail}(\text{idlesuffix}(s \circlearrowright (A \cup \{a\}, \text{tock}) \circlearrowright v))/tt'] \land \\
\text{s} \circlearrowright (A, \text{tock}) \circlearrowright v \in \text{TimedTrace} \land \\
\text{s} \circlearrowright (a) \in \text{TimedTrace}
\end{align*}$$
= \{ \text{by observation (iv) and second clause of hypothesis} \}
= \{ \lnot P[\text{tail(idlesuffix}(s \triangleleft \{a\}))/tt']) \}
= \{ a = \text{head}(\text{trace(idlesuffix}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v)))) \land \\
a \notin \text{refusals(idleprefix}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v))) \land \\
P[\text{tail(idlesuffix}(s \triangleleft (A, \text{tock} \triangleleft v))/tt')] \land \\
s \triangleleft (A, \text{tock} \triangleleft v) \in \text{TimedTrace} \land \\
s \triangleleft \{a\} \in \text{TimedTrace} \land \\
\lnot P[\text{tail(idlesuffix}(s \triangleleft \{a\}))/tt'] \}
\Rightarrow \{ T3(P) \}
= \{ s \triangleleft (A, \text{tock} \triangleleft v) \in \text{TimedTrace} = s \triangleleft \{a\} \in \text{TimedTrace} \}
= \{ a = \text{head}(\text{trace(idlesuffix}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v)))) \land \\
a \notin \text{refusals(idleprefix}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v))) \land \\
s \triangleleft (A, \text{tock} \triangleleft v) \in \text{TimedTrace} \land \\
P[\text{tail(idlesuffix}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v))/tt')] \land \\
s \triangleleft \{a\} \in \text{TimedTrace} \land \\
P[\text{tail(idlesuffix}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v))/tt'] \}
= \{ a \triangleleft \text{false} \triangleleft b = b \}
= \{ a \notin \text{refusals}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v) \land s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v) \in \text{TimedTrace} \land \\
\triangleleft \text{trace}(s \triangleleft (A, \text{tock} \triangleleft v) \in \text{tock}^* \triangleleft \\
\triangleleft a = \text{head}(\text{trace(idlesuffix}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v)))) \land \\
a \notin \text{refusals(idleprefix}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v))) \land \\
P[\text{tail(idlesuffix}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v))/tt')] \land \\
s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v) \in \text{TimedTrace} \}
= \{ \text{trace}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v) \in \text{tock}^* = \text{trace}(s \triangleleft (A, \text{tock} \triangleleft v) \in \text{tock}^*) \}
= \{ a \notin \text{refusals}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v) \land s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v) \in \text{TimedTrace} \land \\
\triangleleft \text{trace}(s \triangleleft \{A \cup \{a\}, \text{tock} \triangleleft v) \in \text{tock}^* \triangleleft \\
\triangleleft a = \text{head}(\text{trace(idlesuffix}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v)))) \land \\
a \notin \text{refusals(idleprefix}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v))) \land \\
P[\text{tail(idlesuffix}(s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v))/tt')] \land \\
s \triangleleft \{A \cup \{a\}, \text{tock} \triangleleft v) \in \text{TimedTrace} \}
= \{ \text{substitution} \}
= \{ a \notin \text{refusals}(tt') \land a = \text{head}(\text{trace(idlesuffix}(tt')))) \land \\
P[\text{tail(idlesuffix}(tt'))/tt'] \}
= \{ \text{definition of prefix} \}
= \{ a \rightarrow P[s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v)/tt'] \}
= \{ b = a \}
= \{ b \rightarrow P[s \triangleleft (A \cup \{a\}, \text{tock} \triangleleft v)/tt'] \}
Case ii: \( b \neq a \)

\[
(b \to P)(s \sim \langle A, tock \rangle \sim v/\tt')
\]
{ definition of prefix }

\[
\begin{align*}
T0 \quad & b \notin \text{refusals}(\tt') \quad \langle \text{trace}(\tt') \in \text{tock}^* \quad b = \text{head}(\text{trace}(\text{idlesuffix}(\tt')))) \land \quad b \notin \text{refusals}(\text{idleprefix}(\tt')) \land \quad P[\text{tail}(\text{idlesuffix}(\tt'))/\tt']
\end{align*}
\]

\[
[ s \sim \langle A, tock \rangle \sim v/\tt']
\]

\[
= \{ \text{substitution} \}
\]

\[
\begin{align*}
& b \notin \text{refusals}(s \sim \langle A, tock \rangle \sim v) \land s \sim \langle A, tock \rangle \sim v \in \text{TimedTrace} \quad \langle \text{trace}(s \sim \langle A, tock \rangle \sim v) \in \text{tock}^* \quad b = \text{head}(\text{trace}(\text{idlesuffix}(s \sim \langle A, tock \rangle \sim v))) \land \quad b \notin \text{refusals}(\text{idleprefix}(s \sim \langle A, tock \rangle \sim v)) \land \quad P[\text{tail}(\text{idlesuffix}(s \sim \langle A, tock \rangle \sim v))/\tt'] \land \quad s \sim \langle A, tock \rangle \sim v \in \text{TimedTrace}
\end{align*}
\]

\[
= \{ s \sim \langle A, tock \rangle \sim v \in \text{TimedTrace} \} = (s \sim \langle A \cup \{a\}, tock \rangle \sim v \in \text{TimedTrace})
\]

\[
\begin{align*}
& b \notin \text{refusals}(s \sim \langle A, tock \rangle \sim v) \land s \sim \langle A \cup \{a\}, tock \rangle \sim v \in \text{TimedTrace} \quad \langle \text{trace}(s \sim \langle A, tock \rangle \sim v) \in \text{tock}^* \quad b = \text{head}(\text{trace}(\text{idlesuffix}(s \sim \langle A, tock \rangle \sim v))) \land \quad b \notin \text{refusals}(\text{idleprefix}(s \sim \langle A, tock \rangle \sim v)) \land \quad P[\text{tail}(\text{idlesuffix}(s \sim \langle A, tock \rangle \sim v))/\tt'] \land \quad s \sim \langle A \cup \{a\}, tock \rangle \sim v \in \text{TimedTrace}
\end{align*}
\]

\[
= \{ b \neq a \land (b \notin \text{refusals}(s \sim \langle A, tock \rangle \sim v)) = (b \notin \text{refusals}(s \sim \langle A \cup \{a\}, tock \rangle \sim v)) \} = \}
\]

\[
\begin{align*}
& (b \notin \text{refusals}(\text{idleprefix}(s \sim \langle A, tock \rangle \sim v)))
\end{align*}
\]

\[
\begin{align*}
& b \notin \text{refusals}(s \sim \langle A \cup \{a\}, tock \rangle \sim v) \land s \sim \langle A \cup \{a\}, tock \rangle \sim v \in \text{TimedTrace} \quad \langle \text{trace}(s \sim \langle A, tock \rangle \sim v) \in \text{tock}^* \quad b = \text{head}(\text{trace}(\text{idlesuffix}(s \sim \langle A, tock \rangle \sim v))) \land \quad b \notin \text{refusals}(\text{idleprefix}(s \sim \langle A \cup \{a\}, tock \rangle \sim v)) \land \quad P[\text{tail}(\text{idlesuffix}(s \sim \langle A, tock \rangle \sim v))/\tt'] \land \quad s \sim \langle A \cup \{a\}, tock \rangle \sim v \in \text{TimedTrace}
\end{align*}
\]

\[
= \{ \text{trace}(s \sim \langle A, tock \rangle \sim v) \in \text{tock}^* \} = (\text{trace}(s \sim \langle A \cup \{a\}, tock \rangle \sim v) \in \text{tock}^*)
\]

\[
\begin{align*}
& b \notin \text{refusals}(s \sim \langle A \cup \{a\}, tock \rangle \sim v) \land s \sim \langle A \cup \{a\}, tock \rangle \sim v \in \text{TimedTrace} \quad \langle \text{trace}(s \sim \langle A, tock \rangle \sim v) \in \text{tock}^* \quad b = \text{head}(\text{trace}(\text{idlesuffix}(s \sim \langle A, tock \rangle \sim v))) \land \quad b \notin \text{refusals}(\text{idleprefix}(s \sim \langle A \cup \{a\}, tock \rangle \sim v)) \land \quad P[\text{tail}(\text{idlesuffix}(s \sim \langle A, tock \rangle \sim v))/\tt'] \land \quad s \sim \langle A \cup \{a\}, tock \rangle \sim v \in \text{TimedTrace}
\end{align*}
\]
= \{ (b = \text{head}(\text{trace}(\text{idlesuffix}(s \sim (A, tock) \sim v)))) = \}
\{ (b = \text{head}(\text{trace}(\text{idlesuffix}(s \sim (A \cup \{a\}, tock) \sim v)))) = \}
\begin{aligned}
&b \notin \text{refusals}(s \sim (A \cup \{a\}, tock) \sim v) \land s \sim (A \cup \{a\}, tock) \sim v \in \text{TimedTrace} \\
&\langle b = \text{head}(\text{trace}(\text{idlesuffix}(s \sim (A \cup \{a\}, tock) \sim v)))) = \rangle \\
&b \notin \text{refusals}(\text{idleprefix}(s \sim (A \cup \{a\}, tock) \sim v)) \land \\
P[\text{tail}(\text{idlesuffix}(s \sim (A, tock) \sim v))/tt'] \land \\
s \sim (A \cup \{a\}, tock) \sim v \in \text{TimedTrace} \\
\neg P[\text{tail}(\text{idlesuffix}(s \sim \langle a \rangle))/tt']
\end{aligned}

= \{ (\neg (b \rightarrow P)[s \sim \langle a \rangle/\text{tt}'] \Rightarrow \neg P[\text{tail}(\text{idlesuffix}(s \sim \langle a \rangle))/\text{tt}']) \}
\begin{aligned}
&b \notin \text{refusals}(s \sim (A \cup \{a\}, tock) \sim v) \land s \sim (A \cup \{a\}, tock) \sim v \in \text{TimedTrace} \\
&\langle b = \text{head}(\text{trace}(\text{idlesuffix}(s \sim (A \cup \{a\}, tock) \sim v)))) = \rangle \\
&b \notin \text{refusals}(\text{idleprefix}(s \sim (A \cup \{a\}, tock) \sim v)) \land \\
P[\text{tail}(\text{idlesuffix}(s \sim (A, tock) \sim v))/\text{tt}'] \land \\
s \sim (A \cup \{a\}, tock) \sim v \in \text{TimedTrace} \\
\neg P[\text{tail}(\text{idlesuffix}(s \sim \langle a \rangle))/\text{tt}']
\end{aligned}

\Rightarrow \{ T3(P) \}
\begin{aligned}
&b \notin \text{refusals}(s \sim (A \cup \{a\}, tock) \sim v) \land s \sim (A \cup \{a\}, tock) \sim v \in \text{TimedTrace} \\
&\langle b = \text{head}(\text{trace}(\text{idlesuffix}(s \sim (A \cup \{a\}, tock) \sim v)))) = \rangle \\
&b \notin \text{refusals}(\text{idleprefix}(s \sim (A \cup \{a\}, tock) \sim v)) \land \\
P[\text{tail}(\text{idlesuffix}(s \sim (A \cup \{a\}, tock) \sim v))/\text{tt}'] \land \\
s \sim (A \cup \{a\}, tock) \sim v \in \text{TimedTrace} \\
\neg P[\text{tail}(\text{idlesuffix}(s \sim \langle a \rangle))/\text{tt}']
\end{aligned}

= \{ \text{substitution} \}
\begin{aligned}
&b \notin \text{refusals}(\text{tt}') \\
&\text{TimedTrace} \langle \text{tt}' \in \text{tock}^* \rangle \\
&b = \text{head}(\text{trace}(\text{idlesuffix}(\text{tt}')) = \rangle \\
&b \notin \text{refusals}(\text{idleprefix}(\text{tt}')) \land \\
P[\text{tail}(\text{idlesuffix}(\text{tt}'))/\text{tt}'] \\
s \sim (A \cup \{a\}, tock) \sim v/\text{tt}''
\end{aligned}

\Rightarrow \{ \text{definition of prefix} \}
\begin{aligned}
&(b \rightarrow P)[s \sim (A \cup \{a\}, tock) \sim v/\text{tt}'']
\end{aligned}

3. Internal choice

Follows from properties of disjunction.
4. External choice

Given $T_3(P)$ and $T_3(Q)$, we want to prove $T_3(P \sqcup Q)$, i.e. that

$$(P \sqcup Q)[s \circ \langle A, \text{tock} \rangle \leadsto v/\tau'] \land \neg (P \sqcup Q)[s \circ \langle a \rangle/\tau']$$

implies

$$(P \sqcup Q)[s \circ \langle A \cup \{a\}, \text{tock} \rangle \leadsto v/\tau']$$

\[
(P \sqcup Q)[s \circ \langle A, \text{tock} \rangle \leadsto v/\tau'] \land \neg (P \sqcup Q)[s \circ \langle a \rangle/\tau']
= \{ \text{definition external choice, substitution} \}
(P \land Q)[\text{idleprefix}(s \circ \langle A, \text{tock} \rangle, v)/\tau'] \land (P \lor Q)[s \circ \langle A, \text{tock} \rangle \leadsto v/\tau'] \land
\neg ((P \land Q)[\text{idleprefix}(s \circ \langle a \rangle)/\tau'] \land (P \lor Q)[s \circ \langle a \rangle/\tau'])
= \{ \text{propositional calculus} \}
P[\text{idleprefix}(s \circ \langle A, \text{tock} \rangle, v)/\tau'] \land Q[\text{idleprefix}(s \circ \langle A, \text{tock} \rangle, v)/\tau'] \land
(P[s \circ \langle A, \text{tock} \rangle \leadsto v/\tau'] \lor Q[s \circ \langle A, \text{tock} \rangle \leadsto v/\tau']) \land
\neg (P[\text{idleprefix}(s \circ \langle a \rangle)/\tau'] \land
Q[\text{idleprefix}(s \circ \langle a \rangle)/\tau']) \land
(P[s \circ \langle a \rangle/\tau'] \lor Q[s \circ \langle a \rangle/\tau'])
\Rightarrow \{ \text{calculus} \}
(P[\text{idleprefix}(s \circ \langle A, \text{tock} \rangle, v)/\tau'] \land \neg P[\text{idleprefix}(s \circ \langle a \rangle)/\tau']) \land
(Q[\text{idleprefix}(s \circ \langle A, \text{tock} \rangle, v)/\tau'] \land \neg Q[\text{idleprefix}(s \circ \langle a \rangle)/\tau']) \land
(P[s \circ \langle A, \text{tock} \rangle \leadsto v/\tau'] \land \neg P[s \circ \langle a \rangle/\tau'] \lor
Q[s \circ \langle A, \text{tock} \rangle \leadsto v/\tau'] \land \neg Q[s \circ \langle a \rangle/\tau'])
\Rightarrow \{ T_3(P), T_3(Q) \}
P[\text{idleprefix}(s \circ \langle A \cup \{a\}, \text{tock} \rangle, v)/\tau'] \land
Q[\text{idleprefix}(s \circ \langle A \cup \{a\}, \text{tock} \rangle, v)/\tau'] \land
(P[s \circ \langle A \cup \{a\}, \text{tock} \rangle \leadsto v/\tau'] \lor
Q[s \circ \langle A \cup \{a\}, \text{tock} \rangle \leadsto v/\tau'])
= \{ \text{distribution} \}
(P \land Q)[\text{idleprefix}(s \circ \langle A \cup \{a\}, \text{tock} \rangle, v)/\tau'] \land (P \lor Q)[s \circ \langle A \cup \{a\}, \text{tock} \rangle \leadsto v/\tau']
= \{ \text{definition} \}
(P \sqcup Q)[s \circ \langle A \cup \{a\}, \text{tock} \rangle \leadsto v/\tau']

5. Parallel Composition

Given $T_3(P)$ and $T_3(Q)$, we want to prove that $T_3(P \parallel_A Q)$, i.e. that

$$(P \parallel_A Q)[s \circ \langle B, \text{tock} \rangle \leadsto v/\tau'] \land \neg (P \parallel_A Q)[s \circ \langle a \rangle/\tau']$$

implies

$$(P \parallel_A Q)[s \circ \langle B \cup \{a\}, \text{tock} \rangle \leadsto v/\tau']$$

We begin with the case where $a \in A$.  

103
First, observe

\[\neg (P[|A| \rightarrow Q][s \leftarrow \langle a \rangle/\langle t \rangle]')\]

= \{ substitution; definition of parallel; \(a \in A\) \}

= \neg (\exists t_s, u_s \in P[t_s \leftarrow \langle a \rangle/\langle t \rangle] Q[u_s \leftarrow \langle a \rangle/\langle t \rangle] \land s \leftarrow \langle a \rangle \in t_s \leftarrow \langle a \rangle |A| u_s \leftarrow \langle a \rangle)\}

= \{ predicate calculus \}

= (\forall t_s, u_s \in \neg P[t_s \leftarrow \langle a \rangle/\langle t \rangle] \lor \neg Q[u_s \leftarrow \langle a \rangle/\langle t \rangle] \lor \neg s \leftarrow \langle a \rangle \in t_s \leftarrow \langle a \rangle |A| u_s \leftarrow \langle a \rangle)\)

= \{ s = t \parallel u; \(a \in A\) \}

= (\forall t_s, u_s \in \neg P[t_s \leftarrow \langle a \rangle/\langle t \rangle] \lor \neg Q[u_s \leftarrow \langle a \rangle/\langle t \rangle])\)

true

= \{ first antecedent \}

\(P[|A| \rightarrow Q][s \leftarrow \langle B, tock \rangle \leftarrow v/\langle t \rangle]')\]

= \{ definition of parallel composition \}

= (\exists t, u \in P[t/\langle t \rangle] \land Q[u/\langle t \rangle] \land \langle t \rangle \in t |A| u)[s \leftarrow \langle B, tock \rangle \leftarrow v/\langle t \rangle]')\]

= \{ substitution \}

= (\exists t, u \in P[t/\langle t \rangle] \land Q[u/\langle t \rangle] \land s \leftarrow \langle B, tock \rangle \leftarrow v \in t |A| u)\}

= \{ t is the prefix of t that contributes to s, t_s \}

= \{ s is the prefix of u that contributes to v \}

= (\exists t_s, t_v, u \in P[t_s \leftarrow \langle B, tock \rangle \leftarrow t_v/\langle t \rangle] \land Q[u/\langle t \rangle] \land (s \leftarrow \langle B, tock \rangle \leftarrow v) \in (t_s \leftarrow \langle B, tock \rangle \leftarrow t_v) |A| u)\}

= \{ by observation, (assume wlog that \(\neg P[t_s \leftarrow \langle a \rangle/\langle t \rangle]') \}

= (\exists t_s, t_v, u \in P[t_s \leftarrow \langle B, tock \rangle \leftarrow t_v/\langle t \rangle] \land \neg P[t_s \leftarrow \langle a \rangle/\langle t \rangle]) Q[u/\langle t \rangle] \land (s \leftarrow \langle B, tock \rangle \leftarrow v) \in (t_s \leftarrow \langle B, tock \rangle \leftarrow t_v) |A| (u_s \leftarrow \langle B, tock \rangle \leftarrow u_v)\}

= \{ T3(P) \}

= (\exists t_s, t_v, u_s, u_v \in P[t_s \leftarrow \langle B \cup \{ a \}, tock \rangle \leftarrow t_v/\langle t \rangle] \land Q[u_s/\langle t \rangle] \land (s \leftarrow \langle B, tock \rangle \leftarrow v) \in (t_s \leftarrow \langle B, tock \rangle \leftarrow t_v) |A| (u_s \leftarrow \langle B, tock \rangle \leftarrow u_v)\}

= \{ definition trace parallel \}

= (\exists t_s, t_v, u_s, u_v \in P[t_s \leftarrow \langle B \cup \{ a \}, tock \rangle \leftarrow t_v/\langle t \rangle] Q[u_s/\langle t \rangle] \land (s \leftarrow \langle B \cup \{ a \}, tock \rangle \leftarrow v) \in (t_s \leftarrow \langle B \cup \{ a \}, tock \rangle \leftarrow t_v) |A| (u_s \leftarrow \langle B, tock \rangle \leftarrow u_v)\}

= \{ renaming: \(t = t_s \parallel \langle B \cup \{ a \}, tock \rangle \leftarrow t_v; \(u = u_s \leftarrow \langle B, tock \rangle \leftarrow u_v\) \}

= \{ substitution \}

= (\exists t, u \in P[t/\langle t \rangle] \land Q[u/\langle t \rangle] \land \langle t \rangle \in t |A| u)[s \leftarrow \langle B \cup \{ a \}, tock \rangle \leftarrow v/\langle t \rangle]')\]

= \{ definition parallel; \(a \in A\) \}

= (P[|A| \rightarrow Q][s \leftarrow \langle B \cup \{ a \}, tock \rangle \leftarrow v/\langle t \rangle]')\}

\(\neg A \in A\) similar

6. Hiding

104
Given \( T_3(P) \), we want to prove that \( T_3(P \setminus A) \), \( i.e \) that

\[
(P \setminus A)[s \leadsto \langle B, tock \rangle \leadsto v/\text{tt}'] \land \neg (P \setminus A)[s \leadsto \langle a \rangle/\text{tt}']
\]

implies

\[
(P \setminus A)[s \leadsto \langle B \cup \{a\}, tock \rangle \leadsto v/\text{tt}']
\]

We consider the case \( a \not\in A \).

First, observe

\[
\neg (P \setminus A)[s \leadsto \langle a \rangle/\text{tt}']
= \{ \text{definition of hiding} \}
\neg (\exists t \bullet P[t/\text{tt}'] \land A \text{ urgent } t \land \text{tt}' = t \setminus A)[s \leadsto \langle a \rangle/\text{tt}']
= \{ \text{substitution} \}
\neg (\exists t \bullet P[t/\text{tt}'] \land A \text{ urgent } t \land s \leadsto \langle a \rangle = t \setminus A)
= \{ a \not\in A; \text{renaming: } t = t_s \leadsto \langle a \rangle \}
\neg (\exists t_s \bullet P[t_s \leadsto \langle a \rangle/\text{tt}'] \land A \text{ urgent } t_s \leadsto \langle a \rangle \land s \leadsto \langle a \rangle = (t_s \leadsto \langle a \rangle) \setminus A)
= \{ \text{predicate calculus} \}
\forall t_s \bullet \neg P[t_s \leadsto \langle a \rangle/\text{tt}'] \lor \neg A \text{ urgent } t_s \leadsto \langle a \rangle \lor s \leadsto \langle a \rangle \not= (t_s \leadsto \langle a \rangle) \setminus A)
\]

Now,
\((P \setminus A)[s \triangleright (B, \text{tock}) \triangleright v/tt']\) 

= \{ \text{definition of hiding} \}

\((\exists t \bullet P[t/\text{tt}'] \land A \text{ urgent } tt = t \setminus A)[s \triangleright (B, \text{tock}) \triangleright v/tt']\)

= \{ \text{substitution} \}

\((\exists t \bullet P[t/\text{tt}'] \land A \text{ urgent } t \land s \triangleright (B, \text{tock}) \triangleright v = t \setminus A)\)

= \{ \text{renaming: } t = t_s \triangleright (B \cup A, \text{tock}) \triangleright t_v \}

\((\exists t_s, t_v \bullet P[t_s \triangleright (B \cup A, \text{tock}) \triangleright t_v/\text{tt}'] \land A \text{ urgent } t_s \triangleright (B \cup A, \text{tock}) \triangleright t_v \land s \triangleright (B, \text{tock}) \triangleright v = (t_s \triangleright (B \cup A, \text{tock}) \triangleright t_v) \setminus A)\)

\(\{ a \not\in A; \text{observation: } (s = t_s \setminus A) = (s \triangleright \langle a \rangle = t_s \triangleright \langle a \rangle / \setminus A)\}\)

\((\exists t_s, t_v \bullet P[t_s \triangleright (B \cup A, \text{tock}) \triangleright t_v/\text{tt}'] \land A \text{ urgent } t_s \triangleright (B \cup A, \text{tock}) \triangleright t_v \land s \triangleright (B, \text{tock}) \triangleright v = (t_s \triangleright (B \cup A, \text{tock}) \triangleright t_v) \setminus A)\)

\(\neg P[t_s \triangleright \langle a \rangle / \text{tt}'] \land A \text{ urgent } t_s \triangleright \langle a \rangle\}

\((\exists t_s, t_v \bullet P[t_s \triangleright (B \cup A \cup \{a\}, \text{tock}) \triangleright t_v/\text{tt}'] \land A \text{ urgent } t_s \triangleright (B \cup A \cup \{a\}, \text{tock}) \triangleright t_v \land s \triangleright (B, \text{tock}) \triangleright v = (t_s \triangleright (B \cup A \cup \{a\}, \text{tock}) \triangleright t_v) \setminus A)\)

\(\{ a \not\in A; \text{rename: } t = t_s \triangleright (B \cup A \cup \{a\}, \text{tock}) \triangleright t_v \}

\((\exists t \bullet P[t/\text{tt}'] \land A \text{ urgent } tt = t \setminus A)[s \triangleright (B \cup \{a\}, \text{tock}) \triangleright v/tt']\)

= \{ \text{substitution} \}

\((\exists t \bullet P[t/\text{tt}'] \land A \text{ urgent } t \land tt = t \setminus A)[s \triangleright (B \cup \{a\}, \text{tock}) \triangleright v/tt']\)

= \{ \text{definition hiding} \}

\((P \setminus A)[s \triangleright (B \cup \{a\}, \text{tock}) \triangleright v/tt']\)
7. Timeout

Case \( u \leq s \)

\[
\begin{align*}
(P \supset Q)[s \supset (A \cup \{a\}) \land v/\tt'] \land \neg (P \supset Q)[s \supset \langle a \rangle/\tt'] \\
= \{ \text{definition of timeout} \} \\
(\exists u \cdot u \leq \tt' \land \text{trace}(u) = \text{tock} \land P[u/\tt'] \land Q[tt' - u/\tt']) \\
\land \neg \left( \text{tock} \leq \text{trace}(\tt') \supset P \right) \\
= \{ \text{case assumption: } u \leq s \land \text{trace}(u) = \text{tock} \Rightarrow \text{tock} \leq \text{trace}(s \supset \langle A, \text{tock} \rangle \land v) \} \\
(\exists u \cdot u \leq \tt' \land \text{trace}(u) = \text{tock} \land P[u/\tt'] \land Q[tt' - u/\tt'])[s \supset \langle a \rangle/\tt'] \\
\land \neg (\exists w \cdot w \leq \tt' \land \text{trace}(w) = \text{tock} \land P[w/\tt'] \land Q[tt' - w/\tt'])[s \supset \langle a \rangle/\tt'] \\
= \{ \text{substitution} \} \\
(\exists u \cdot u \leq s \supset (A, \text{tock}) \land v \land \text{trace}(u) = \text{tock} \land P[u/\tt'] \land Q[(s \supset \langle A, \text{tock} \rangle \land v) - u/\tt'] \\
\land \neg (\exists w \cdot w \leq \tt' \land \text{trace}(w) = \text{tock} \land P[w/\tt'] \land Q[tt' - w/\tt'])[s \supset \langle a \rangle/\tt'] \\
= \{ \text{case assumption: } u \leq s, u \leq s \Rightarrow u \leq s \supset (A \cup \{a\}, \text{tock}) \land v \} \\
(\exists u \cdot u \leq s \supset (A \cup \{a\}) \land v \land \text{trace}(u) = \text{tock} \land P[u/\tt'] \land Q[(s \supset \langle A, \text{tock} \rangle \land v) - u/\tt'] \\
\land \neg (\exists w \cdot w \leq \tt' \land \text{trace}(w) = \text{tock} \land P[w/\tt'] \land Q[tt' - w/\tt'])[s \supset \langle a \rangle/\tt'] \\
= \{ \text{predicate calculus} \} \\
(\exists u \cdot u \leq s \supset (A \cup \{a\}) \land v \land \text{trace}(u) = \text{tock} \land P[u/\tt'] \land Q[(s \supset \langle A, \text{tock} \rangle \land v) - u/\tt'] \\
\land \neg (\exists w \cdot w \leq \tt' \land \text{trace}(w) = \text{tock} \land P[w/\tt'] \land Q[tt' - w/\tt'])[s \supset \langle a \rangle/\tt'] \\
= \{ \text{substitution} \} \\
(\exists u \cdot u \leq s \supset (A \cup \{a\}) \land v \land \text{trace}(u) = \text{tock} \land P[u/\tt'] \land Q[(s \supset \langle A, \text{tock} \rangle \land v) - u/\tt'] \\
\land \neg (\exists w \cdot w \leq \tt' \land \text{trace}(w) = \text{tock} \land P[w/\tt'] \land Q[tt' - w/\tt'])[s \supset \langle a \rangle/\tt'] \\
= \{ \text{predicate calculus, } w = u \} \\
(\exists u \cdot u \leq s \supset (A \cup \{a\}) \land v \land \text{trace}(u) = \text{tock} \land P[u/\tt'] \land Q[(s \supset \langle A, \text{tock} \rangle \land v) - u/\tt'] \\
\land \neg (u \leq s \supset \langle a \rangle) \lor \neg (\text{trace}(u) = \text{tock} \land \neg P[u/\tt'] \lor \neg Q[(s \supset \langle a \rangle) - w/\tt']) \\
\Rightarrow \neg (u \leq s \supset \langle a \rangle) \lor \neg (\text{trace}(u) = \text{tock} \land \neg P[u/\tt'] \lor \neg Q[(s \supset \langle a \rangle) - w/\tt'])
\end{align*}
\]
Definition 2 (Public)

Definition of timeout

\[
\exists u \cdot u \leq s \cap (A \cup \{a\}) \cap v \land \text{trace}(u) = \text{tock}^n \land \\
P[u/tt'] \land Q[(s - u) \cap (A, \text{tock}) \cap v/tt'] \\
\land \\
\neg (u \leq s \cap \langle a \rangle) \lor \neg \text{trace}(u) = \text{tock}^n \lor \neg P[u/tt'] \lor \neg Q[(s \cap \langle a \rangle) - u/tt'] \\
= \{ u \leq s \Rightarrow u \leq s \cap \langle a \rangle \} \\
\exists u \cdot u \leq s \cap (A \cup \{a\}) \cap v \land \text{trace}(u) = \text{tock}^n \land \\
P[u/tt'] \land Q[(s - u) \cap (A, \text{tock}) \cap v/tt'] \\
\land \\
\neg \text{trace}(u) = \text{tock}^n \lor \neg P[u/tt'] \lor \neg Q[(s \cap \langle a \rangle) - u/tt'] \\
= \{ a \land (\neg a \lor b) = a \land b \} \\
\exists u \cdot u \leq s \cap (A \cup \{a\}) \cap v \land \text{trace}(u) = \text{tock}^n \land \\
P[u/tt'] \land Q[(s - u) \cap (A, \text{tock}) \cap v/tt'] \\
\land \\
\neg P[u/tt'] \lor \neg Q[(s \cap \langle a \rangle) - u/tt'] \\
= \{ a \land (\neg a \lor b) = a \land b \} \\
\exists u \cdot u \leq s \cap (A \cup \{a\}) \cap v \land \text{trace}(u) = \text{tock}^n \land \\
P[u/tt'] \land Q[(s - u) \cap (A, \text{tock}) \cap v/tt'] \\
\land \\
\neg Q[(s \cap \langle a \rangle) - u/tt'] \\
= \{ u \leq s \Rightarrow (s \cap \langle a \rangle) - u = (s - u) \cap \langle a \rangle \} \\
\exists u \cdot u \leq s \cap (A \cup \{a\}) \cap v \land \text{trace}(u) = \text{tock}^n \land \\
P[u/tt'] \land Q[(s - u) \cap (A, \text{tock}) \cap v/tt'] \\
\land \\
\neg Q[(s - u) \cap \langle a \rangle)/tt'] \\
\Rightarrow \{ T3(Q) \} \\
\exists u \cdot u \leq s \cap (A \cup \{a\}) \cap v \land \text{trace}(u) = \text{tock}^n \land \\
P[u/tt'] \land Q[(s - u) \cap (A \cup \{a\}, \text{tock}) \cap v/tt'] \\
= \{ \text{trace arithmetic} \} \\
\exists u \cdot u \leq s \cap (A \cup \{a\}) \cap v \land \text{trace}(u) = \text{tock}^n \land \\
P[u/tt'] \land Q[(s \cap (A \cup \{a\}, \text{tock}) \cap v - u/tt'] \\
= \{ \text{substitution} \} \\
\exists u \cdot u \leq tt' \land \text{trace}(u) = \text{tock}^n \land P[u/tt'] \land Q[tt' - u/tt')] \cap (s \cap (A \cup \{a\}) \cap v/tt'] \\
= \{ \text{definition of conditional; tock}^n \leq tt' \} \\
(\exists u \cdot u \leq tt' \land \text{trace}(u) = \text{tock}^n \land P[u/tt'] \land Q[tt' - u/tt']) \\
\frac{\langle \text{tock}^n \leq \text{trace}(tt') \Rightarrow }{P} \\
\Rightarrow [s \cap (A \cup \{a\}) \cap v/tt'] \\
= \{ \text{definition of timeout} \} \\
(P \Rightarrow Q)[s \cap (A \cup \{a\}) \cap v/tt']
\[(P \triangleright Q)[s \sqsubseteq (A \cup \{a\}) \triangleright v / tt'] \land \neg (P \triangleright Q)[s \sqsubseteq (a) \triangleright tt']\]

= \{ definition of timeout \}

\[ (\exists u \cdot u \leq tt' \land \text{trace}(u) = \text{tock}^n \land P[u/\text{tt'}] \land Q[\text{tt'} - u/\text{tt'}]) \]

\[ \land \]

\[ (\exists w \cdot w \leq tt' \land \text{trace}(w) = \text{tock}^n \land P[w/\text{tt'}] \land Q[\text{tt'} - w/\text{tt'}]) \]

\[ \neg \left( \frac{\text{tock}^n \leq \text{trace}(\text{tt'})}{P} \right) \]

\[ [s \sqsubseteq (A, \text{tock}) \triangleright v / \text{tt'}] \land \neg \left( \frac{\text{tock}^n \leq \text{trace}(\text{tt'})}{P} \right) \]

= \{ substitution \}

\[ (\exists u \cdot u \leq s \sqsubseteq (A, \text{tock}) \triangleright v \land \text{trace}(u) = \text{tock}^n \land P[u/\text{tt'}] \land Q[(s \sqsubseteq (A, \text{tock}) \triangleright v) - u/\text{tt'}]) \]

\[ \land \]

\[ (\exists w \cdot w \leq s \sqsubseteq (a) \land \text{trace}(w) = \text{tock}^n \land P[w/\text{tt'}] \land Q[(s \sqsubseteq (a) - w/\text{tt'})]) \]

\[ \neg \left( \frac{\text{tock}^n \leq \text{trace}(s \sqsubseteq (a))}{P} \right) \]

= \{ case assumption: \( u = s \sqsubseteq (A, \text{tock}) \Rightarrow \text{tock}^n \leq s \sqsubseteq (A, \text{tock}) \triangleright v \} \]

\[ (\exists u \cdot u \leq s \sqsubseteq (A, \text{tock}) \triangleright v \land \text{trace}(u) = \text{tock}^n \land P[u/\text{tt'}] \land Q[(s \sqsubseteq (A, \text{tock}) \triangleright v) - u/\text{tt'}]) \]

\[ \land \]

\[ (\exists w \cdot w \leq s \sqsubseteq (a) \land \text{trace}(w) = \text{tock}^n \land P[w/\text{tt'}] \land Q[(s \sqsubseteq (a) - w/\text{tt'})]) \]

\[ \neg \left( \frac{\text{tock}^n \leq \text{trace}(s \sqsubseteq (a))}{P} \right) \]

= \{ case assumption: \( u = s \sqsubseteq (A, \text{tock}) \land \text{trace}(u) = \text{tock}^n \Rightarrow \neg (\text{tock}^n \leq s \sqsubseteq (a)) \} \]

\[ (\exists u \cdot u \leq s \sqsubseteq (A, \text{tock}) \triangleright v \land \text{trace}(u) = \text{tock}^n \land P[u/\text{tt'}] \land Q[(s \sqsubseteq (A, \text{tock}) \triangleright v) - u/\text{tt'}]) \]

\[ \land \neg P[s \sqsubseteq (a) / \text{tt'}] \]

= \{ case assumption: \( u = s \sqsubseteq (A, \text{tock}) \}

\[ (s \sqsubseteq (A, \text{tock}) \leq s \sqsubseteq (A, \text{tock}) \triangleright v \land \text{trace}(s \sqsubseteq (A, \text{tock})) = \text{tock}^n \land P[s \sqsubseteq (A, \text{tock}) / \text{tt'}] \land Q[(s \sqsubseteq (A, \text{tock}) \triangleright v) - s \sqsubseteq (A, \text{tock}) / \text{tt'}]) \]

\[ \land \neg P[s \sqsubseteq (a) / \text{tt'}] \]

= \{ T3(P) \}

\[ (s \sqsubseteq (A, \text{tock}) \leq s \sqsubseteq (A, \text{tock}) \triangleright v \land \text{trace}(s \sqsubseteq (A, \text{tock})) = \text{tock}^n \land P[s \sqsubseteq (A \cup \{a\}, \text{tock}) / \text{tt'}] \land Q[(s \sqsubseteq (A, \text{tock}) \triangleright v) - s \sqsubseteq (A, \text{tock}) / \text{tt'}]) \]

= \{ trace arithmetic \}

\[ (s \sqsubseteq (A \cup \{a\}, \text{tock}) \leq s \sqsubseteq (A \cup \{a\}, \text{tock}) \triangleright v \land \text{trace}(s \sqsubseteq (A \cup \{a\}, \text{tock})) = \text{tock}^n \land P[s \sqsubseteq (A \cup \{a\}, \text{tock}) / \text{tt'}] \land Q[(s \sqsubseteq (A, \text{tock}) \triangleright v) - s \sqsubseteq (A, \text{tock}) / \text{tt'}]) \]

= \{ trace arithmetic \}

\[ (s \sqsubseteq (A \cup \{a\}, \text{tock}) \leq s \sqsubseteq (A \cup \{a\}, \text{tock}) \triangleright v \land \text{trace}(s \sqsubseteq (A \cup \{a\}, \text{tock})) = \text{tock}^n \land P[s \sqsubseteq (A \cup \{a\}, \text{tock}) / \text{tt'}] \land Q[(s \sqsubseteq (A, \text{tock}) \triangleright v) - s \sqsubseteq (A, \text{tock}) / \text{tt'}]) \]

= \{ trace arithmetic \}

\[ (s \sqsubseteq (A \cup \{a\}, \text{tock}) \leq s \sqsubseteq (A \cup \{a\}, \text{tock}) \triangleright v \land \text{trace}(s \sqsubseteq (A \cup \{a\}, \text{tock})) = \text{tock}^n \land P[s \sqsubseteq (A \cup \{a\}, \text{tock}) / \text{tt'}] \land Q[(s \sqsubseteq (A \cup \{a\}, \text{tock}) \triangleright v) - s \sqsubseteq (A \cup \{a\}, \text{tock}) / \text{tt'}]) \]

\[ \land P[s \sqsubseteq (A \cup \{a\}, \text{tock}) / \text{tt'}] \land Q[(s \sqsubseteq (A \cup \{a\}, \text{tock}) \triangleright v) - s \sqsubseteq (A \cup \{a\}, \text{tock}) / \text{tt'}]) \]
\[
\begin{align*}
&\left( s \circlearrowleft ( A \cup \{a\}, \text{tock} ) \preceq s \circlearrowleft ( A \cup \{a\}, \text{tock} ) \circlearrowright v \land \text{trace}(s \circlearrowleft ( A \cup \{a\}, \text{tock} )) = \text{tock}^n \\
&\quad \land P[s \circlearrowleft ( A \cup \{a\}, \text{tock} ) \circlearrowright tt'] \land Q[(s \circlearrowleft ( A \cup \{a\}, \text{tock} ) \circlearrowright v ) - s \circlearrowleft ( A \cup \{a\}, \text{tock} ) / tt']
\end{align*}
\]

{definition of conditional}

\[
\begin{align*}
&\left( s \circlearrowleft ( A \cup \{a\}, \text{tock} ) \preceq s \circlearrowleft ( A \cup \{a\}, \text{tock} ) \circlearrowright v \land \text{trace}(s \circlearrowleft ( A \cup \{a\}, \text{tock} )) = \text{tock}^n \\
&\quad \land P[s \circlearrowleft ( A \cup \{a\}, \text{tock} ) \circlearrowright tt'] \land Q[(s \circlearrowleft ( A \cup \{a\}, \text{tock} ) \circlearrowright v ) - s \circlearrowleft ( A \cup \{a\}, \text{tock} ) / tt']
\end{align*}
\]

\[
\begin{align*}
&\preceq \text{tock}^n \leq \text{trace}(s \circlearrowleft ( A \cup \{a\}, \text{tock} ) \circlearrowright v ) \bowtie
\end{align*}
\]

\[
\begin{align*}
P
\end{align*}
\]

\[
\begin{align*}
&\equiv \{ \text{substitution} \} \\
&\left( u \leq s \circlearrowleft ( A \cup \{a\}, \text{tock} ) \circlearrowright v \land \text{trace}(u) = \text{tock}^n \\
&\quad \land P[u / tt'] \land Q[(s \circlearrowleft ( A \cup \{a\}, \text{tock} ) \circlearrowright v ) - u / tt']
\end{align*}
\]

\[
\begin{align*}
&\preceq \text{tock}^n \leq \text{trace}(tt') \bowtie
\end{align*}
\]

\[
\begin{align*}
P
\end{align*}
\]

\[
\begin{align*}
&\equiv \{ \text{exists-introduction} \} \\
&\left( \exists u \bullet u \leq tt' \land \text{trace}(u) = \text{tock}^n \\
&\quad \land P[u / tt'] \land Q[tt' - u / tt']
\end{align*}
\]

\[
\begin{align*}
&\preceq \text{tock}^n \leq \text{trace}(tt') \bowtie
\end{align*}
\]

\[
\begin{align*}
P
\end{align*}
\]

\[
\begin{align*}
&\equiv \{ \text{definition of timeout} \} \\
&P \Rightarrow Q)[s \circlearrowleft ( A \cup \{a\}, \text{tock} ) \circlearrowright v / tt']
\end{align*}
\]
Case $\neg (u \leq s \triangleleft (A, \text{tock}))$

\[
(P \multimap Q)[s \triangleleft (A, \text{tock}) \triangleleft v/tt'] \land \neg (P \multimap Q)[s \triangleleft \langle a \rangle/tt']
\]
\[
= \{ \text{definition of timeout} \}
\]
\[
\left( \exists u \bullet u \leq tt' \land \text{trace}(u) = \text{tock}^n \land P[u/tt'] \land Q[tt' - u/tt'] \right)
\]
\[
\text{\quad} \supset \text{tock}^n \leq \text{trace}(tt') \supset
\]
\[
\text{\quad} [s \triangleleft (A, \text{tock}) \triangleleft v/tt']
\]
\[
\land
\]
\[
\left( \exists w \bullet w \leq tt' \land \text{trace}(w) = \text{tock}^n \land P[w/tt'] \land Q[tt' - w/tt'] \right)
\]
\[
\text{\quad} \supset \text{tock}^n \leq \text{trace}(tt') \supset
\]
\[
\text{\quad} [s \triangleleft \langle a \rangle/tt']
\]
\[
= \{ \text{substitution} \}
\]
\[
\left( \exists u \bullet u \leq s \triangleleft (A, \text{tock}) \triangleleft v \land \text{trace}(u) = \text{tock}^n \right)
\]
\[
\land P[u/tt'] \land Q[(s \triangleleft (A, \text{tock}) \triangleleft v) - u/tt']
\]
\[
\text{\quad} \supset \text{tock}^n \leq \text{trace}(s \triangleleft (A, \text{tock}) \triangleleft v) \supset
\]
\[
\text{\quad} P[s \triangleleft (A, \text{tock}) \triangleleft v/tt']
\]
\[
\land
\]
\[
\left( \exists w \bullet w \leq s \triangleleft (a) \land \text{trace}(w) = \text{tock}^n \land P[w/tt'] \land Q[(s \triangleleft (a) - w/tt')] \right)
\]
\[
\text{\quad} \supset \text{tock}^n \leq \text{trace}(s \triangleleft (a)) \supset
\]
\[
\text{\quad} P[s \triangleleft (A, \text{tock}) \triangleleft v/tt'] \land \neg P[s \triangleleft (a)/tt']
\]
\[
= \{ \text{T3}(P) \}
\]
\[
\frac{\neg (u \leq s \triangleleft (A, \text{tock})) \land u = \text{tock}^n \Rightarrow \neg (\text{tock}^n \leq \text{trace}(s \triangleleft (A, \text{tock}) \triangleleft v))}{P[s \triangleleft (A, \text{tock}) \triangleleft v/tt'] \land \neg P[s \triangleleft (a)/tt']}
\]
\[
= \{ \text{definition of conditional; } \neg \text{tock}^n \leq \text{trace}(s \triangleleft (A \cup \{a\}, \text{tock}) \triangleleft v) \}
\]
\[
\left( \exists u \bullet u \leq s \triangleleft (A \cup \{a\}, \text{tock}) \triangleleft v \land \text{trace}(u) = \text{tock}^n \right)
\]
\[
\land P[u/tt'] \land Q[(s \triangleleft (A \cup \{a\}, \text{tock}) \triangleleft v) - u/tt']
\]
\[
\text{\quad} \supset \text{tock}^n \leq \text{trace}(s \triangleleft (A \cup \{a\}, \text{tock}) \triangleleft v) \supset
\]
\[
\text{\quad} P[s \triangleleft (A \cup \{a\}, \text{tock}) \triangleleft v/tt']
\]
\[
= \{ \text{substitution} \}
\]
\[
\left( \exists u \bullet u \leq tt' \land \text{trace}(u) = \text{tock}^n \right)
\]
\[
\land P[u/tt'] \land Q[tt' - u/tt']
\]
\[
\text{\quad} \supset \text{tock}^n \leq \text{trace}(tt') \supset
\]
\[
\text{\quad} [s \triangleleft (A \cup \{a\}, \text{tock}) \triangleleft v/tt']
\]
\[
= \{ \text{definition of timeout} \}
\]
\[
(P \multimap Q)[s \triangleleft (A \cup \{a\}, \text{tock}) \triangleleft v/tt']
\]

**Definition A.1.2** Lowe & Ouaknine Axiom

\[ T4 \quad P[tt' \setminus \{\}, \text{tock}]/tt'] \]

*Time can always pass.*
Theorem A.1.2 Every program operator apart from hiding satisfies $T_4$.

1. **STOP**

   \[
   T_4^{STOP} = \{ \text{definition: STOP } \} 
   \]

   \[
   T_4^{T_0(\text{trace}(tt') \in tock^*)} = \{ T_4 \}
   \]

   \[
   (T_0(\text{trace}(tt') \in tock^*))[tt' \sim (\{\}, tock)/tt'] 
   = \{ \text{substitution} \}
   \]

   \[
   T_0(\text{trace}(tt') \sim (\{\}, tock) \in tock^*) 
   = \{ \forall t \cdot t \in a^* \Rightarrow t \sim a \in a^* \}
   \]

   \[
   T_0(\text{trace}(tt') \in tock^*) 
   = \{ \text{definition} \}
   \]

   \[
   STOP 
   \]

2. **Prefix**

   Assume $T_4(P), a \rightarrow P$.

   Would like to prove $T_4(a \rightarrow P)$.

   \[
   T_4^{(a \rightarrow P)} = \{ \text{definition } a \rightarrow P \} 
   \]

   \[
   T_4^{T_0} = \left( \begin{array}{c}
   a \notin \text{refusals}(tt') \\
   \lhd \text{trace}(tt') \in tock^* \\
   a = \text{head}(\text{trace(idesuffix}(tt')) ) \\
   a \notin \text{refusals}(idlerefifix(tt')) \\
   P[\text{tail(idesuffix}(tt'))/tt'] \\
   \end{array} \right) 
   \]

   \[
   = \{ \text{definition } T_4 \}
   \]

   \[
   T_0 = \left( \begin{array}{c}
   a \notin \text{refusals}(tt') \\
   \lhd \text{trace}(tt') \in tock^* \\
   a = \text{head}(\text{trace(idesuffix}(tt')) ) \\
   a \notin \text{refusals}(idlerefifix(tt')) \\
   P[\text{tail(idesuffix}(tt'))/tt'] \\
   \end{array} \right) 
   \]

   \[
   = \{ \text{substitution} \}
   \]

   \[
   T_0 = \left( \begin{array}{c}
   a \notin \text{refusals}(tt' \sim (\{\}, tock)) \\
   \lhd \text{trace}(tt' \sim (\{\}, tock)) \in tock^* \\
   a = \text{head}(\text{trace(idesuffix}(tt' \sim (\{\}, tock)))) ) \\
   a \notin \text{refusals}(idlerefifix(tt' \sim (\{\}, tock)) ) \\
   P[\text{tail(idesuffix}(tt' \sim (\{\}, tock))/tt'] \\
   \end{array} \right) 
   \]

   \[
   = \{ \forall t \cdot a \notin \text{refusals}(t \sim (\{\}, tock)) \Leftrightarrow a \notin \text{refusals}(t) \}
   \]
\[ a \notin \text{refusals}(tt') \]
\[ \text{< trace}(tt' \cap \{(\epsilon\,),\, tock\}) \in \text{tock*} \]

**Definition 2 (Public)**

\[ (a = \text{head}((\text{idlesuffix}(tt') \cap \{(\epsilon\,),\, tock\})) \cap \text{refusals}(idleprefix(tt)) \cap \text{tock*} \]
\[ \text{< tock*} \]

\[ \text{definition of tock*} \mapsto \text{idlesuffix}(t \cap s) = \text{idlesuffix}(t \cap s) \]

\[ \text{< tock*} \]

3. **Internal choice**

Assume \( P[tt' \cap \{(\epsilon\,),\, tock\}/tt'] \cap Q[tt' \cap \{(\epsilon\,),\, tock\}/tt'] \)

WTP \( (P \cap Q)[tt' \cap \{(\epsilon\,),\, tock\}/tt'] \)

\[ T4(P \cap Q) \]

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

\[ a \mapsto P \]
\[(P \sqcap Q)[tt' \sim \{\}, tock]/tt']\]
= \{ definition of \(\sqcap\) \}
\[(P \lor Q)[tt' \sim \{\}, tock]/tt']\]
= \{ distributivity \}
\[P[tt' \sim \{\}, tock]/tt'] \lor Q[tt' \sim \{\}, tock]/tt']\]
= \{ assumptions: \(T4(P)\), \(T4(Q)\) \}
\[P \lor Q\]
= \{ definition: internal choice \}
\[P \sqcap Q\]

4. **external choice**

Assume \(P[tt' \sim \{\}, tock]/tt'] \land Q[tt' \sim \{\}, tock]/tt']\)

WTP \((P \Box Q)[tt' \sim \{\}, tock]/tt']\)

\[\textbf{T4}(P \Box Q)\]
= \{ definition \(T4\) \}
\[(P \Box Q)[tt' \sim \{\}, tock]/tt']\]
= \{ definition external choice \}
\[((P \land Q)[idleprefix(tt')/tt'] \land (P \lor Q))[tt' \sim \{\}, tock]/tt']\]
= \{ substitution \}
\[((P \land Q)[idleprefix(tt' \sim \{\}, tock)]/tt'] \land (P \lor Q)[tt' \sim \{\}, tock]/tt']\]
= \{ propositional calculus \}
\[P[idleprefix(tt' \sim \{\}, tock)]/tt' \land Q[idleprefix(tt' \sim \{\}, tock)]/tt'] \land
(P[tt' \sim \{\}, tock]/tt' \lor Q[tt' \sim \{\}, tock]/tt'])\]
= \{ assumptions: \(T4(P)\), \(T4(Q)\) \}
\[P[idleprefix(tt' \sim \{\}, tock)]/tt' \land Q[idleprefix(tt' \sim \{\}, tock)]/tt'] \land (P \lor Q)\]
= \{ \forall t \cdot t \in tock' \Rightarrow idleprefix(t \sim \{\}, tock) = idleprefix(t \sim \{\}, tock) \}
\[P[idleprefix(tt' \sim \{\}, tock)/tt'] \land Q[idleprefix(tt' \sim \{\}, tock)/tt'] \land (P \lor Q)\]
= \{ assumptions: \(T4(P)\), \(T4(Q)\) \}
\[P[idleprefix(tt')/tt'] \land Q[idleprefix(tt')/tt'] \land (P \lor Q)\]
= \{ predicate calculus \}
\[(P \land Q)[idleprefix(tt')/tt'] \land (P \lor Q)\]
= \{ definition \}
\[P \Box Q\]

5. **Parallel**

Assume \(\textbf{T4}(P)\), \(\textbf{T4}(Q)\), \(P \parallel_A Q\)

WTP \(\textbf{T4}(P \parallel_A Q)\)

\[\textbf{T4}(P \parallel_A Q)\]
D23.3-1 - CML Definition 2 (Public)

6. Timeout

Assume \( T4(P), T4(Q), T4(P \gg Q) \)

Case 1. tock^n \leq trace(tt') Case 2. tock^n = trace(tt') \sim \{tock\} Case 3. tock^n > trace(tt') \sim \{tock\}

case 1

\[
T4(P \gg Q) = \{ \text{definition timeout} \} \\
T4 \left( \left( \exists u \bullet u \leq tl' \land \text{trace}(u) = \text{tock}^n \land P[u/tl'] \land Q[tl' - u/tl'] \right) \right) \\
\left( \text{tock}^n \leq \text{trace}(tt') \gg \right) \\
\subseteq \{ \text{definition } T4 \} \\
\left( \exists u \bullet u \leq tl' \land \text{trace}(u) = \text{tock}^n \land P[u/tl'] \land Q[tl' - u/tl'] \right) \\
\left( \text{tock}^n \leq \text{trace}(tt') \gg \right) \\
\{ tl' \sim \{\}, \text{tock} \}/tt' \right) \\
= \{ \text{substitution} \} \\
\left( \exists u \bullet u \leq tl' \sim \{\}, \text{tock} \right) \land \left( \text{trace}(u) = \text{tock}^n \right) \land \left( P[u/tl'] \land Q[(tt' \sim \{\}, \text{tock})] - u/tl' \right) \\
\left( \text{tock}^n \leq \text{trace}(tt') \gg \right) \\
P(tt' \sim \{\}, \text{tock}) \gg \\
\left( \text{trace}(a \sim \{\}, \text{tock}) = \text{trace}(a) \sim \text{trace}(\{\}, \text{tock}) = \text{trace}(a) \sim \{\text{tock}\} \right) \\
\left( \exists u \bullet u \leq tl' \sim \{\}, \text{tock} \right) \land \left( \text{trace}(u) = \text{tock}^n \right) \land \left( P[u/tl'] \land Q[(tt' \sim \{\}, \text{tock})] - u/tl' \right) \\
\left( \text{tock}^n \leq \text{trace}(tt') \sim \{\text{tock}\} \gg \right) \\
P(tt' \sim \{\}, \text{tock}) \gg \\
\left( \text{(i) case assumption (ii) } a^n \leq b \Rightarrow a^n \leq b \sim a \right) \}
\[
\exists u \cdot u \leq tt' \cap \{\}, \text{tock} \land (\text{trace}(u) = \text{tock}^n) \\
\land P[u/\text{tt'}] \land Q((\text{tt'} \cap \{\}, \text{tock}) - u) \land \text{tock}^n \leq \text{trace(\text{tt'})} \land \text{tock}^n \leq \text{trace(\text{tt'})} \cap \{\text{tock}\}
\]
\[
= \{ r \circ b \triangleright s \land b = r \land b \}
\]
\[
\exists u \cdot u \leq tt' \cap \{\}, \text{tock} \land (\text{trace}(u) = \text{tock}^n) \land \\
P[u/\text{tt'}] \land Q((\text{tt'} \cap \{\}, \text{tock}) - u) \land \text{tock}^n \leq \text{trace(\text{tt'})} \land \text{tock}^n \leq \text{trace(\text{tt'})} \cap \{\text{tock}\}
\]
\[
= \{ \text{substitution} \}
\]
\[
\exists u \cdot u \leq tt' \cap \{\}, \text{tock} \land (\text{trace}(u) = \text{tock}^n) \land \\
P[u/\text{tt'}] \land Q((\text{tt'} \cap \{\}, \text{tock}) - u) \land \text{tock}^n \leq \text{trace(\text{tt'})}
\]
\[
\Leftarrow \{ a \Rightarrow a \land b \}
\]
\[
\exists u \cdot u \leq tt' \cap \{\}, \text{tock} \land (\text{trace}(u) = \text{tock}^n) \land \\
P[u/\text{tt'}] \land Q((\text{tt'} \cap \{\}, \text{tock}) - u) \land \text{tock}^n \leq \text{trace(\text{tt'})}
\]
\[
\Leftarrow \{ a \leq b \Rightarrow (a - b) \cap c = (a \cap c) - b \}
\]
\[
\exists u \cdot u \leq tt' \cap \{\}, \text{tock} \land (\text{trace}(u) = \text{tock}^n) \land \\
P[u/\text{tt'}] \land Q((\text{tt'} - u) \cap \{\}, \text{tock}) \land \text{tock}^n \leq \text{trace(\text{tt'})}
\]
\[
= \{ \text{T4}(Q) \}
\]
\[
\exists u \cdot u \leq tt' \cap \{\}, \text{tock} \land (\text{trace}(u) = \text{tock}^n) \land \\
P[u/\text{tt'}] \land Q(\text{tt'} - u) \land \text{tock}^n \leq \text{trace(\text{tt'})}
\]
\[
= \{ \text{substitution} \}
\]
\[
\exists u \cdot u \leq tt' \cap \{\}, \text{tock} \land (\text{trace}(u) = \text{tock}^n) \land \\
P[u/\text{tt'}] \land Q(\text{tt'} - u/\text{tt'}) \land \text{tock}^n \leq \text{trace(\text{tt'})}
\]
\[
\Leftarrow \{ a \leq b \Rightarrow a \leq b \cap c \}
\]
\[
\exists u \cdot u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land \\
P[u/\text{tt'}] \land Q(\text{tt'} - u/\text{tt'}) \land \text{tock}^n \leq \text{trace(\text{tt'})}
\]
\[
= \{ r \circ b \triangleright s \land b = r \land b \}
\]
\[
\exists u \cdot u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[u/\text{tt'}] \land Q(\text{tt'} - u/\text{tt'})
\]
\[
\Leftarrow \text{tock}^n \leq \text{trace(\text{tt'})} \triangleright
\]
\[
\]
Assume \text{T4}(P), \text{T4}(Q), \text{T4}(P \triangleright Q)
\]
Case 2. tock\textsuperscript{n} = \text{trace(\text{tt'})} \cap \{\text{tock}\}
\]
\[
\text{T4}(P \triangleright Q)
\]

116
\[
\begin{align*}
&= \{ \text{definition timeout} \} \\
&= \{ \text{definition } \mathcal{T}4 \} \\
&= \{ \text{substitution} \} \\
&= \{ \text{case assumption: } \text{tokn} = \text{trace}(tt') \land \text{tokk} \} \\
&= \{ \text{substitution of equals} \} \\
&= \{ \text{substitution of equals} \} \\
&= \{ \text{substitution of equals} \} \\
\end{align*}
\]
\[ P \land Q(\cdot / tt) \land \text{tock}^n = \text{trace}(tt') \cap \langle \text{tock} \rangle \]
\[ = \{ \text{one point rule} \} \]
\[ P \land \text{tock}^n = \text{trace}(tt') \cap \langle \text{tock} \rangle \]
\[ = \{ T2(Q) \} \]
\[ P \land \text{tock}^n = \text{trace}(tt') \cap \langle \text{tock} \rangle \]
\[ = \{ a \land b \land (a \Rightarrow b) = a \} \]
\[ P \land \text{tock}^n = \text{trace}(tt') \cap \langle \text{tock} \rangle \land \neg (\text{tock}^n \leq \text{trace}(tt')) \]
\[ = \{ (r \land b \lor s) \land \neg b = s \land \neg b \} \]
\[ \exists u \cdot u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[u/ tt'] \land Q[tt' - u/ tt'] \]
\[ \langle \neg \text{tock}^n \leq \text{trace}(tt') \rangle \]
\[ \langle \text{tock}^n = \text{trace}(tt') \cap \langle \text{tock} \rangle \land \neg (\text{tock}^n \leq \text{trace}(tt')) \rangle \]
\[ = \{ \text{case assumption: tock}^n = \text{trace}(tt') \cap \langle \text{tock} \rangle \} \]
\[ \exists u \cdot u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[u/ tt'] \land Q[tt' - u/ tt'] \]
\[ \langle \neg \text{tock}^n \leq \text{trace}(tt') \rangle \]
\[ = \{ \text{definition timeout} \} \]
\[ (P \triangleright Q) \]

**case 3. tock}^n > \text{trace}(tt') \cap \langle \text{tock} \rangle**

\[ T4(P \triangleright Q) \]
\[ = \{ \text{definition timeout} \} \]
\[ T4 \left( \exists u \cdot u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[u/ tt'] \land Q[tt' - u/ tt'] \right) \]
\[ \langle \neg \text{tock}^n \leq \text{trace}(tt') \rangle \]
\[ = \{ \text{definition T4} \} \]
\[ (\exists u \cdot u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[u/ tt'] \land Q[tt' - u/ tt']) \]
\[ \langle \neg \text{tock}^n \leq \text{trace}(tt') \rangle \]
\[ = \{ \text{substitution} \} \]
\[ (\exists u \mid u \leq tt' \cap \langle \{ \}, \text{tock} \rangle \land (\text{trace}(u) = \text{tock}^n) \land P[u/ tt'] \land Q[(tt' \cap \langle \{ \}, \text{tock} \rangle) - u/ tt']) \]
\[ \langle \neg \text{tock}^n \leq \text{trace}(tt' \cap \langle \{ \}, \text{tock} \rangle) \rangle \]
\[ P(tt' \cap \langle \{ \}, \text{tock} \rangle) \]
\[ = \{ \text{case assumption: tock}^n > \text{trace}(tt') \cap \langle \text{tock} \rangle \} \]
\[\begin{align*}
(\exists u \cdot u \leq tt' \land (\text{trace}(u) = \text{tokc}^n)) \\
\land P[u/\text{tokc}'] \land Q[tt' \land (\text{trace}(u) = \text{tokc}^n)] \\
\land \text{tokc}^n \leq \text{trace}(tt' \land (\text{trace}(u) = \text{tokc}^n)) \\
P(tt' \land (\text{trace}(u) = \text{tokc}^n))
\end{align*}\]

\[\text{tokc}^n > \text{trace}(tt') \land (\text{trace}(u) = \text{tokc}^n)\]

\[P \land \text{tokc}^n \land \text{trace}(tt') \land (\text{trace}(u) = \text{tokc}^n)\]

\[\text{substitution}\]

\[P[tt' \land (\text{trace}(u) = \text{tokc}^n)] \land \text{tokc}^n \land \text{trace}(tt') \land (\text{trace}(u) = \text{tokc}^n)\]

\[\text{definition timeout}\]

7. **Recursion** \(T4\) can be written as the conjunctive idempotent \(T4(P) = P \land P[tt' \land (\text{trace}(u) = \text{tokc}^n)]\). Recursion therefore satisfies \(T4\), by [12].
A.1.3 Zeno Freedom

Theorem 4.2.5 (Zeno freedom) Suppose that \( P \) is a time-guarded process, then for every \( k \) there is an \( n \), such that \( P \) is \( T_5 \)-healthy.

Proof A.1.3 Every program operator satisfies \( T_5 \)

1. \( STOP \)

\[
T_5(STOP)
= \{ T_5 \}
\]

\( \forall k. \exists n. \forall tt' \cdot STOP \Rightarrow (\#(tt' \downarrow \text{tock}) \leq k \Rightarrow \#(\text{trace}(tt')) \leq n)) \)

\( = \{ \text{definition } STOP \} \)

\( \forall k. \exists n. \forall tt' \cdot \text{trace}(tt') \in \text{tock}^* \Rightarrow (\#(tt' \downarrow \text{tock}) \leq k \Rightarrow \#(\text{trace}(tt')) \leq n)) \)

\( = \{ \text{witness } k = n; \ tt' \downarrow \text{tock} = \text{trace}(tt') \} \)

\( \forall tt' \cdot \text{trace}(tt') \in \text{tock}^* \Rightarrow (\#(tt' \downarrow \text{tock}) \leq \#(tt' \downarrow \text{tock}) \Rightarrow \#(\text{trace}(tt')) \leq \#(\text{trace}(tt')))) \)

\( \iff \{ \text{predicate calculus} \} \)

\( \forall tt' \cdot \text{trace}(tt') \in \text{tock}^* \Rightarrow \#(tt' \downarrow \text{tock}) = \#(tt' \downarrow \text{tock}) \Rightarrow \#(\text{trace}(tt')) = \#(\text{trace}(tt')) \)

\( = \{ \text{predicate calculus} \} \)

\( \forall tt' \cdot \text{trace}(tt') \in \text{tock}^* \land tt' \downarrow \text{tock} = \text{trace}(tt') \)

\( \iff \{ \forall\text{-elimination} \} \)

\( \text{trace}(tt') \in \text{tock}^* \land tt' \downarrow \text{tock} = \text{trace}(tt') \)

\( = \{ \text{trace}(tt') \in \text{tock}^* \Rightarrow tt' \downarrow \text{tock} = \text{trace}(tt') \} \)

\( \text{trace}(tt') \in \text{tock}^* \)

\( = \{ \text{definition } STOP \} \)

\( STOP \)

2. Prefix

- Case: \( \text{trace}(tt') \in \text{tock}^* \)

\[
T_5(a \rightarrow P)
= \{ \text{definition prefix} \}
\]

\[
\left( a \notin \text{refusals}(tt') \right.
\left. \triangleright \text{trace}(tt') \in \text{tock}^* \gtrless \right)
\]

\[
T_5
= \left( \begin{array}{l}
\left( a = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \land \\
\text{a} \notin \text{refusals}(\text{idleprefix}(tt')) \land \\
P[\text{tail}(\text{idlesuffix}(tt'))/tt']
\end{array} \right)
\]

\( = \{ \text{Case: } \text{trace}(tt') \in \text{tock}^* \} \)

\[
T_5(a \notin \text{refusals}(tt') \land \text{trace}(tt') \in \text{tock}^*)
= \{ \text{calculus} \}
\]

\( a \notin \text{refusals}(tt') \land T_5(\text{trace}(tt') \in \text{tock}^*) \)
\[ \text{Definition 2 (Public)} \]
\[
\begin{align*}
&\iff \{ T5(\text{STOP}) \} \\
&\quad \text{a} \notin \text{refusals}(tt') \land \text{trace}(tt') \in \text{tock}^* \\
&\quad = \{ \text{Case: trace}(tt') \in \text{tock}^* \} \\
&\quad \quad \text{a} \notin \text{refusals}(tt') \\
&\quad \quad \quad \land \text{trace}(tt') \in \text{tock}^* \land \text{trace}(tt') \\
&\quad \quad \quad \text{a} = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \land \\
&\quad \quad \quad \text{a} \notin \text{refusals}(\text{idleprefix}(tt')) \land \\
&\quad \quad \quad \text{P}[\text{tail}(\text{idlesuffix}(tt'))/tt'] \\
&\quad = \{ \text{definition prefix} \} \\
&\quad \quad \text{a} \to P \\
&\end{align*}
\]

- **Case:** \( \neg \text{trace}(tt') \in \text{tock}^* \)

\[
\begin{align*}
&\quad \text{T5}(a \to P) \\
&\quad = \{ \text{definition prefix} \} \\
&\quad \quad \text{a} \notin \text{refusals}(tt') \\
&\quad \quad \quad \land \text{trace}(tt') \in \text{tock}^* \land \text{trace}(tt') \\
&\quad \quad \quad \text{a} = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \land \\
&\quad \quad \quad \text{a} \notin \text{refusals}(\text{idleprefix}(tt')) \land \\
&\quad \quad \quad \text{P}[\text{tail}(\text{idlesuffix}(tt'))/tt'] \\
&\quad = \{ \text{Case: } \neg \text{trace}(tt') \in \text{tock}^* \} \\
&\quad \quad \quad \text{a} = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \land \\
&\quad \quad \quad \text{a} \notin \text{refusals}(\text{idleprefix}(tt')) \land \\
&\quad \quad \quad \text{P}[\text{tail}(\text{idlesuffix}(tt'))/tt'] \\
&\quad = \{ \text{T5-conjunction} \} \\
&\quad \quad \quad \neg \text{trace}(tt') \in \text{tock}^* \land a = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \land \\
&\quad \quad \quad a \notin \text{refusals}(\text{idleprefix}(tt')) \land \text{T5}[\text{P}[\text{tail}(\text{idlesuffix}(tt'))/tt']] \\
&\quad \iff \{ \text{T5}(P) \} \\
&\quad \quad \neg \text{trace}(tt') \in \text{tock}^* \land a = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \land \\
&\quad \quad a \notin \text{refusals}(\text{idleprefix}(tt')) \land \text{P}[\text{tail}(\text{idlesuffix}(tt'))/tt'] \\
&\quad = \{ \text{Case: } \neg \text{trace}(tt') \in \text{tock}^*, \text{if-then-else} \} \\
&\quad \quad \quad a \notin \text{refusals}(tt') \\
&\quad \quad \quad \land \text{trace}(tt') \in \text{tock}^* \land \text{trace}(tt') \\
&\quad \quad \quad \text{a} = \text{head}(\text{trace}(\text{idlesuffix}(tt'))) \land \\
&\quad \quad \quad \text{a} \notin \text{refusals}(\text{idleprefix}(tt')) \land \\
&\quad \quad \quad \text{P}[\text{tail}(\text{idlesuffix}(tt'))/tt'] \\
&\quad = \{ \text{definition prefix} \} \\
&\quad \quad \text{a} \to P \\
&\end{align*}
\]

3. **Internal Choice**

\[
\text{T5}(P \land Q)
\]
Definition 2 (Public) = \{ \text{definition internal choice} \}

\[ T_5(P \lor Q) = \{ \text{\textit{T5-disjunction}} \} \]

\[ T_5(P) \lor T_5(Q) = \{ T_5(P); T_5(Q) \} \]

\[ P \lor Q = \{ \text{definition internal choice} \} \]

4. External Choice

\[ T_5(P \sqcap Q) = \{ \text{definition of external choice} \} \]

\[ T_5((P \sqcap Q)[t'/tt'] \land (P \lor Q)) = \{ \text{\textit{T5-conjunction}, \textit{T5-disjunction}} \} \]

\[ T_5(P)[t'/tt'] \land T_5(Q)[t'/tt'] \land T_5(P) \lor T_5(Q) = \{ T_5(P); T_5(Q) \} \]

\[ (P \land Q)[t'/tt'] \land (P \lor Q) = \{ \text{definition external choice} \} \]

\[ P \sqcap Q \]

5. Parallel Composition

\[ T_5(P \parallel_A Q) = \{ \text{definition parallel} \} \]

\[ T_5(\exists t, u \bullet P[t/\text{tt}] \land Q[u/\text{tt}] \land t \parallel_A u) = \{ \text{definition \textit{T5}} \} \]

\[ \forall k. \exists n.(\exists t, u \bullet P[t/\text{tt}] \land Q[u/\text{tt}] \land t \parallel_A u) \]

\[ \Rightarrow \#(t's \uparrow \text{tock}) \leq k \Rightarrow \#(\text{trace}(t')) \leq n \]

\[ = \{ \text{change of scope} \} \]

\[ \exists t, u \bullet P[t/\text{tt}] \land Q[u/\text{tt}] \land t' \parallel_A u \]

\[ \forall k. \exists n \bullet P(t) \land Q(u) \Rightarrow \]

\[ \#(t's \uparrow \text{tock}) \leq k \Rightarrow \#(\text{trace}(t')) \leq n \]

\[ = \{ \text{change of variable} k = k_t, n = n_t + n_u \} \]

\[ \exists t, u \bullet P[t/\text{tt}] \land Q[u/\text{tt}] \land t' \parallel_A u \]

\[ \forall k_t. \exists n_t \bullet \exists n_u \bullet P(t) \land Q(u) \Rightarrow \]

\[ \#(t's \uparrow \text{tock}) \leq k_t \Rightarrow \#(\text{trace}(t')) \leq n_t + n_u \]

\[ = \{ t' = t \parallel_A u \text{ (replace } t, u \text{ with contributing prefixes for } tt') \} \]

\[ \exists t, u \bullet P[t/\text{tt}] \land Q[u/\text{tt}] \land t' \parallel_A u \]

\[ \forall k_t. \exists n_t \bullet \exists n_u \bullet P(t) \land Q(u) \Rightarrow \]
Definition 2 (Public)

- **trace properties**

\[
\text{\#}(t \parallel_A u) \uparrow \text{tock} \leq k_t \Rightarrow \text{\#}(\text{trace}(t \parallel_A u)) \leq n_t + n_u
\]

\[
= \{ \text{calculation} \}
\]

\[
\exists t, u \bullet P[t \uparrow tt'] \land Q[u \uparrow tt'] \land tt' \in t \parallel_A u \land
\]

\[
\forall k_t \bullet \exists n_t \bullet \forall k_u \bullet \exists n_u \bullet P(t) \land Q(u) \Rightarrow
\]

\[
\text{\#}(t \uparrow \text{tock}) \leq k_t \land \text{\#}(u \uparrow \text{tock}) \leq k_u \Rightarrow \text{\#}(\text{trace}(t)) \leq n_t \land \text{\#}(\text{trace}(u)) \leq n_u
\]

\[
\cap
\]

\[
\exists t, u \bullet P[t \uparrow tt'] \land Q[u \uparrow tt'] \land tt' \in t \parallel_A u \land
\]

\[
\forall k_t \bullet \exists n_t \bullet P(t) \Rightarrow \text{\#}(t \uparrow \text{tock}) \leq k_t \Rightarrow \text{\#}(\text{trace}(t)) \leq n_t
\]

\[
\land
\]

\[
\forall k_u \bullet \exists n_u \bullet Q(u) \Rightarrow \text{\#}(u \uparrow \text{tock}) \leq k_u \Rightarrow \text{\#}(\text{trace}(u)) \leq n_u
\]

\[
\{ \text{T5}(Q(u)) \}
\]

\[
\exists t, u \bullet P[t \uparrow tt'] \land Q[u \uparrow tt'] \land tt' \in t \parallel_A u \land
\]

\[
\forall k_t \bullet \exists n_t \bullet P(t) \Rightarrow \text{\#}(t \uparrow \text{tock}) \leq k_t \Rightarrow \text{\#}(\text{trace}(t)) \leq n_t
\]

\[
\{ \text{T5}(P(t)) \}
\]

\[
\exists t, u \bullet P[t \uparrow tt'] \land Q[u \uparrow tt'] \land tt' \in t \parallel_A u
\]

\[
P \parallel_A Q
\]

6. **Hiding**

\[
\text{T5}(P \setminus A)
\]

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

\[
\text{T5}(\exists t. P[t \uparrow tt'] \land A \text{ urgent } t \land tt' = t \setminus A)
\]

\[
= \{ \text{T5} \}
\]

\[
\forall k. \exists n. \forall tt'. \exists t.(P[t \uparrow tt'] \land A \text{ urgent } t \land tt' = t \setminus A)
\]

\[
\Rightarrow \text{\#}(tt' \uparrow \text{tock}) \leq k \Rightarrow \text{\#}(\text{trace}(tt')) \leq n
\]

\[
= \{ \text{calculation} \}
\]

\[
\forall k. \exists n. \forall tt'. \exists t.(P[t \uparrow tt'] \Rightarrow \text{\#}(tt' \uparrow \text{tock}) \leq k \Rightarrow \text{\#}(\text{trace}(tt')) \leq n) \land
\]

\[
A \text{ urgent } t \land tt' = t \setminus A
\]

\[
\leftarrow \{ \text{forall, existential commute} \}
\]

\[
\exists t. \forall k. \exists n. \forall tt'. (P[t \uparrow tt'] \Rightarrow \text{\#}(tt' \uparrow \text{tock}) \leq k \Rightarrow \text{\#}(\text{trace}(tt')) \leq n) \land
\]

\[
A \text{ urgent } t \land tt' = t \setminus A
\]

\[
= \{ \text{definition T5} \}
\]

\[
\exists t. \text{T5}(P[t \uparrow tt']) \land A \text{ urgent } t \land tt' = t \setminus A
\]

\[
= \{ \text{T5}(P) \}
\]

\[
\exists t. P[t \uparrow tt'] \land A \text{ urgent } t \land tt' = t \setminus A
\]

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

\[
P \setminus A
\]

7. **Timeout**

123
(a) Case \( \neg \text{tock}^n \leq \text{trace}(tt') \)

\[
T_5(P \xrightarrow{\pi} Q) = \{ \text{definition timeout} \}
\]

\[
T_5 \left( \exists u \bullet u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[\text{u/}tt'] \land Q[tt' - u/\text{tt'}] \right)
\]

\[
\neg \text{tock}^n \leq \text{trace}(tt') \supset \hspace{1cm} P
\]

\[
T_5(\neg \text{tock}^n \leq \text{trace}(tt') \land T_5(P)) = \{ T_5 \text{ conjunction} \}
\]

\[
\neg \text{tock}^n \leq \text{trace}(tt') \land T_5(P)
\]

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

\[
T_5(P) \land \neg \text{tock}^n \leq \text{trace}(tt')
\]

\[
= \{ T_5(P) \}
\]

\[
P \land \neg \text{tock}^n \leq \text{trace}(tt')
\]

\[
= \{ \text{case} \neg \text{tock}^n \leq \text{trace}(tt') \}
\]

\[
\exists u \bullet u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[\text{u/}tt'] \land Q[tt' - u/\text{tt'}]
\]

\[
\neg \text{tock}^n \leq \text{trace}(tt') \supset \hspace{1cm} P
\]

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

\[
P \xrightarrow{\pi} Q
\]

(b) Case \( \text{tock}^n \leq \text{trace}(tt') \).

\[
T_5(P \xrightarrow{\pi} Q) = \{ \text{definition timeout} \}
\]

\[
T_5 \left( \exists u \bullet u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[\text{u/}tt'] \land Q[tt' - u/\text{tt'}] \right)
\]

\[
\neg \text{tock}^n \leq \text{trace}(tt') \supset \hspace{1cm} P
\]

\[
T_5(\exists u \bullet u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[\text{u/}tt'] \land Q[tt' - u/\text{tt'}])
\]

\[
= \{ T_5 \text{ definition} \}
\]

\[
\forall k \bullet \exists q \bullet (\exists u \bullet u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[\text{u/}tt'] \land Q[tt' - u/\text{tt'}])
\]

\[
\Rightarrow \#(\text{tt'} \uparrow \text{tock}) \leq k \Rightarrow \#(\text{trace}(\text{tt'})) \leq q
\]

\[
= \{ \text{change of variable} q = m + p \}
\]

\[
\forall k \bullet \exists p \bullet \exists m \bullet (\exists u \bullet u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[\text{u/}tt'] \land Q[tt' - u/\text{tt'}])
\]

\[
\Rightarrow \#(\text{tt'} \uparrow \text{tock}) \leq k \Rightarrow \#(\text{trace}(\text{tt'})) \leq m + p
\]

\[
= \{ \text{change of variable} k = j + l \}
\]

\[
\forall l \bullet \exists p \bullet \forall j \bullet \exists m \bullet (\exists u \bullet u \leq tt' \land (\text{trace}(u) = \text{tock}^n) \land P[\text{u/}tt'] \land Q[tt' - u/\text{tt'}])
\]

\[
\Rightarrow \#(\text{tt'} \uparrow \text{tock}) \leq j + l \Rightarrow \#(\text{trace}(\text{tt'})) \leq m + p
\]

\[
\iff \{ \text{trace properties} \}
\]
∀ l ⋅ ∃ p ⋅ ∀ j ⋅ ∃ m ⋅ (∃ u ⋅ u ≤ tt' ∧ (trace(u) = tockⁿ) ∧ P[u/ut'] ∧ Q[tt' − u/ut'])
⇒ (#(u ↑ tock) ≤ j ∧ #(tt' − u ↑ tock) ≤ l) ⇒
#(trace(u)) ≤ m ∧ #(trace(tt' − u)) ≤ p
= { calculus }

∀ l ⋅ ∃ p ⋅ ∀ j ⋅ ∃ m ⋅ (∃ u ⋅ u ≤ tt' ∧ (trace(u) = tockⁿ) ∧ P[u/ut'] ∧ Q[tt' − u/ut'])
⇒ #((u ↑ tock) ≤ j ⇒ #(trace(u)) ≤ m
∧
⇒ #((tt' − u ↑ tock) ≤ l ⇒ #(trace(tt' − u)) ≤ p
= { T5(Q[tt' − u/ut']) }  

∀ j ⋅ ∃ m ⋅ (∃ u ⋅ u ≤ tt' ∧ (trace(u) = tockⁿ) ∧ P[u/ut'] ∧ Q[tt' − u/ut'])
⇒ #((u ↑ tock) ≤ j ⇒ #(trace(u)) ≤ m
= { T5(P[u/ut']) }  

∃ u ⋅ u ≤ tt' ∧ (trace(u) = tockⁿ) ∧ P[u/ut'] ∧ Q[tt' − u/ut']
= { tockⁿ ≤ trace(tt') }

∃ u ⋅ u ≤ tt' ∧ (trace(u) = tockⁿ) ∧ P[u/ut'] ∧ Q[tt' − u/ut']
≤ tockⁿ ≤ trace(tt') ⊢
P
= { definition timeout }
P ⊳ Q

8. Recursion

Observe T5 is idempotent and disjunctive.

W.T.P μ F = T5(μ F), where μ F = ∩{X | X = F(X)}, given F(X) = T5(F(X)).

First, note

Y = F(Y)
= { F(X) = T5(F(X)) }  
Y = T5(F(Y))
⇒ { T5 functional }

T5(Y) = T5(T5(F(Y)))
= { idempotence of T5 }  

T5(Y) = T5(F(Y))
= { F(X) = T5(F(X)) }  

T5(Y) = F(Y)

(a) Case 1. {X | X = F(X)} ⊆ {T5(X) | X = F(X)}

{X | X = F(X)} ⊆ {T5(X) | X = F(X)}
= { comprehension }
\[ X = F(X) \implies \exists Y \cdot X = T_5(Y) \land Y = F(Y) \]
\[ = \{ F(X) = T_5(F(X)) \} \]
\[ X = F(X) \implies \exists Y \cdot X = F(Y) \land Y = F(Y) \]
\[ = \{ \text{liebniz} \} \]
\[ X = F(X) \implies \exists Y \cdot X = Y \land Y = F(Y) \]
\[ = \{ \text{one point rule} \} \]
\[ X = F(X) \implies X = F(X) \]
\[ = \{ \text{propositional calculus} \} \]
\[ \text{true} \]

(b) Case 2. \( \{ T_5(X) \mid X = F(X) \} \subseteq \{ X \mid X = F(X) \} \)
\[ \forall Y \cdot X = T_5(Y) \land Y = F(Y) \implies X = F(X) \]
\[ = \{ \text{liebniz} \} \]
\[ \forall Y \cdot X = T_5(F(Y)) \land Y = F(Y) \implies X = F(X) \]
\[ \iff \{ \text{one-point rule} \} \]
\[ T_5(F(Y)) = F(T_5(F(Y)) \]
\[ = \{ F(X) = T_5(F(X)) \} \]
\[ F(Y) = F(T_5(F(Y)) \]
\[ \iff \{ F(X) = T_5(F(X)) \} \]
\[ F(Y) = F(Y) \]
\[ = \{ \text{equality} \} \]
\[ \text{true} \]
Appendix B

Proof of parallel precedence
Lemma B.0.1 (parallel-precedence)

\[ tt' \in s_1 \parallel_A u_1 \land t \preceq tt' \Rightarrow \exists s, u \bullet s \preceq s_1 \land u \preceq u_1 \land t \in s \parallel_A u \]

Assume

\[ t \preceq tt' \land tt' \in p \parallel_A q \Rightarrow \exists s, u \bullet t \in s \parallel_A u \land s \preceq p \land u \preceq q \]

and

\[ a, b \in A; c, d \notin A \]

Proof is by induction on \( s_1 \) and \( u_1 \).

We begin with the cases where \( tt' \) must be \( \{ \} \).

1. case: \( s_1 = \{ \} \land u_1 = \{ \} \)

\[ \exists s, u \bullet t \in s \parallel_A u \land s \preceq s_1 \land u \preceq u_1 \]

\[ = \{ \text{case: } s_1 = \{ \} \} \]

\[ \exists s, u \bullet t \in s \parallel_A u \land s \preceq \{ \} \land u \preceq u_1 \]

\[ = \{ \text{case: } u_1 = \{ \} \} \]

\[ \exists s, u \bullet t \in s \parallel_A u \land s \preceq \{ \} \land u \preceq \{ \} \]

\[ = \{ \text{precedence: } (w \preceq \{ \}) = (w = \{ \}) \} \]

\[ \exists s, u \bullet t \in s \parallel_A u \land s = \{ \} \land u = \{ \} \]

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

\[ t \in \{ \} \parallel_A \{ \} \]

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

\[ t \in \{ \{ \} \} \]

\[ = \{ t \in \{ e \} = (t = e) \} \]

\[ t = \{ \} \]

\[ = \{ \text{conjunction} \} \]

\[ t = \{ \} \land tt' = \{ \} \]

\[ = \{ \text{precedence: } \{ \} \preceq \{ \} \} \]

\[ t \preceq tt' \land tt' = \{ \} \]

\[ = \{ (v = w) = (v \in \{ w \}) \} \]

\[ t \preceq tt' \land tt' \in \{ \{ \} \} \]

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

\[ t \preceq tt' \land tt' \in \{ \} \parallel_A \{ \} \]

\[ = \{ \text{case } s_1 = \{ \} \} \]

\[ t \preceq tt' \land tt' \in s_1 \parallel_A \{ \} \]

\[ = \{ \text{case } u_1 = \{ \} \} \]

\[ t \preceq tt' \land tt' \in s_1 \parallel_A u_1 \]
2. case: \( s_1 = \langle \rangle \land u_1 = \langle b \rangle \setminus y \)

\[ \exists s, u \cdot t \in s \parallel_A u \land s \preceq s_1 \land u \preceq u_1 \]
\[ \iff \{ \text{false} \Rightarrow P \} \]

false
\[ = \{ \text{set theory} \} \]
\[ t \preceq tt' \land tt' \in \{ \} \]
\[ = \{ \text{definition} \parallel_A, b \in A \} \]
\[ t \preceq tt' \land tt' \in \langle \rangle \parallel_A \langle b \rangle \setminus y \]
\[ = \{ \text{case } s_1 = \langle \rangle \} \]
\[ t \preceq tt' \land tt' \in s_1 \parallel_A \langle b \rangle \setminus y \]
\[ = \{ \text{case } u_1 = \langle b \rangle \setminus y \} \]
\[ t \preceq tt' \land tt' \in s_1 \parallel_A u_1 \]

3. case: \( s_1 = \langle \rangle \land u_1 = \langle d \rangle \setminus y \)

We make the assumption here that \( u_1 \neq \langle \rangle \). Otherwise \( s_1 = u_1 = \langle \rangle \) and the case reduces to case 1.

\[ \exists s, u \cdot s \preceq s_1 \land u \preceq u_1 \land t \in s \parallel_A u \]
\[ = \{ \text{case: } s_1 = \langle \rangle \} \]
\[ \exists s, u \cdot s \preceq \langle \rangle \land u \preceq u_1 \land t \in s \parallel_A u \]
\[ = \{ \text{case: } u_1 = \langle d \rangle \setminus y \} \]
\[ \exists s, u \cdot s \preceq \langle \rangle \land u \preceq \langle d \rangle \setminus y \land t \in s \parallel_A u \]
\[ = \{ \exists \text{-introduction: } (u \neq \langle \rangle) \} \]
\[ \exists s, u, v_2 \cdot v_2 = u - \langle d \rangle \land t \in s \parallel_A u \land s \preceq \langle \rangle \land u \preceq \langle d \rangle \setminus y \]
\[ = \{ \text{sequence difference} \} \]
\[ \exists s, u, v_2 \cdot u = \langle d \rangle \setminus v_2 \land t \in s \parallel_A u \land s \preceq \langle \rangle \land u \preceq \langle d \rangle \setminus y \]
\[ = \{ \text{one point rule} \} \]
\[ \exists s, v_2 \cdot t \in s \parallel_A \langle d \rangle \setminus v_2 \land s \preceq \langle \rangle \land t \setminus d \setminus y \]
\[ \iff \{ \text{precedence: } s = \langle \rangle \Rightarrow s \preceq \langle \rangle \} \]
\[ \exists s, v_2 \cdot t \in s \parallel_A \langle d \rangle \setminus v_2 \land s = \langle \rangle \land t \setminus d \setminus y \]
\[ = \{ \text{one point rule} \} \]
\[ \exists v_2 \cdot t \in \langle \rangle \parallel_A \langle d \rangle \setminus v_2 \land t \setminus d \setminus y \]
\[ = \{ \text{definition of } \parallel_A \} \]
\[ \exists v_2 \cdot t \in \{ \langle d \rangle \setminus t_2 \mid t_2 \in \langle \rangle \parallel_A u_2 \} \land t \setminus d \setminus y \]
\[ = \{ \text{axiom of comprehension} \} \]
\[ t_2, v_2 \cdot t = \langle d \rangle \setminus t_2 \land t_2 \in \langle \rangle \parallel_A u_2 \land \langle d \rangle \setminus u_2 \preceq \langle d \rangle \setminus y \]
\[ = \{ \text{sequence difference} \} \]
\[ t_2, v_2 \cdot t = t - \langle d \rangle \land t_2 \in \langle \rangle \parallel_A u_2 \land \langle d \rangle \setminus u_2 \preceq \langle d \rangle \setminus y \]
\[ = \{ \text{one point rule} \} \]
\[ \exists u_2 \bullet t \in \langle d \rangle \parallel_A u_2 \land \langle d \rangle \models u_2 \preceq \langle d \rangle \models y \]

\[ = \{ \text{case: } s_2 = \langle \rangle \} \]

\[ \exists u_2 \bullet t \in \langle d \rangle \parallel_A u_2 \land \langle d \rangle \models u_2 \preceq \langle d \rangle \models y \]

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

\[ \exists s_2, u_2 \bullet t \in \langle d \rangle \models s_2 \parallel_A u_2 \land \langle d \rangle \models u_2 \preceq \langle d \rangle \models y \land s_2 = \langle \rangle \]

\[ \{ \langle \rangle \preceq \langle \rangle = (\langle \rangle = \langle \rangle) \} \]

\[ \exists s_2, u_2 \bullet t \in \langle d \rangle \parallel_A u_2 \land \langle d \rangle \models u_2 \preceq \langle d \rangle \models y \land s_2 \preceq \langle \rangle \]

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

\[ \exists s_2, u_2 \bullet t \in \langle d \rangle \models s_2 \parallel_A u_2 \land u_2 \preceq y \land s_2 \preceq \langle \rangle \]

\[ \iff \{ \text{Induction assumption} \} \]

\[ t \preceq \langle d \rangle \preceq v \land v \in \langle \rangle \parallel_A y \land \langle \rangle \preceq \langle \rangle \]

\[ = \{ \langle \rangle \preceq \langle \rangle \} \]

\[ t \preceq \langle d \rangle \preceq v \land v \in \langle \rangle \parallel_A y \]

\[ = \{ \text{sequence prefix} \} \]

\[ t \preceq \langle d \rangle \models v \land v \in \langle \rangle \parallel_A y \]

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

\[ t \preceq tt' \land tt'' = \langle d \rangle \models v \land v \in \langle \rangle \parallel_A y \]

\[ = \{ \text{axiom of comprehension} \} \]

\[ t \preceq tt' \land tt'' \in \{ \langle d \rangle \models v \mid v \in \langle \rangle \parallel_A y \} \]

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

\[ t \preceq tt' \land tt'' \in \langle \rangle \parallel_A \langle d \rangle \models y \]

\[ = \{ \text{case: } u_1 = \langle d \rangle \models y \} \]

\[ t \preceq tt' \land tt'' \in \langle \rangle \parallel_A u_1 \]

\[ = \{ \text{case: } s_1 = \langle \rangle \} \]

\[ t \preceq tt' \land tt'' \in s_1 \parallel_A u_1 \]

\[ \text{4. case: } s_1 = \langle \rangle \land u_1 = \langle T, \text{tock} \rangle \models y \]

\[ \exists s, u \bullet t \in s \parallel_A u \land s \preceq s_1 \land u \preceq u_1 \]

\[ \iff \{ \text{false } \Rightarrow P \} \]

\[ \text{false} \]

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

\[ t \preceq tt' \land tt'' \in \{ \} \]

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

\[ t \preceq tt' \land tt'' \in \langle \rangle \parallel_A \langle T, \text{tock} \rangle \models y \]

\[ = \{ \text{case } s_1 = \langle \rangle \} \]

\[ t \preceq tt' \land tt'' \in s_1 \parallel_A \langle T, \text{tock} \rangle \models y \]

\[ = \{ \text{case } u_1 = \langle T, \text{tock} \rangle \models y \} \]

\[ t \preceq tt' \land tt'' \in s_1 \parallel_A u_1 \]
5. case: \( s_1 = \langle a \rangle \upharpoonright x \land u_1 = \langle a \rangle \upharpoonright y \land a \in A \)

We make the assumptions here that \( u, s \) and \( t \) are not empty. Otherwise \( s = u = t = \langle \rangle \) and the case reduces to case 1.

\[
\exists s, u \cdot s \leq s_1 \land u \leq u_1 \land t \in s \|_A u
\]
\[
\exists s, u \cdot s \leq \langle a \rangle \upharpoonright x \land u \leq u_1 \land t \in s \|_A u
\]
\[
\exists s, u \cdot s \leq \langle a \rangle \upharpoonright x \land u \leq \langle a \rangle \upharpoonright y \land t \in s \|_A u
\]
\[
\exists s, u \cdot s \leq \langle a \rangle \upharpoonright x \land u \leq \langle a \rangle \upharpoonright y
\]
\[
\exists s, u, s_2, u_2 \cdot s = s - \langle a \rangle \land u_2 = u - \langle a \rangle \land t \in s \|_A u
\]
\[
\exists s, u, s_2, u_2 \cdot s = s - \langle a \rangle \land u_2 = u - \langle a \rangle \land t \in s \|_A u
\]
\[
\exists s, u, s_2, u_2 \cdot s = s - \langle a \rangle \land u_2 = u - \langle a \rangle \land t \in s \|_A u
\]
\[
\exists s, u, s_2, u_2 \cdot s = s - \langle a \rangle \land u_2 = u - \langle a \rangle \land t \in s \|_A u
\]
\[ t \leq tt' \wedge tt' \in \{ (a) \.bytes v \mid v \in x \parallel_A y \} \]
\[ = \{ \text{definition of } \parallel_A \} \]
\[ t \leq tt' \wedge tt' \in \{ (a) \.bytes x \parallel_A (a) \.bytes y \} \]
\[ = \{ \text{case: } u_1 = (a) \.bytes y \} \]
\[ t \leq tt' \wedge tt' \in \{ (a) \bytes x \parallel_A u_1 \} \]
\[ = \{ \text{case: } s_1 = (a) \bytes x \} \]
\[ t \leq tt' \wedge tt' \in s_1 \parallel_A u_1 \]

6. case: \( s_1 = (a) \.bytes x \wedge u_1 = (b) \bytes y \wedge a, b \in A \) for some \( x \) and \( y \).

\[ \exists s, u \bullet t \in s \parallel_A u \wedge s \leq s_1 \wedge u \leq u_1 \]
\[ \iff \{ \text{false } \Rightarrow P \} \]
\[ \text{false} \]
\[ = \{ \text{set theory} \} \]
\[ t \leq tt' \wedge tt' \in \{ \} \]
\[ = \{ \text{definition } \parallel_A \} \]
\[ t \leq tt' \wedge tt' \in (a) \bytes x \parallel_A (b) \bytes y \]
\[ = \{ \text{case } s_1 = (a) \bytes x \} \]
\[ t \leq tt' \wedge tt' \in s_1 \parallel_A (b) \bytes y \]
\[ = \{ \text{case } u_1 = (b) \bytes y \} \]
\[ t \leq tt' \wedge tt' \in s_1 \parallel_A u_1 \]

7. case: \( s_1 = (a) \bytes x \wedge u_1 = (d) \bytes y \).

We make the assumptions here that \( u, s \) and \( t \) are not empty. Otherwise \( s = u = t = \langle \rangle \) and the case reduces to case 1.

\[ \exists s, u \bullet s \leq s_1 \wedge u \leq u_1 \wedge t \in s \parallel_A u \]
\[ = \{ \text{case: } s_1 = (a) \bytes x \} \]
\[ \exists s, u \bullet s \leq (a) \bytes x \wedge u \leq u_1 \wedge t \in s \parallel_A u \]
\[ = \{ \text{case: } u_1 = (d) \bytes y \} \]
\[ \exists s, u \bullet s \leq (a) \bytes x \wedge u \leq (d) \bytes y \wedge t \in s \parallel_A u \]
\[ = \{ \exists \text{-introduction: } (s \neq \langle \rangle, u \neq \langle \rangle) \} \]
\[ \exists s, u, s_2, u_2 \bullet s = s - (a) \wedge u = u - (d) \wedge t \in s \parallel_A u \]
\[ \wedge s \leq (a) \bytes x \wedge u \leq (d) \bytes y \]
\[ = \{ \text{sequence difference} \} \]
\[ \exists s, u, s_2, u_2 \bullet s = (a) \bytes s_2 \wedge u = (d) \bytes u_2 \wedge t \in s \parallel_A u \]
\[ \wedge s \leq (a) \bytes x \wedge u \leq (d) \bytes y \]
\[ = \{ \text{one point rule} \} \]
\[ \exists s_2, u_2 \bullet t \in (a) \bytes s_2 \parallel_A (d) \bytes u_2 \wedge (a) \bytes s_2 \leq (a) \bytes x \wedge (a) \bytes u_2 \leq (d) \bytes y \]
\[ = \{ \text{definition of } \parallel_A \} \]

132
\exists s_2, u_2 \cdot t \in \{ (d) \triangleleft t_2 \mid t_2 \in (a) \triangleleft s_2 \parallel_A u_2 \}
\wedge (a) \triangleleft s_2 \leq (a) \triangleleft x \wedge (d) \triangleleft u_2 \leq (d) \triangleleft y
= \{ \text{axiom of comprehension} \}
\exists s_2, u_2, t_2 \cdot t = (d) \triangleleft t_2 \triangleleft t_2 \in (a) \triangleleft s_2 \parallel_A u_2
\wedge (a) \triangleleft s_2 \leq (a) \triangleleft x \wedge (d) \triangleleft u_2 \leq (d) \triangleleft y
= \{ \text{sequence difference} \}
\exists s_2, u_2, t_2 \cdot t = t - (d) \triangleleft t_2 \triangleleft t_2 \in (a) \triangleleft s_2 \parallel_A u_2 \wedge
(a) \triangleleft s_2 \leq (a) \triangleleft x \wedge (d) \triangleleft u_2 \leq (d) \triangleleft y
= \{ \text{one point rule} \}
\exists s_2, u_2 \cdot t - (d) \in (a) \triangleleft s_2 \parallel_A u_2 \wedge (a) \triangleleft s_2 \leq (a) \triangleleft x \wedge (d) \triangleleft u_2 \leq (d) \triangleleft y
= \{ \text{definition of } \leq \}
\exists s_2, u_2 \cdot t - (d) \in (a) \triangleleft s_2 \parallel_A u_2 \wedge (a) \triangleleft s_2 \leq (a) \triangleleft x \wedge u_2 \leq y
\iff \{ \text{Induction hypothesis} \}
t - (d) \leq v \wedge v \in (a) \parallel_A y
= \{ \text{sequence prefix } t \neq (\cdot) \}
t \leq (d) \wedge v \wedge v \in (a) \parallel_A y
= \{ \text{one point rule} \}
t \leq tt' \wedge tt' = (d) \wedge v \wedge v \in (a) \wedge y
= \{ \text{axiom of comprehension} \}
t \leq tt' \wedge tt' \in \{ (d) \wedge v \mid v \in (a) \parallel_A y \}
= \{ \text{definition of } \parallel_A \}
t \leq tt' \wedge tt' \in (a) \wedge (d) \wedge y
= \{ \text{case: } u_2 = (d) \parallel_A y \}
t \leq tt' \wedge tt' \in (a) \wedge u_1
= \{ \text{case: } s_1 = (a) \parallel_A x \}
t \leq tt' \wedge tt' \in s_1 \parallel_A u_1

8. case: \ s_1 = (a) \triangleleft x \wedge u_1 = (T, tock) \wedge y \wedge a \in A \text{ for some } x \text{ and } y.
\exists s, u \cdot t \in s \parallel_A u \wedge s \leq s_1 \wedge u \leq u_1
\iff \{ \text{false } \Rightarrow P \}

false
= \{ \text{set theory} \}
t \leq tt' \wedge tt' \in \{ \}
= \{ \text{definition } \parallel_A \}
t \leq tt' \wedge tt' \in (a) \wedge \{ (T, tock) \wedge y
= \{ \text{case } s_1 = (a) \parallel_A x \}
t \leq tt' \wedge tt' \in s_1 \parallel_A (T, tock) \wedge y
= \{ \text{case } u_2 = (T, tock) \wedge y \}
t \leq tt' \wedge tt' \in s_1 \parallel_A u_1
133
9. case: \( s_1 = \langle c \rangle \cap x \land u_1 = \langle d \rangle \cap y \).

We make the assumptions here that \( u \), \( s \) and \( t \) are not empty. Otherwise \( s = u = t = \langle \rangle \) and the case reduces to case 1.

\[
\exists s, u \cdot s \subseteq s_1 \land u \subseteq u_1 \land t \in s \|_A u
\]
\[
\exists s, u \cdot s \subseteq \langle c \rangle \cap x \land u \subseteq u_1 \land t \in s \|_A u
\]
\[
\exists s, u \cdot s \subseteq \langle c \rangle \cap x \land u \subseteq \langle d \rangle \cap y \land t \in s \|_A u
\]

= \{ \text{\textit{case:} } s_1 = \langle c \rangle \cap x \} \\
= \{ \text{\textit{case:} } u_1 = \langle d \rangle \cap y \} \\
= \{ \exists \text{-introduction} \} \\
= \{ \text{\textit{sequence difference}} \} \\
= \{ \text{\textit{one point rule}} \} \\
= \{ \text{\textit{definition of} } \|_A \} \\
= \{ \text{\textit{sequence difference twice}} \} \\
= \{ \text{\textit{one point rule twice}} \} \\
= \{ \text{\textit{definition of} } \leq \} \\
= \{ \text{\textit{Induction hypothesis twice}} \} \\
= \{ \text{\textit{sequence prefix} (} t \neq \langle \rangle \} \} \\

134
\[ t \preceq (d) \land v \land v \in (d) \land x \parallel_A y \lor \\
  t \preceq (c) \land v \land v \in x \parallel_A (c) \land y \]

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

\[ t \preceq tt' \land tt' = (d) \land v \land v \in (c) \land x \parallel_A y \lor \\
  t \preceq tt' \land tt' = (c) \land v \land v \in x \parallel_A (d) \land y \]

= \{ \text{axiom of comprehension} \}

\[ t \preceq tt' \land tt' \in \{(d) \land v \mid v \in (c) \land x \parallel_A y\} \cup \{(c) \land v \mid v \in x \parallel_A (d) \land y\} \]

= \{ \text{definition of } \parallel_A \}

\[ t \preceq tt' \land tt' \in \langle c \rangle \land x \parallel_A (d) \land y \]

= \{ \text{case: } u_1 = (d) \land y \}

\[ t \preceq tt' \land tt' \in \langle c \rangle \land x \parallel_A u_1 \]

= \{ \text{case: } s_1 = (c) \land x \}

\[ t \preceq tt' \land tt' \in s_1 \parallel_A u_1 \]

10. case: \( s_1 = \langle c \rangle \land x \land u_1 = \langle T, tock \rangle \land y \). We make the assumptions here that \( u, s \) and \( t \) are not empty. Otherwise \( s = u = t = \{\} \) and the case reduces to case 1.

\[ \exists s, u \bullet s \preceq s_1 \land u \preceq u_1 \land t \in s \parallel_A u \]

= \{ \text{case: } s_1 = \langle c \rangle \land x \}

\[ \exists s, u \bullet s \preceq \langle c \rangle \land x \land u \preceq u_1 \land t \in s \parallel_A u \]

= \{ \text{case: } u_1 = \langle T, tock \rangle \land y \}

\[ \exists s, u \bullet s \preceq \langle c \rangle \land x \land u \preceq \langle T, tock \rangle \land y \land t \in s \parallel_A u \]

= \{ \exists\text{-introduction} \}

\[ \exists s, u, s_2, s_2 \bullet s_2 = s - \langle c \rangle \land u_2 = u - \langle T, tock \rangle \land t \in s \parallel_A u \land s \preceq \langle c \rangle \land x \land \\
  u \preceq \langle T, tock \rangle \land y \]

= \{ \text{sequence difference} \}

\[ \exists s, u, s_2, u_2 \bullet s = \langle c \rangle \land s_2 \land u = \langle T, tock \rangle \land u_2 \land t \in s \parallel_A u \land s \preceq \langle c \rangle \land x \land \\
  u \preceq \langle T, tock \rangle \land y \]

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

\[ \exists s_2, u_2 \bullet t \in \langle c \rangle \land s_2 \parallel_A \langle T, tock \rangle \land u_2 \land \langle c \rangle \land s_2 \preceq \langle c \rangle \land x \land \\
  \langle c \rangle \land u_2 \preceq \langle T, tock \rangle \land y \]

= \{ \text{definition of } \parallel_A \}

\[ \exists s_2, u_2 \bullet t \in \langle c \rangle \land t_2 \land t_2 \in s_2 \parallel_A \langle T, tock \rangle \land u_2 \land \langle c \rangle \land s_2 \preceq \langle c \rangle \land x \land \\
  \langle T, tock \rangle \land u_2 \preceq \langle T, tock \rangle \land y \]

= \{ \text{axiom of comprehension} \}

\[ \exists s_2, u_2, t_2 \bullet t = \langle c \rangle \land t_2 \land t_2 \in s_2 \parallel_A \langle T, tock \rangle \land u_2 \land \langle c \rangle \land s_2 \preceq \langle c \rangle \land x \land \\
  \langle T, tock \rangle \land u_2 \preceq \langle T, tock \rangle \land y \]

= \{ \text{sequence difference} \}

\[ \exists s_2, u_2, t_2 \bullet t = t - \langle c \rangle \land t_2 \land t_2 \in s_2 \parallel_A \langle T, tock \rangle \land u_2 \land \langle c \rangle \land s_2 \preceq \langle c \rangle \land x \land \\
  \langle T, tock \rangle \land u_2 \preceq \langle T, tock \rangle \land y \]
We make the assumptions here that \( u, s \) and \( t \) are not empty. Otherwise \( s = u = t = \langle \rangle \) and the case reduces to case 1.

11. \textit{case: } \( s_1 = \langle S, \text{tock} \rangle \backsimeq x \land u_1 = \langle T, \text{tock} \rangle \backsimeq y \).

\[
\exists s_2, u_2 \bullet t - \langle c \rangle \in s_2 \parallel_A \langle T, \text{tock} \rangle \backsimeq u_2 \land \langle c \rangle \backsimeq s_2 \leq \langle c \rangle \backsimeq x \land \\
\langle T, \text{tock} \rangle \backsimeq u_2 \leq \langle T, \text{tock} \rangle \backsimeq y
\]

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

\[
\exists s_2, u_2 \bullet t - \langle c \rangle \in s_2 \parallel_A \langle T, \text{tock} \rangle \backsimeq u_2 \land s_2 \leq x \land \\
\langle T, \text{tock} \rangle \backsimeq u_2 \leq \langle T, \text{tock} \rangle \backsimeq y
\]

\[
\leadsto \{ \text{Induction hypothesis} \}
\]

\[
t - \langle c \rangle \leq v \land v \in x \parallel_A \langle T, \text{tock} \rangle \backsimeq y
\]

\[
= \{ \text{sequence prefix } (t \neq \langle \rangle) \}
\]

\[
t \leq \langle c \rangle \land v \land v \in x \parallel_A \langle T, \text{tock} \rangle \backsimeq y
\]

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

\[
t \leq tt' \land tt' = \langle c \rangle \land v \land v \in x \parallel_A \langle T, \text{tock} \rangle \backsimeq y
\]

\[
= \{ \text{axiom of comprehension} \}
\]

\[
t \leq tt' \land tt' \in \{ \langle c \rangle \land v \mid v \in x \parallel_A \langle T, \text{tock} \rangle \backsimeq y \}
\]

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

\[
t \leq tt' \land tt' \in \{ \langle c \rangle \land x \parallel_A \langle T, \text{tock} \rangle \backsimeq y \}
\]

\[
= \{ \text{case: } u_1 = \langle T, \text{tock} \rangle \backsimeq y \}
\]

\[
t \leq tt' \land tt' \in \{ \langle c \rangle \land x \parallel_A u_1 \}
\]

\[
= \{ \text{case: } s_1 = \langle c \rangle \land x \}
\]

\[
t \leq tt' \land tt' \in s_1 \parallel_A u_1
\]
\( \exists s_2, u_2, t_2 \cdot t \in \{ \langle U, \text{tock} \rangle \cap t_2 \mid U \in S \cap A \cap_t \forall \in s_2 \parallel_A u_2 \} \land \langle S, \text{tock} \rangle \cap s_2 \leq \langle S, \text{tock} \rangle \cap x \land \langle T, \text{tock} \rangle \cap u_2 \leq \langle T, \text{tock} \rangle \cap y \)

\[ = \{ \text{axiom of comprehension} \} \]

\( \exists s_2, u_2, t_2 \cdot t = \langle U, \text{tock} \rangle \cap t_2 \land U \in S \cap A \cap_t \forall \in s_2 \parallel_A u_2 \land \langle S, \text{tock} \rangle \cap s_2 \leq \langle S, \text{tock} \rangle \cap x \land \langle T, \text{tock} \rangle \cap u_2 \leq \langle T, \text{tock} \rangle \cap y \)

\[ = \{ \text{sequence difference} \} \]

\( \exists s_2, u_2, t_2 \cdot t = t - \langle U, \text{tock} \rangle \land U \in S \cap A \cap_t \forall \in s_2 \parallel_A u_2 \land \langle S, \text{tock} \rangle \cap s_2 \leq \langle S, \text{tock} \rangle \cap x \land \langle T, \text{tock} \rangle \cap u_2 \leq \langle T, \text{tock} \rangle \cap y \)

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

\( \exists s_2, u_2 \cdot U \in S \cap A \cap_t \forall \in s_2 \parallel_A u_2 \land \langle S, \text{tock} \rangle \cap s_2 \leq \langle S, \text{tock} \rangle \cap x \land \langle T, \text{tock} \rangle \cap u_2 \leq \langle T, \text{tock} \rangle \cap y \)

\[ = \{ \text{definition of \( \leq \) twice} \} \]

\( \exists s_2, u_2 \cdot U \in S \cap A \cap_t \forall \in s_2 \parallel_A u_2 \land s_2 \leq x \land u_2 \leq y \)

\[ \iff \{ \text{Induction hypothesis} \} \]

\( U \in S \cap A \cap_t \forall \in s_2 \parallel_A u_2 \land \langle S, \text{tock} \rangle \cap v \leq x \land v \in x \parallel_A y \)

\[ = \{ \text{sequence prefix (assumption \( t \neq \langle \rangle \))} \} \]

\( U \in S \cap A \cap_t \forall \leq \langle U, \text{tock} \rangle \cap v \leq x \parallel_A y \)

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

\( U \in S \cap A \cap_t \forall \leq \langle U, \text{tock} \rangle \cap v \leq x \parallel_A y \)

\[ = \{ \text{axiom of comprehension} \} \]

\( t \leq tt' \land tt' = \langle U, \text{tock} \rangle \cap v \leq x \parallel_A y \)

\[ = \{ \text{definition of \( \parallel_A \)} \} \]

\( t \leq tt' \land tt' \in \{ \langle U, \text{tock} \rangle \cap v \mid U \in S \cap A \cap_t \forall \leq x \parallel_A y \} \)

\[ = \{ \text{definition of \( \parallel_A \)} \} \]

\( t \leq tt' \land tt' \in \{ \langle S, \text{tock} \rangle \cap x \parallel_A \langle T, \text{tock} \rangle \cap y \}

\[ = \{ \text{case: } u_1 = \langle T, \text{tock} \rangle \cap y \} \]

\( t \leq tt' \land tt' \in \{ \langle S, \text{tock} \rangle \cap x \parallel_A u_1 \}

\[ = \{ \text{case: } s_1 = \langle S, \text{tock} \rangle \cap x \} \]

\( t \leq tt' \land tt' \in s_1 \parallel_A u_1 \)
B.1 Proof of Lemma 4.3.1

Lemma B.1.1 \((\text{RT} \text{ functions are monotonic idempotents})\)

1. \(\text{RT}_0 - \text{RT}_5\) are all monotonic idempotents.

2. \(\text{RT}_0 - \text{RT}_5\) enjoy certain commutativity properties, detailed below.

Proof B.1.1 Monotonicity

1. \(\text{RT}_0\) is monotonic

\[
\begin{align*}
\text{RT}_0(P) &= \{ \text{definition of } \text{RT}_0 \} \\
T_0(P) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace} &\Rightarrow \{ a \land b \Rightarrow a \} \\
P
\end{align*}
\]

2. \(\text{RT}_1\) is monotonic. Assume \(P \sqsubseteq Q\)

\[
\begin{align*}
\text{RT}_1(P) &= \{ \text{RT}_1 \} \\
P \land (rt' = rt \bowtie tt') &\iff \{ P \sqsubseteq Q = Q \Rightarrow P \} \\
Q \land (rt' = rt \bowtie tt') &= \{ \text{RT}_1 \} \\
\text{RT}_1(Q)
\end{align*}
\]

3. \(\text{RT}_2\) is monotonic. Assume \(P \sqsubseteq Q\).

\[
\begin{align*}
\text{RT}_2(P) &= \{ \text{RT}_2 \} \\
P[\langle \rangle, tt'/rt, rt'] &\iff \{ P \sqsubseteq Q \} \\
Q[\langle \rangle, tt'/rt, rt'] &= \{ \text{RT}_2 \} \\
\text{RT}_2(Q)
\end{align*}
\]

4. \(\text{RT}_3\) is monotonic. Assume \(P \sqsubseteq Q\).

\[
\begin{align*}
\text{RT}_3(P) &= \{ \text{RT}_3 \} \\
\Pi_{\text{rt}} < \text{wait} \triangleright P &\iff \{ \text{conditional defn} \} \\
\text{wait} \land \Pi_{\text{rt}} \land \neg \text{wait} \land P &\iff \{ a \Rightarrow b = a \lor c \Rightarrow b \lor c \} \\
\text{wait} \land \Pi_{\text{rt}} \land \neg \text{wait} \land Q &= \{ \text{conditional defn} \} \\
\Pi_{\text{rt}} < \text{wait} \triangleright Q &= \{ \text{RT}_3 \} \\
\text{RT}_3(Q)
\end{align*}
\]
5. **RT4** is monotonic. Assume $P \sqsubseteq Q$.

\[
RT4(P) = \{ RT4 \}
\]
\[
RT0 \circ TTO(\neg ok) \lor P
\]
\[
\iff \{ a \Rightarrow b = a \lor c \Rightarrow b \lor c \}
\]
\[
RT0 \circ TTO(\neg ok) \lor Q
\]
\[
= \{ RT4 \}
\]
\[
RT4(Q)
\]

6. **RT5** is monotonic. Assume $P \sqsubseteq Q$.

\[
RT5(P)
\]
\[
= \{ RT5 \}
\]
\[
P ; J
\]
\[
\iff \{ Q ; J \Rightarrow P ; J \}
\]
\[
Q ; J
\]
\[
= \{ RT5 \}
\]
\[
RT5(Q)
\]

**Commutativity Properties**

1. **RT0(RT1(P)) = RT1(RT0(P))** 

\[
RT0(RT1(P))
\]
\[
= \{ RT1 \}
\]
\[
RT0(P \land rt' = rt \land tt')
\]
\[
= \{ RT0 \}
\]
\[
T0(P \land rt' = rt \land tt') \land rt \in timedTrace \land rt' \in timedTrace
\]
\[
= \{ T0 \}
\]
\[
(P \land rt' = rt \land tt') \land rt \in timedTrace \land rt' \in timedTrace \land tt' \in timedTrace
\]
\[
= \{ T0 \}
\]
\[
T0(P) \land rt \in timedTrace \land rt' \in timedTrace \land rt' = rt \land tt'
\]
\[
= \{ RT1 \}
\]
\[
RT1(T0(P) \land rt \in timedTrace \land rt' \in timedTrace)
\]
\[
= \{ RT0 \}
\]
\[
RT1(RT0(P))
\]

2. **RT0(RT2(P)) \Rightarrow RT2(RT0(P))**

\[
RT0(RT2(P))
\]
\[
= \{ RT2 \}
\]
\[
RT0(P(([]), tt'/rt, rt'))
\]
\[
= \{ RT0 \}
\]
\[
T0(P(([]), tt'/rt, rt')) \land rt \in timedTrace \land rt' \in timedTrace
\]
\[
\Rightarrow \{ \text{propositional calculus} \}
\]
\[
T0(P \land rt \in timedTrace \land rt' \in timedTrace)(([]), tt'/rt, rt')
\]
\[
= \{ RT2 \}
\]
\[
RT2(T0(P) \land rt \in timedTrace \land rt' \in timedTrace)
\]
\[
= \{ RT0 \}
\]
\[
RT2(RT0(P))
\]
3. $RT_0(RT_3(P)) = RT_3(RT_0(P))$

\[
RT_0(RT_3(P)) = \{RT_3\}
\]
\[
RT_0(\Pi_{RT} \triangleleft \text{wait} \triangleright P) = \{RT_0\}
\]
\[
T_0(\Pi_{RT} \triangleleft \text{wait} \triangleright P) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace} = \{T_0\}
\]
\[
\Pi_{RT} \triangleleft \text{wait} \triangleright (P \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace} \land tt' \in \text{timedTrace}) = \{\text{propositional calculus}\}
\]
\[
\Pi_{RT} \triangleleft \text{wait} \triangleright (T_0(P) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace}) = \{RT_3\}
\]
\[
RT_3(T_0(P)) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace} = \{RT_0\}
\]
\[
RT_3(RT_0(P)) = \{RT_0\}
\]

4. $RT_0(RT_4(P)) = RT_4(RT_0(P))$

\[
RT_0(RT_4(P)) = \{RT_4\}
\]
\[
RT_0(RT_01 \circ T_0(\neg ok) \lor P) = \{RT_0\}
\]
\[
T_0(RT_01 \circ T_0(\neg ok) \lor P) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace} = \{T_0\ \text{distributivity}\}
\]
\[
(RT_01 \circ T_0(\neg ok) \lor T_0(P)) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace} = \{\text{calculus}\}
\]
\[
RT_01 \circ T_0(\neg ok) \lor (T_0(P) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace}) = \{RT_4\}
\]
\[
RT_4(T_0(P) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace}) = \{RT_0\}
\]
\[
RT_4(RT_0(P)) = \{RT_0\}
\]

5. $RT_0(RT_5(P)) = RT_5(RT_0(P))$
\[ RT_0(\text{RT}_5(P)) \]
\[ = \{ \text{RT}_5 \} \]
\[ RT_0(P ; J) \]
\[ = \{ RT_0 \} \]
\[ T_0(P ; J) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace} \]
\[ = \{ \text{relational composition} \} \]
\[ \exists ok_0 \cdot T_0(P)[ok_0/ok'] \land J[ok_0/ok] \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace} \]
\[ = \{ \text{contract scope} \} \]
\[ \exists ok_0 \cdot (T_0(P) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace})[ok_0/ok'] \land J[ok_0/ok] \]
\[ = \{ \text{relational composition} \} \]
\[ (T_0(P) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace}) ; J \]
\[ = \{ \text{RT}_5 \} \]
\[ \text{RT}_5(T_0(P) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace}) \]
\[ = \{ \text{RT}_0 \} \]
\[ RT_5(\text{RT}_0(P)) \]

6. \[ \text{RT}_2(\text{RT}_1(P)) = \text{RT}_1(\text{RT}_2(P)) \]

\[ \text{RT}_1(\text{RT}_2(P)) \]
\[ = \{ \text{RT}_2 \} \]
\[ \text{RT}_1(P[\{\}, tt'/rt, rt']) \]
\[ = \{ \text{RT}_1 \} \]
\[ P[\{\}, tt'/rt, rt'] \land rt' = rt \land tt' \]
\[ = \{ \text{RT}_2 \} \]
\[ \text{RT}_2(P(rt, rt')) \land rt' = rt \land tt' \]
\[ = \{ \text{RT}_1 \} \]
\[ \text{RT}_1(\text{RT}_2(P)) \]

7. \[ \text{RT}_3(\text{RT}_1(P)) = \text{RT}_1(\text{RT}_3(P)) \]

\[ \text{RT}_1(\text{RT}_3(P)) \]
\[ = \{ \text{RT}_3 \} \]
\[ \text{RT}_1(\Pi_{rt} \land \text{wait} \lor \neg \text{wait} \land P) \]
\[ = \{ \text{definition conditional} \} \]
\[ \text{RT}_1(\Pi_{rt} \land \text{wait} \lor \neg \text{wait} \land P) \]
\[ = \{ \text{RT}_1 \} \]
\[ (\Pi_{rt} \land \text{wait} \lor \neg \text{wait} \land P) \land rt' = rt \land tt' \]
\[ = \{ \text{distribution} \} \]
\[ \Pi_{rt} \land \text{wait} \land rt' = rt \land tt' \lor \neg \text{wait} \land P \land rt' = rt \land tt' \]
\[ = \{ \Pi_{rt} \} \]
\[ \text{RT}(\text{true} \models \Pi) \land \text{wait} \land rt' = rt \land tt' \lor \neg \text{wait} \land P \land rt' = rt \land tt' \]
\[ = \{ \text{RT} \Rightarrow \text{RT}_1 \} \]
\[ \text{RT}(\text{true} \models \Pi) \land \text{wait} \land rt' = rt \land tt' \lor \neg \text{wait} \land P \land rt' = rt \land tt' \]
\[ = \{ \Pi_{rt} \} \]
\[ \Pi_{rt} \land \text{wait} \land rt' = rt \land tt' \lor \neg \text{wait} \land P \land rt' = rt \land tt' \]
\[ \text{RT}_3(P \land rt' = rt \land tt') \]
\[ = \{ \text{RT}_1 \} \]
\[ \text{RT}_3(\text{RT}_1(P)) \]
8. $\text{RT}_1(\text{RT}_4(P)) = \text{RT}_4(\text{RT}_1(P))$

\[
\begin{align*}
\text{RT}_1(\text{RT}_4(P)) & = \{ \text{RT}_4 \} \\
\text{RT}_1(\text{RT}_0 \circ T_0(\neg \text{ok}) \lor P) & = \{ \text{RT}_1 \} \\
(\text{RT}_0 \circ T_0(\neg \text{ok}) \lor P) \land rt' = rt \land tt' & = \{ \text{de Morgans} \} \\
(\text{RT}_0 \circ T_0(\neg \text{ok}) \land rt' = rt \land tt') \lor (P \land rt' = rt \land tt') & = \{ \text{idempotence of RT}_1 \} \\
\text{RT}_0 \circ T_0(\neg \text{ok}) \lor (P \land rt' = rt \land tt') & = \{ \text{RT}_4 \} \\
\text{RT}_4(P \land rt' = rt \land tt') & = \{ \text{RT}_1 \} \\
\text{RT}_4(\text{RT}_1(P)) & = \{ \text{de Morgans} \} \\
\end{align*}
\]

9. $\text{RT}_3(\text{RT}_2(P)) = \text{RT}_2(\text{RT}_3(P))$

\[
\begin{align*}
\text{RT}_2(\text{RT}_3(P)) & = \{ \text{RT}_3 \} \\
\text{RT}_2(\Xi_{rt} \triangleleft \text{wait} \triangleright P) & = \{ \text{RT}_2 \} \\
(\Xi_{rt} \triangleleft \text{wait} \triangleright P)[\emptyset, tt'/rt, rt'] & = \{ \Xi_{rt} \} \\
(\text{RT}(\text{true} \vdash \Xi)[\emptyset, tt'/rt, rt']) \lt \text{wait} \triangleright P[\emptyset, tt'/rt, rt'] & = \{ \text{distribution} \} \\
(\text{RT}(\text{true} \vdash \Xi)[\emptyset, tt'/rt, rt']) \lt \text{wait} \triangleright P[\emptyset, tt'/rt, rt'] & = \{ \text{RT} = \text{RT} \circ \text{RT}_2 \} \\
(\text{RT}(\text{true} \vdash \Xi)[\emptyset, tt'/rt, rt']) \lt \text{wait} \triangleright P[\emptyset, tt'/rt, rt'] & = \{ \Xi_{rt} \} \\
(\Xi_{rt} \triangleleft \text{wait} \triangleright P[\emptyset, tt'/rt, rt']) & = \{ \text{RT}_3 \} \\
\text{RT}_3(P[\emptyset, tt'/rt, rt']) & = \{ \text{RT}_2 \} \\
\text{RT}_3(\text{RT}_2(P)) & = \{ \text{RT}_3 \}
\end{align*}
\]

10. $\text{RT}_4(\text{RT}_2(P)) \implies \text{RT}_2(\text{RT}_4(P))$
\( RT4(\text{RT2}(P)) \)
\[ = \{ RT4 \} \]
\[ RT01 \circ T0(\neg ok) \lor RT2(P) \]
\[ = \{ RT2 \} \]
\[ RT01 \circ T0(\neg ok) \lor P[\{\}, tt'/rt, rt'] \]
\[ \Rightarrow \{ tt' = () \land tt' \land () \in \text{timedTrace} \land tt' \in \text{timedTrace} \} \]
\[ (RT01 \circ T0(\neg ok) \land tt' = () \land tt' \land () \in \text{timedTrace} \land tt' \in \text{timedTrace}) \]
\[ \lor P[\{\}, tt'/rt, rt'] \]
\[ = \{ \} \]
\[ (RT01 \circ T0(\neg ok) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace} \land rt' = tt \land tt')[\{\}, tt'/rt, rt'] \]
\[ \lor P[\{\}, tt'/rt, rt'] \]
\[ = \{ RT01 \} \]
\[ (RT01 \circ T0(\neg ok) \lor P)[\{\}, tt'/rt, rt'] \]
\[ = \{ RT4 \} \]
\[ (RT4(P))[\{\}, tt'/rt, rt'] \]
\[ = \{ RT2 \} \]
\[ RT2(\text{RT4}(P)) \]

11. \( RT4(\text{RT3}(P)) = RT3(\text{RT4}(P)) \)

\( RT3(\text{RT4}(P)) \)
\[ = \{ RT4 \} \]
\[ RT3 \circ RT01 \circ T0(\neg ok) \lor P \]
\[ = \{ RT3 \} \]
\[ \Pi rt \land \text{wait} \triangleright (RT01 \circ T0(\neg ok) \lor P) \]
\[ = \{ RT4(\Pi rt) = \Pi rt \} \]
\[ \Pi rt \lor RT01 \circ T0(\neg ok) \land \text{wait} \triangleright (RT01 \circ T0(\neg ok) \lor P) \]
\[ = \{ \text{distribute or through if} \} \]
\[ (\Pi rt \land \text{wait} \triangleright P) \lor (RT01 \circ T0(\neg ok)) \]
\[ = \{ RT3 \} \]
\[ RT3(P) \lor (RT01 \circ T0(\neg ok)) \]
\[ = \{ RT4 \} \]
\[ RT4(\text{RT3}(P)) \]

12. \( RT0(\text{RT5}(P)) = RT5(\text{RT0}(P)) \)
\[ \text{RT5}(\text{RT0}(P)) \]
\[ = \{ \text{RT5} \} \]
\[ \text{RT0}(P) ; J \]
\[ = \{ \text{RT0} \} \]
\[ (P \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace}) ; J \]
\[ = \{ \text{relational composition} \} \]
\[ \exists ok_0 \bullet (P \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace})[ok_0/ok'] \land J[ok_0/ok] \]
\[ = \{ \text{contract scope} \} \]
\[ \exists ok_0 \bullet (P[ok_0/ok'] \land J[ok_0/ok]) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace} \]
\[ = \{ \text{relational composition} \} \]
\[ (P ; J) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace} \]
\[ = \{ \text{RT5} \} \]
\[ \text{RT5}(P) \land rt \in \text{timedTrace} \land rt' \in \text{timedTrace} \]
\[ = \{ \text{RT0} \} \]
\[ \text{RT0}(\text{RT5}(P)) \]

13. \[ \text{RT1}(\text{RT5}(P)) = \text{RT5}(\text{RT1}(P)) \]

\[ \text{RT1}(\text{RT5}(P)) \]
\[ = \{ \text{RT5} \} \]
\[ \text{RT1}(P ; J) \]
\[ = \{ \text{RT1} \} \]
\[ P ; J \land rt' = rt \land tt' \]
\[ = \{ \text{relational composition} \} \]
\[ \exists ok_0 \bullet P[ok_0/ok'] \land J[ok_0/ok] \land rt' = rt \land tt' \]
\[ = \{ \text{contract scope} \} \]
\[ \exists ok_0 \bullet (P \land rt' = rt \land tt')[ok_0/ok'] \land J[ok_0/ok] \]
\[ = \{ \text{relational composition} \} \]
\[ (P \land rt' = rt \land tt') ; J \]
\[ = \{ \text{RT1} \} \]
\[ \text{RT1}(P) ; J \]
\[ = \{ \text{RT5} \} \]
\[ \text{RT5}(\text{RT1}(P)) \]

14. \[ \text{RT2}(\text{RT5}(P)) = \text{RT5}(\text{RT2}(P)) \]
\[ \text{RT5}(\text{RT2}(P)) = \{ \text{RT2} \} \]
\[ \text{RT5}(P[\ell, tt' / rt, rt']) = \{ \text{RT5} \} \]
\[ (P[\ell, tt' / rt, rt']) ; J = \{ \text{relational composition} \} \]
\[ \exists v_0 \cdot P[\ell, tt' / rt, rt'][v_0/v' \land ((ok_0 \Rightarrow ok') \land rt' = rt \land tt' = tt \land v' = v)[v_0/v] = \{ \text{predicate calculus} \} \]
\[ \exists v_0 \cdot (P[\ell, tt' / rt, rt'][v_0/v] \land (ok_0 \Rightarrow ok')) = \{ \text{rt' free in ok_0 \Rightarrow ok'} \} \]
\[ \exists v_0 \cdot (P \land (ok_0 \Rightarrow ok'))[v_0/v'][\ell, tt' / rt, rt'] = \{ \text{free in ok_0 \Rightarrow ok'} \} \]
\[ \exists v_0 \cdot (P[v_0/v'] \land (ok_0 \Rightarrow ok'))[\ell, tt' / rt, rt'] = \{ \text{predicate calculus} \} \]
\[ (\exists v_0 \cdot P[v_0/v] \land (ok_0 \Rightarrow ok') \land rt' = rt_0 \land tt' = tt_0 \land v' = v_0)[\ell, tt'/rt, rt'] = \{ \text{relational composition, definition of J} \} \]
\[ (P ; J)[\ell, tt' / rt, rt'] = \{ \text{RT2} \} \]
\[ \text{RT2}(P ; J) = \{ \text{RT5} \} \]
\[ \text{RT2}(\text{RT5}(P)) \]

15. \[ \text{RT3}(\text{RT5}(P)) = \text{RT5}(\text{RT3}(P)) \]

\[ \text{RT5}(\text{RT3}(P)) = \{ \text{RT3} \} \]
\[ \text{RT5}(\Pi_{\text{RT} \leftarrow \text{wait} \Rightarrow P}) = \{ \text{RT5} \} \]
\[ (\Pi_{\text{RT} \leftarrow \text{wait} \Rightarrow P}) ; J = \{ \text{J distributes through conditional} \} \]
\[ \Pi_{\text{RT} \\; J \leftarrow \text{wait} \Rightarrow P} = \{ \text{RT3} \} \]
\[ \text{RT3}(P ; J) = \{ \text{RT5} \} \]
\[ \text{RT3}(\text{RT5}(P)) \]
16. $RT_4(RT_5(P)) = RT_5(RT_4(P))$

\[
RT_5(RT_4(P))
\]
\[
= \{ RT_4 \}
\]
\[
RT_5(RT_0 \circ T_0(\neg ok) \lor P)
\]
\[
= \{ RT_5 \}
\]
\[
(RT_0 \circ T_0(\neg ok) \lor P) ; J
\]
\[
= \{ J \text{ distributes through disjunction} \}
\]
\[
(RT_0 \circ T_0(\neg ok) ; J) \lor P ; J
\]
\[
= \{ \text{propositional calculus} \}
\]
\[
RT_0 \circ T_0(\neg ok) \lor (P ; J)
\]
\[
= \{ RT_4 \}
\]
\[
RT_4(P ; J)
\]
\[
= \{ RT_5 \}
\]
\[
RT_4(RT_5(P))
\]
CML Definition 2 — Hoare Logic

Deliverable Number: D23.3-2

Version: 0.4

Date: 7 March 2013

Public Document

http://www.compass-research.eu
Contributors:

Samuel Canham, York  
Ana Cavalcanti, York  
Augusto Sampaio, UFPE  
Jim Woodcock, York

Editors:

Jim Woodcock, York

Reviewers:

Joey Coleman, Aarhus  
Marcel Oliveira, UFPE  
Richard Payne, Newcastle
Document History

<table>
<thead>
<tr>
<th>Ver</th>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>06-03-2013</td>
<td>Samuel Canham</td>
<td>Initial document version</td>
</tr>
<tr>
<td>0.2</td>
<td>06-03-2013</td>
<td>Jim Woodcock</td>
<td>Working document version</td>
</tr>
<tr>
<td>0.3</td>
<td>29-03-2013</td>
<td>Samuel Canham</td>
<td>Update following Internal Review</td>
</tr>
<tr>
<td>0.4</td>
<td>30-03-2013</td>
<td>Jim Woodcock</td>
<td>Sundry changes</td>
</tr>
</tbody>
</table>
Contents

1 Hoare Logic for CML
  1.1 Axiom schema for STOP, SKIP and assignment .................. 6
  1.2 Sequential Composition ........................................ 6
  1.3 Prefix .......................................................... 7
  1.4 Internal Choice .................................................. 7
  1.5 External Choice .................................................. 7
  1.6 Parallel Composition ............................................ 7
  1.7 Hiding ............................................................ 8
  1.8 Timeout .......................................................... 8
  1.9 Recursion ......................................................... 8
    1.9.1 While loops .................................................. 9
    1.9.2 While loop Examples ....................................... 9
Chapter 1

Hoare Logic for CML

A Hoare Logic is constructed to underpin the soundness of the UTP semantics. As part of the construction of this document, a revised UTP semantics was developed in a form more convenient for the logic, which is used throughout this section.

A Hoare Logic is a proof system used to construct proofs for Hoare triples. A Hoare triple takes the form \( \{P\} C \{Q\} \), which states that if the command \( C \) is executed in a state satisfying an assumption \( P \) and terminates, then it will terminate in a state satisfying the commitment \( Q \). In the semantics, this is equivalent to the following definition, which takes into account the Healthiness conditions.

**Definition 1.0.1 (Hoare triple)**

\[
\{P\} C \{Q\} \equiv \text{RT}(P \vdash Q) \sqsubseteq C
\]

As usual for a CML process, we assume that \( P \) and \( Q \) are free in \( \text{ok} \) and \( \text{ok}' \). Since \( P \) and \( Q \) are made into an \( \text{RT} \)-healthy design, we also have that they are free in \( \text{wait} \), since they are screened by \( \text{RT3} \). We will further require that they are both \( \text{RT2} \)-healthy, and also that \( P \) is free in \( v' \) and \( \text{wait}' \), since it is an assumption.

This definition can be expanded and simplified into the following form, which is useful for calculating preconditions for Hoare triples.

**Lemma 1.0.1 (Equivalent Form)**

\[
\{P\} C \{Q\} \equiv \text{RT16}(P) \Rightarrow (\neg C_t^f \land (C_f^t \Rightarrow Q))
\]

where \( \text{RT16} = \text{RT1} \circ \text{RT6} \).

For given \( Q \), the most general form of \( P \) for which this statement is true satisfies \( P = \text{RT16}(\text{true}) \Rightarrow (\neg C_t^f \land (C_f^t \Rightarrow Q)) \). The task is then to evaluate this expression removing explicit references to \( C \).

For an operator which is defined explicitly in terms of the observation variables and does not refer to other existing operators, those references can be removed by replacing them with their defined values and simplifying as appropriate. The result will give an axiom schema, which allows exact Hoare triples regarding that operator to be introduced for any of the possible values of \( Q \).
Hoare triples for operators which are defined referring to other commands cannot know anything about the commands they are applied to, other than that they are CML processes. Consequently, while the expressions for $C(f)$ and $C(t)$ can be rearranged to depend only on the constituent commands, these expressions cannot be expanded further. The required simplification can then be found by using Hoare Triples for the constituent commands. The result is an Inference Rule that, given a hypothesis about the constituent commands, allows a conclusion in the form of a Hoare triple for the composite command to be deduced.

If only one constituent command $C_1$ is required for an operator, the hypothesis will always be $\{P_1\}C_1\{Q_1\}$, and if a second $C_2$ is required, it will include also $\{P_2\}C_2\{Q_2\}$. We take the approach of constructing an operator $P$ out of $P_1$ and $P_2$ which guarantees the precondition of $C$, then calculate the maximal condition $Q$ which can be deduced as a commitment from $P$ in terms of $Q_1$ and $Q_2$. When this is applied, $Q_1$ and $Q_2$ should be chosen to generate the necessary condition $Q$ and then $P_1$, $P_2$ and $P$ calculated subsequently.

### 1.1 Axiom schema for STOP, SKIP and assignment

**Lemma 1.1.1 (STOP)**

$$\{RT1(events(tt') = \langle \rangle) \Rightarrow \forall v_0 \cdot Q[true, v_0/wait', v']\} \text{STOP}\{Q\}$$

To guarantee that STOP will establish a condition, provided that no events are observed, that condition must hold as a precondition, with wait' true, for all possible values of the variables.

**Lemma 1.1.2 (Assignment)**

$$\{RT1(events(tt') = \langle \rangle) \Rightarrow Q[true, e/wait', v']\{v := e\}\} \text{STOP}\{Q\}$$

To guarantee that assignment will establish a condition, provided that no events are observed, that condition must hold as a precondition, with wait' false and the variables taking their assigned values.

**Lemma 1.1.3 (SKIP)**

$$\{RT1(events(tt') = \langle \rangle) \Rightarrow Q[true, v/wait', v']\} \text{SKIP}\{Q\}$$

To guarantee that SKIP will establish a condition, provided that no events are observed, that condition must hold as a precondition, with wait' false and the variables taking their initial values as final values.

### 1.2 Sequential Composition

Assuming the necessary hypothesis,

**Lemma 1.2.1 (Sequential Composition)**

$$\{RT16(true) \Rightarrow (RT16(P_1) ; true \land \neg (Q_1 ; RT16(P_2)) \land (Q_1 \land Q_2 \Rightarrow Q) ) \} C_1 ; C_2\{Q\}$$
1.3 Prefix

Let \( clean(t, a) = (\text{events}(t) = \emptyset) \land a \notin \text{refusals}(t) \). Assuming the necessary hypothesis,

Lemma 1.3.1 (Prefix)

\[
\begin{align*}
\text{RT16}(\text{true}) & \Rightarrow \\
(\forall u \cdot clean(u, a) \land u \preceq \langle a \rangle) \land (\langle a \rangle \leq tt' \Rightarrow P_1[rt \land u \preceq \langle a \rangle / rt]) \\
& \land (\exists clean(tt', a) \Rightarrow Q[\text{true}, v / wait', v']) \\
& \land \forall wait_0, v_0 \cdot clean(u, a) \\
& \land (\exists a \preceq \langle a \rangle) \land (\langle a \rangle \leq tt' \Rightarrow Q[rt \land u \preceq \langle a \rangle, wait_0, v_0 / wait', v']) \\
& \Rightarrow Q[wait_0, v_0 / wait', v'] \\
\end{align*}
\]

\( a \rightarrow C\{Q\} \)

1.4 Internal Choice

Assuming the necessary hypothesis,

Lemma 1.4.1 (Internal choice)

\[
\{ P_1 \land P_2 \land \forall wait_0, v_0 \cdot (Q_1 \lor Q_2 \Rightarrow Q)[wait_0, v_0 / wait', v'] \} C_1 \sqcap C_2 \{Q\}
\]

1.5 External Choice

Assuming the necessary hypothesis,

Lemma 1.5.1 (External choice)

\[
\{ P_1 \land P_2 \land \forall wait_0, v_0 \cdot (Q_1 \lor Q_2 \Rightarrow Q)[rt \land \text{idleprefix}(tt') / rt'] \} [wait_0, v_0 / wait', v'] C_1 \sqcap C_2 \{Q\}
\]

1.6 Parallel Composition

Assuming the necessary hypothesis,

Lemma 1.6.1 (Parallel composition) for disjoint \( P \) and \( Q \)

\[
\begin{align*}
P_1 \land P_2 \land \forall wait_0, v_0, wait_1, wait_2, tt_1, tt_2 & \bullet \\
\text{RT16} & \Rightarrow (tt' \in tt_1 \parallel_A tt_2) \\
& \land (wait_0 = wait_1 \lor wait_2) \\
& \land Q_1[wait_1, v_0, rt \land tt_1 / wait', v', rt'] \\
& \land Q_2[wait_2, v_0, rt \land tt_2 / wait', v', rt'] \\
& \Rightarrow Q[wait_0, v_0 / wait', v'] \\
& \parallel_A C_2 \{Q\}
\end{align*}
\]
1.7 Hiding

Assuming the necessary hypothesis,

Lemma 1.7.1 (Hiding)

\[
\begin{align*}
RT_{16}(\forall u \cdot A \subseteq \text{rejects}(u) \land u/A = tt') \\
\land \forall wait_0, v_0 \cdot \\
\Rightarrow \left( \exists u \cdot \text{Asubseteqrejects}(u) \land u/A = tt' \land Q_1[rt \leftarrow u, wait_0, v_0/rt', wait', v'] \right) \\
\Rightarrow Q[wait_0, v_0/\text{wait}', v']
\end{align*}
\]

1.8 Timeout

Assuming the necessary hypothesis,

Lemma 1.8.1 (Timeout)

\[
\begin{align*}
\forall tl_0, v_0 \cdot \\
RT_{16}(\text{tock}^n = \text{tock}(tl_0) \\
\land Q_1[rt \leftarrow tl_0, v_0, false/rt', v', \text{wait}'] \\
\land \forall wait_1, v_1, rt_0, v_0 \cdot \\
rt \leq rt_0 \\
\land \text{tock}^n = \text{tocks}(rt_0 - rt) \\
\land Q_1[rt_0, v_0, false/rt', v', \text{wait}'] \\
\land \left( Q_2[rt_0, v_0/rt, v] \land \langle \text{tock}^n \leq \text{tocks}(tt') \rangle \Rightarrow Q \right) \\
\Rightarrow C_1^n \triangleright C_2[Q]
\end{align*}
\]

1.9 Recursion

An approximation chain for a condition \( B \) is a series of conditions \( E_n \) such that \( E_0 = \text{false} \) and \( \forall n \cdot E_n \Rightarrow E_{n+1} \), which satisfy \( B = \bigvee_i E_i \), as defined by Hoare. \( F \) is \( E \)-constructive if \( \forall n \cdot E_n \land F(X) = E_n \land F(X \land E_{n+1}) \), again, as defined by Hoare. If \( E_n \) is such a chain, then \( B \) is a condition for the strongest and weakest fixed points to be the same — a termination condition.

Supposing \( P \) guarantees \( B \), then \( RT(P \vdash Q) \) refines \( \mu F \) precisely when \( F(RT(P \vdash Q)) \Rightarrow RT(P \vdash Q) \). Equivalently, using the functions \( F_1 \) and \( F_2 \),

\[
RT_{16}(P \Rightarrow (F_1(P, Q) \land (F_2(P, Q) \Rightarrow Q))
\]

The expressions \( F_1 \) and \( F_2 \) are arbitrary and need not necessarily simplify well. However, if such a form can be found, \( \{P\} \mu X \cdot F(X) \{Q\} \) can be deduced. If \( F \) is composed of CML operators, the Hoare triple can usually be inferred from suitable Hoare triples about the component operators.
1.9.1 While loops

This technique will be demonstrated for the while loop $b * C$, where $b$ is a condition on undashed variables only. We have $\mu X \bullet (C ; X) < b \triangleright Skip$. If, at the end of any loop, $b$ is false, then $Q$ must be true. Conversely, if $b$ is true, the precondition for the next loop must be true. Initially, if $b$ is true, the precondition for the first loop must be true.

Define $C_n = (b \land C)^n ; (\neg b \land Skip)$. Then take

$$E_0 = false, E_1 = C_0, E_2 = C_1 \lor E_1$$

$$E_n = \exists i \leq n \bullet C_i$$

$$B = \exists m \bullet C_m$$

Suppose that

$$\forall n > 0 \bullet \{P_n + 1 \land b\} C\{P_n[w'/w] < b' \triangleright Q\}$$

and that

$$\{P_0 \land b\} C\{\neg b' \land Q\}$$

$$F_1(P, Q) = b \Rightarrow (\neg C_f ; true) \land \neg (C_f'[false/wait'] \land \neg P)$$

$$F_2(P, Q) = (C_f ; (II < wait \triangleright Q) < b \triangleright v' = v \land \neg wait' \land events(t') = \emptyset)$$

We require a $P$ satisfying

$$RT_{16}(P \Rightarrow (F_1(P, Q) \land (F_2(P, Q) \Rightarrow Q))$$

and $P \Rightarrow B$.

Claim:

$$RT_{16}(true) \Rightarrow \exists m \bullet P_m < b \triangleright \forall u \mid events(u) = \emptyset \bullet (Q[v, rt \leftarrow u, false/v', rt', wait'])$$

satisfies the above. Proof: If $b$ is false, $F_1(P, Q) = true$ and $P = \forall u \mid events(u) = \emptyset \bullet Q[v, rt \leftarrow u, false/v', rt', wait']$, and $Q$ satisfies $B$ with index 0. If $b$ is true, we have $P_m \land b \Rightarrow \neg C_f ; true$ and $P_m \land b \land C_f' \Rightarrow P_{m-1}[w'/w] < b' \triangleright Q$. We require that $P_m \Rightarrow \forall u_0 \mid\{P_{m-1}[w_0/w] < b[u_0/w'] \triangleright Q[u_0/w'] \land \neg wait_0 \bullet P[u_0/w]\}$. Since $P$ is free in $w'$, if $m > 0$, this reduces to $\forall u_0 \mid\{P_{m-1}[w_0/w] < b[u_0/w'] \triangleright Q[u_0/w'] \land \neg wait_0 \bullet P[u_0/w]\}$, and both these expressions are of form $P$ with $m - 1$ and 0 as indices respectively. If $m = 0$, only the 0 case occurs. QED

1.9.2 While loop Examples

Suppose $C = (i := i + 1; v := v * i)$, $b = i < n$ and $Q = (v' = n!)$. Take $P_m = (m + i) = n \land v = i!$. This has the required form. Then

$$P = RT_{16}(true) \Rightarrow \exists m \bullet P_m < i < n \triangleright v' = n![v/v']$$

$$= RT_{16}(true) \Rightarrow v = i! < i < n \triangleright n!$$
So,

\[
\{ \text{RT16}(\text{true}) \Rightarrow v = i! \land i < n \land n! \} (i < n) \ast (i := i + 1; v := v \ast i)\{v' = n!\}
\]

Suppose

\[
C = a \rightarrow t := 1 \triangleright t := 0 \triangleleft t = 1 \triangleright v := v + 1
\]

\[b = (v < 5), \, Q = \text{events}(tt') \neq \emptyset.\] This process wants the enviroment to engage in event \(a\) at least once each time unit. If an \(a\) occurs, the flag \(t\) is set to 1 and the loop terminates. If the timeout activates and the value of \(t\) is 1, the flag is reset and the process continues. If \(t\) is 0, no \(a\) occurred in the interval so \(v\) is incremented, keeping track of the number of intervals in which an \(a\) didn’t occur. When this reaches 5, the process terminates.

To ensure this process terminates, we label the events at which time passes.

\[
B = \exists X \mid \#X = 5 \bullet \#tt' = X \Rightarrow \text{events}(\text{last}(tt')) = \emptyset
\]

\[
P = \text{RT16}(\text{true}) \Rightarrow \exists X \mid \#X = 5 \bullet \#tt' = X \Rightarrow \text{events}(\text{last}(tt')) = \emptyset \land \exists y \mid y < 5 \bullet y \notin X
\]
CML Definition 2 — Object Orientation

Deliverable Number: D23.3-3
Version: 0.2
Date: 31 March 2013
Public Document

http://www.compass-research.eu
Contributors:

Thiago Santos, UFPE
Augusto Sampaio, UFPE
Ana Cavalcanti, York

Editors:

Jim Woodcock, York

Reviewers:

Joey Coleman, Aarhus
Marcel Oliveira, UFPE
Richard Payne, Newcastle
## Document History

<table>
<thead>
<tr>
<th>Ver</th>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>24-02-2013</td>
<td>Jim Woodcock</td>
<td>Initial document version</td>
</tr>
<tr>
<td>0.2</td>
<td>27-03-2013</td>
<td>Jim Woodcock</td>
<td>Updated following internal review</td>
</tr>
</tbody>
</table>
## Contents

1 Object Orientation in UTP .................................................. 5
  1.1 Introduction ..................................................................... 5
  1.1.1 Assumptions .............................................................. 6
  1.2 Observational Variables .................................................. 7
  1.3 Healthiness Conditions .................................................... 8
  1.4 Declarations .................................................................. 10
    1.4.1 Classes ................................................................. 11
    1.4.2 State Components .................................................... 13
    1.4.3 Operations ............................................................. 15
  1.5 Variables ....................................................................... 17
  1.6 Expressions .................................................................... 18
    1.6.1 Well-definedness ...................................................... 18
    1.6.2 Object Creation ........................................................ 20
    1.6.3 Type Test ............................................................... 20
    1.6.4 Type Cast ............................................................... 21
    1.6.5 State Component Access .......................................... 21
  1.7 Commands ..................................................................... 21
    1.7.1 Well-definedness ...................................................... 22
    1.7.2 Assignments ............................................................ 23
    1.7.3 Conditional ............................................................. 25
    1.7.4 Recursion ............................................................... 25
    1.7.5 Operation Call ........................................................ 26
  1.8 Conclusions ................................................................... 28
    1.8.1 Verification ............................................................. 28
Chapter 1

Object Orientation in UTP

We describe a UTP theory for object orientation that closely follows the work of Thiago Santos, Augusto Sampaio, and Ana Cavalcanti. This theory forms the basic treatment of the object-oriented features of CML. The state part of CML is object oriented: it allows the definition of classes with state components and operations, inheritance, and dynamic binding. The behavioural features of the language is not object oriented. Like other language features, object orientation is treated orthogonally. Galois connections will establish the links between this theory and the rest of the language semantics; this topic will be addressed in deliverables D23.4 and D23.5.

Section 1.1 introduces the approach taken in this report to defining the semantics of object orientation in UTP. Section 1.2 introduces the alphabet for the theory of object orientation in UTP, describing the observational variables used to capture subtyping, inheritance, and dynamic binding. Section 1.3 gives the healthiness conditions for the theory and proves some laws. Section 1.4 defines class, state component, and operation declarations. Section 1.5 considers how to capture type information explicitly for variables. Section 1.6 describes well-definedness rules for expressions and the meaning of object creation, type test, type cast, and state component access. Section 1.7 reviews the semantics of commands emphasising operation calls. Finally, Section 1.8 concludes the report.

1.1 Introduction

In this report we show how subtyping, inheritance, mutually recursive operations, and dynamic binding can be described in UTP by combining and extending the theories of designs and higher-order procedures. Our approach is modular, with each language feature being described in isolation. A copy semantics for an object-oriented language treats objects as denoting values; a reference semantics treats objects as denoting references to values. In this deliverable, we separate the concerns of object-oriented constructs and pointers and consider only copy semantics for objects; reference semantics will be addressed in deliverable D23.4.

In the UTP theory we present, we have separate constructs to declare classes and their

---

immediate superclasses, state components, and operations. This follows the approach presented in [8].

**Example 1.1.1 (Syntax example)** Consider a simple banking system; we define a class `Account`, and its state components and operations as follows.

```plaintext
class Account
begin
  state
    Account accno: Z := 100
    Account bal : Z := 0
  operations
    Account Credit =
    (val x : Z •
      self.balance := self.balance + x)
end
```

The declarations of the state components and operations are independent, and combined in sequence; in particular, the declarations indicate the classes of the state components and operations that are introduced. This approach simplifies the semantics, and makes the treatment of (mutual) recursion straightforward.

Types play a central role in the semantics of an object-oriented language due to subtyping and dynamic binding [5]. In our theory, we have a collection of observational variables that are used to model declarations. They record important typing information and are used to define the semantics of commands. We do not assume that expressions are total, due to the possibility of attempting to access state components and operations of a `null` object. As a consequence, we have to characterise well-defined expressions, and extend the semantics of assignments and conditionals accordingly.

Operation names are part of the alphabet of our theory: their values are parametrised programs in the sense of Back [1]. Their treatment follows the approach originally proposed in [9], and adopted in [5] to handle operations. It is also the approach followed in the UTP for higher-order procedures.

Dynamic binding is reflected in the value of an operation variable: it is conditional on the type of the target object (`self`) and determines the right program that defines the behaviour of the operation in each case. In this way, we capture dynamic binding in isolation. This follows the style adopted in the algebraic semantics for object-orientation presented in [4].

### 1.1.1 Assumptions

Object-oriented features such as state component overriding, variable shading, and the use of `super` or related notations (to refer to elements of a superclass) are not considered here. They are only syntactic abbreviations that can be easily eliminated by preprocessing.

We consider that the names of classes, state components, operations (except for operation overriding), local variables, and parameters belong to different namespaces. This allows us to write simpler predicates, while not imposing any relevant practical limitation, both in denotational semantics and in algebraic laws.
1.2 Observational Variables

In addition to the programming variables, their dashed counterparts, and the design theory observational variables $ok$ and $ok'$, our theory includes several extra observational variables. We introduce a variable $cls$ to record class names; a variable $sc$ to record the subclass relation; and a variable $atts$ to record the names and types of the state components of every class.

**Definition 1.2.1 (Classes)** The set of classes is recorded in the observational variable $cls : \mathbb{P} Name$.

This observational variable allows us to introduce new types in addition to the primitive ones. An example is given below:

$$cls_0 = \{ \texttt{Object} \}$$

**Object** is a valid class name.

**Definition 1.2.2 (Subclasses)** The subclass relation is recorded in $sc : Name \rightarrow Name$.

This is a mapping that maps a class name to the name of its immediate superclass. An example is

$$sc_0 = \emptyset$$

**Object** does not have a parent, and so cannot be present in the domain of $sc$. This is the subject of the healthiness condition $OO2$ presented in the next section.

Using $sc$, we define the subtyping relation $A \preceq B$ that holds if $A$ is related to $B$ in the reflexive, transitive closure $sc^*$. The inclusion of primitive types (such as $\mathbb{B}$ (booleans) and $\mathbb{Z}$ (integers)) allows us to simplify definitions.

$$\preceq \triangleq sc^*$$

The subtyping relation is important in an object-oriented context to establish the well-definedness of assignments and state component accesses, as we explain in Sections 1.6.1 and 1.7.1. The strict subtyping relation is denoted by $<$, and is defined by

$$A \prec B \triangleq (A \preceq B) \land (A \neq B)$$

**Definition 1.2.3 (State Components)** The state component information is recorded in $atts : Name \rightarrow (Name \rightarrow Type)$.

This is a mapping from a class name to a description of its state components that maps each state component name to its type. Here $Type$ stands for any primitive type or any name in $cls$, that is,

$$Type \triangleq \{ \mathbb{B}, \mathbb{Z} \} \cup cls$$

Once again, because of **Object**’s special characteristics, we define a special value for this mapping to be used in examples. In this case, we have a mapping which says that the set of state components for **Object** is empty.

$$atts_0 = \{ \texttt{Object} \mapsto \emptyset \}$$

Notice, however, that using the normalisation strategy presented in [4], this set of state components associated with **Object** can be extended.
Definition 1.2.4 (Operations) Operation names are part of the alphabet of the theory. Their values are texts of parametrised programs \((\text{pds} @ \text{p})\), where \text{pds} is a list of parameter declarations and \text{p} is a program: the body of the parametrised program that uses the parameters.

We write \((\text{pds} @ \text{p})\) rather than \((\text{pds} \cdot \text{p})\) to indicate that the values are texts \((\text{pds} @ \text{p})\) of parametrised programs, rather than their meanings \((\text{pds} \cdot \text{p})\). Value \((\text{val})\), result \((\text{res})\), and value-result \((\text{vres})\) parameters are allowed. The notation \text{pds} stands for any parameter declaration list, possibly including the three parameter passing mechanisms. For example,

\[
\begin{align*}
\text{val} & \quad x : X ; \quad \text{res} \quad y : Y ; \quad \text{vres} \quad z : Z
\end{align*}
\]

is a valid instance of \text{pds}, where \(x\), \(y\), and \(z\) are variable names and \(X\), \(Y\), and \(Z\) are their types. The function \text{types} applied to a list of parameter declarations yields the parameter types as a set. For instance, \text{types} applied to the example above yields \{\(X\), \(Y\), \(Z\)\}.

The values of the observational variables named after operations are parametrised nested conditionals with each branch representing the meaning of an operation redefinition. For instance, considering that \(C\) is a subclass of \(B\), which is itself a subclass of \(A\), and that \(m\) is a parameterless operation defined in \(A\) (with body \(ma\)), and redefined in both \(B\) and \(C\), with bodies \(mb\) and \(mc\), the value of \(m\) is

\[
\begin{align*}
\text{vres} & \quad \text{self} : \text{Object} @ \\
& \quad \text{mc} \leftarrow \text{self is } C \triangleright (\text{mb} \leftarrow \text{self is } B \triangleright (\text{ma} \leftarrow \text{self is } A \triangleright \text{abort}))
\end{align*}
\]

Based on the type of the current object \text{self}, the nested conditional allows selection of the most specialised version of \(m\). When \(m\) is not defined for a given class, then the behaviour of a call to \(m\) with an object of this class as a target is unpredictable (the bottom predicate of our theory \(\bot_{OO}\), defined in the next section). The type test \text{self is } N\), for a class name \(N\), checks whether the value of \text{self} is an object of class \(N\), or one of its subclasses. This is why, in the type tests for \text{self}, the more specialised classes are considered first. In UTP, programs (and specifications) are predicates; there is no notation to distinguish the text of a program from its semantics. Here, just like in [HH98, Chapter 10] we introduce the distinction. The values of operation observational variables have to be texts to allow the use of a syntactic function to capture dynamic binding (see Section 1.4.3).

As usual, for each programming variable \(x\), the alphabet contains \(x\) and \(x'\); but we also include two more observational variables \(x_t\) and \(x'_t\) to record the declared type of \(x\). This is potentially different from the actual (runtime) type of the value of \(x\), which can be an object of a subclass of the type recorded in \(x_t\), when this is a class.

1.3 Healthiness Conditions

The following healthiness conditions characterise our theory of object-oriented relations. Healthiness condition \text{OO1} says that \text{Object} is a valid type.

\[
\text{OO1} \quad P = P \land \text{Object} \in \text{cls}
\]

Our theory relies on a superclass of all classes, represented by the \text{Object} type. An example of a \text{cls} instance that satisfies this predicate is \(\text{cls}_0\) (defined on page 7). Healthiness condition
**OO2** says that every class has a parent, except **Object**.

\[ OO2 \quad P = P \land (\text{dom } \text{sc} = \text{cls} \setminus \{\text{Object}\}) \]

The top-most superclass for all classes is **Object**, therefore cyclic references are not allowed. Healthiness condition **OO3** says that a parent class in \(\text{sc}\) is necessarily present in \(\text{cls}\).

\[ OO3 \quad P = P \land \forall C : \text{dom } \text{sc} \bullet (C, \text{Object}) \in \text{sc}^+ \]

Healthiness condition **OO4** says that, for all classes present in \(\text{cls}\), there is a corresponding mapping in \(\text{atts}\) that records the state component names and types.

\[ OO4 \quad P = P \land (\text{dom } \text{atts} = \text{cls}) \]

Healthiness condition **OO5** says that the names of state components are different for all classes.

\[ OO5 \quad P = P \land \forall C_1, C_2 : \text{dom } \text{atts} \bullet (C_1 \neq C_2) \Rightarrow (\text{dom}(\text{atts}(C_1)) \cap \text{dom}(\text{atts}(C_2)) = \emptyset) \]

Healthiness condition **OO6** says that all state components must have valid types, primitives or defined in \(\text{cls}\).

\[ OO6 \quad P = P \land (\text{ran}(\bigcup \text{ran } \text{atts}) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup \text{cls}) \]

We name the composition of these conditions

**OOI \cong OO1 \circ \cdots \circ OO6**

These healthiness conditions constrain the initial values of the variables. Predicates in our theory must preserve these properties for the final values of these observational variables; thus we have a similar set of healthiness conditions for the output variables.

\[ OO7 \quad P = P \land \text{Object} \in \text{cls}' \]
\[ OO8 \quad P = P \land (\text{dom } \text{sc}' = \text{cls}' \setminus \{\text{Object}\}) \]
\[ OO9 \quad P = P \land \forall C : \text{dom } \text{sc}' \bullet (C, \text{Object}) \in \text{sc}'^+ \]
\[ OO10 \quad P = P \land (\text{dom } \text{atts}' = \text{cls}') \]
\[ OO11 \quad P = P \land \forall C_1, C_2 : \text{dom } \text{atts}' \bullet (C_1 \neq C_2) \Rightarrow (\text{dom}(\text{atts}'(C_1)) \cap \text{dom}(\text{atts}'(C_2)) = \emptyset) \]
\[ OO12 \quad P = P \land (\text{ran}(\bigcup \text{ran } \text{atts}') \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup \text{cls}' \]

The healthiness conditions **OO7–OO12** can be replaced by the following condition expressed in terms of the identity of our theory; in Law **OO7–OO12–OO13**-equivalence we prove this equivalence.

**OO13**

\[ P = P \ ; \ \Pi_{oo} \]

The set of predicates that satisfy all these healthiness conditions is our theory of object-orientation. It is a complete lattice, and its bottom element is

\[ \bot_{oo} \cong \text{oo}(\bot) \]
where \(OO\) is the functional composition of \(OO_1 \circ \cdots \circ OO_{12}\). The identity of our theory, denoted by \(I\) \(OO\), is the result of the application of \(OO\) to the relational identity \(I\).

\[
I \circ OO = OO_1(I)
\]

Our healthiness conditions are idempotent, and the UTP constructs are closed under these conditions. Below, we present some laws that are valid confirming this result; the rules left out are similar.

Provided \(P\), \(Q\), and \(F(X)\) are \(OO_1\)

\[
\begin{align*}
OO_1 \circ OO_1 &= OO_1 & \text{[\(OO_1\)-idempotent]} \\
P \land Q &= OO_1(P \land Q) & \text{[\(OO_1\)-\(\land\)-closure]} \\
P \lor Q &= OO_1(P \lor Q) & \text{[\(OO_1\)-\(\lor\)-closure]} \\
Q \circ b \cdot Q &= OO_1(P \circ b \cdot Q) & \text{[\(OO_1\)-\(\circ\)-closure]} \\
F \cdot Q &= OO_1(F \cdot Q) & \text{[\(OO_1\)-\(\cdot\)-closure]} \\
\mu X \cdot F(X) &= OO_1(\mu X \cdot F(X)) & \text{[\(OO_1\)-\(\mu\)-closure]}
\end{align*}
\]

Similar results are proved for the other healthiness conditions in much the same way. The order of application of \(OO_1\) and \(OO_2\) is irrelevant. This result, the forthcoming laws related to commutativity, and similar results for \(OO_7 \circ \cdots \circ OO_{12}\) prove that there is a subset of relations involving \(cls\), \(sc\), and \(atts\) where all healthiness conditions \(OO_1\) and \(OO_{12}\) are satisfied. As already said, this subset is our theory of object-orientation, and we have already shown that the application of conjunction, disjunction, sequence and recursion are closed in this subset.

\[
\begin{align*}
OO_1 \circ OO_2 &= OO_2 \circ OO_1 & \text{[\(OO_1\)-\(OO_2\)-commutativity]} \\
OO_1 \circ OO_3 &= OO_3 \circ OO_1 & \text{[\(OO_1\)-\(OO_3\)-commutativity]} \\
OO_1 \circ OO_4 &= OO_4 \circ OO_1 & \text{[\(OO_1\)-\(OO_4\)-commutativity]} \\
OO_1 \circ OO_5 &= OO_5 \circ OO_1 & \text{[\(OO_1\)-\(OO_5\)-commutativity]} \\
OO_1 \circ OO_6 &= OO_6 \circ OO_1 & \text{[\(OO_1\)-\(OO_6\)-commutativity]} \\
OO_2 \circ OO_3 &= OO_3 \circ OO_2 & \text{[\(OO_2\)-\(OO_3\)-commutativity]} \\
OO_2 \circ OO_4 &= OO_4 \circ OO_2 & \text{[\(OO_2\)-\(OO_4\)-commutativity]}
\end{align*}
\]

The composition of \(OO_7 \circ \cdots \circ OO_{12}\) can be replaced by the application of \(OO_{13}\).

\[
OO_{13} = OO_7 \circ \cdots \circ OO_{12} & \text{[\(OO_7 \circ \cdots \circ OO_{12}\)-equivalence]}
\]

### 1.4 Declarations

In this section we provide the meaning for class, state component, and operation declarations with some examples. The general form of the declarations is shown in Table 1.1. To each design that we define, we apply the composition of the healthiness conditions \(OO\); that is, \(OO_1 \circ \cdots \circ OO_{12}\), to guarantee that the initial values of the variables are valid and the restrictions over \(cls', sc',\) and \(atts'\) hold.
### 1.4.1 Classes

Our aim is to add each feature of object-orientation in isolation. In this direction, a class declaration introduces a new type, with an empty set of state components and operations.

**Definition 1.4.1 (Class introduction)** The declaration of a class is defined as shown below.

\[
\text{class } A \text{ extends } B = \begin{cases} 
\text{where } (w = \text{ina}(\text{class } A \text{ extends } B) \setminus \{\text{cls}, \text{sc}, \text{atts}\}) \\
\text{OO} \left\langle \begin{array}{l}
A \notin \text{Type} \land B \in \text{cls} \\
\vdash \\
(\text{cls}' = \text{cls} \cup \{A\}) \\
\land (\text{sc}' = \text{sc} \cup \{A \rightarrow B\}) \\
\land (\text{atts}' = \text{atts} \cup \{A \rightarrow \emptyset\}) \\
\land (w' = w)
\end{array} \right. 
\end{cases}
\]

The design introduces a record of class \( A \) in \( \text{cls} \) and maps it in the relation \( \text{sc} \) to \( B \) as its immediate superclass. Only new names are allowed (\( A \notin \text{Type} \)), and class \( B \) needs to have been previously declared (\( B \in \text{cls} \)). An entry for \( A \) in \( \text{atts} \) is mapped to the empty set. No other observational variable \( w \) is modified. As explained before, in the UTP, \( \text{ina}(\text{class } A \text{ extends } B) \) is the input alphabet of the program **class** \( A \text{ extends } B \).

The postcondition of the design establishes new final values for the observational variables of our theory; these values satisfy the properties required by the healthiness conditions **OO7-OO12**. More specifically, we do not remove **Object** from \( \text{cls}' \) (**OO7** is satisfied); the domain of \( \text{sc} \) is extended with a new class \( A \) in \( \text{cls} \) associated with \( B \), and the precondition guarantees that \( B \) is already in \( \text{cls}' \) (**OO8** and **OO9** are satisfied); the domain of \( \text{atts} \) is extended to include the new class \( A \) introduced in \( \text{cls} \) (**OO10** is satisfied); and the state component name is new and has a valid type (**OO11** and **OO12** are satisfied). For a simple declaration **class** \( A \), we have the obvious meaning.

\[
\text{class } A = \text{class } A \text{ extends } \text{Object}
\]

**Example 1.4.1 (Class declaration)** For our simple banking application, we declare the classes **Account**, which depicts an account of a bank, **BAccount**, an extension of **Account** to hold bonus information, **Contact**, to hold traditional contact information, and **EContact**, an extension of **Contact** to hold electronic contact information. The meaning of the sequence
of declarations of these classes is the sequence below.

```plaintext
class Account
class BAccount extends Account
class Contact
class EContact extends Contact

OO =
(Account \notin \{B, Z\} \cup cls
\land Object \in cls
\land (cls' = cls \cup \{Account\})
\land (sc' = sc \cup \{Account \mapsto Object\})
\land (atts' = atts \cup \{Account \mapsto \emptyset\})
(BAccount \notin \{B, Z\} \cup cls
\land Account \in cls
\land (cls' = cls \cup \{BAccount\})
\land (sc' = sc \cup \{BAccount \mapsto Account\})
\land (atts' = atts \cup \{BAccount \mapsto \emptyset\})
(Contact \notin \{B, Z\} \cup cls
\land Object \in cls
\land (cls' = cls \cup \{Contact\})
\land (sc' = sc \cup \{Contact \mapsto Object\})
\land (atts' = atts \cup \{Contact \mapsto \emptyset\})
(EContact \notin \{B, Z\} \cup cls
\land Contact \in cls
\land (cls' = cls \cup \{EContact\})
\land (sc' = sc \cup \{EContact \mapsto Contact\})
\land (atts' = atts \cup \{EContact \mapsto \emptyset\})
```

The meaning of sequence in our theory is the same as that in the UTP. If we take `cls`, `sc`, and `atts` to be `cls_0`, `sc_0`, and `atts_0`, respectively, the sequence above specifies the following values for `cls'`, `sc'` and `atts'`.

```plaintext
cls' = \{Object, Account, BAccount, Contact, EContact\}
\land (cls' = cls \cup \{Account \mapsto Object\})
\land (sc' = sc \cup \{Account \mapsto Object\})
\land (atts' = atts \cup \{Account \mapsto \emptyset\})
\land (cls' = cls \cup \{BAccount \mapsto Account\})
\land (sc' = sc \cup \{BAccount \mapsto Account\})
\land (atts' = atts \cup \{BAccount \mapsto \emptyset\})
\land (cls' = cls \cup \{Contact \mapsto Object\})
\land (sc' = sc \cup \{Contact \mapsto Object\})
\land (atts' = atts \cup \{Contact \mapsto \emptyset\})
\land (cls' = cls \cup \{EContact \mapsto Contact\})
\land (sc' = sc \cup \{EContact \mapsto Contact\})
\land (atts' = atts \cup \{EContact \mapsto Contact\})
\land Object \mapsto \emptyset,
\land Account \mapsto \emptyset,
\land BAccount \mapsto \emptyset,
\land Contact \mapsto \emptyset,
\land EContact \mapsto \emptyset
```
1.4.2 State Components

We introduce state components in \textit{atts} for those classes already in \textit{cls}.

\textbf{Definition 1.4.2 (State Component introduction)} To introduce a state component \(x\) of type \(T\) in class \(A\) we can use the construct defined below.

\[
\text{att } A \ x : T \triangleq \begin{cases} 
A \in \text{cls} \\
\land x \notin \text{dom}\ C(\text{atts}, \text{cls}) \\
\land T \in \text{Type} \\
\models (\text{atts}' = \text{atts} \oplus \{A \mapsto (\text{atts}(A) \cup \{x \mapsto T\})\}) \\
\land (w' = w)
\end{cases}
\]

where \(w = \text{ina}(\text{att } A \ x : T) \setminus \{\text{atts}\}\)
\text{and } \(C(\text{amap}, \text{cset}) = \bigcup \{N : \text{cset} \bullet \text{amap}(N)\}\)
\text{and } \text{amap} \text{ is a state component mapping, and } \text{cset} \text{ is class set}

If we try to declare a state component of a class that has not been declared previously, with a name that was already used, or of a type that is not primitive or present in \textit{cls}, then the declaration aborts. The set \(C\) (defined above as \(C(\text{amap}, \text{cset}) = \bigcup \{N : \text{cset} \bullet \text{amap}(N)\}\)) is a useful abbreviation for a mapping of all state components of any class to their corresponding types, calculated from a state component mapping as defined for \textit{atts}, and a class set as \textit{cls}. Our healthiness conditions \textit{OO7–OO12} are guaranteed by the design not changing the variables \textit{cls} and \textit{sc} nor the domain of \textit{atts}.

We can declare several state components simultaneously, with the obvious meaning.

\[
\text{att } A \ x_1 : T_1 ; x_2 : T_2 ; \ldots = \text{att } A \ x_1 : T_1 ; \text{att } A \ x_2 : T_2 ; \ldots \\
\text{att } A \ x_1 : T_1 ; B \ x_2 : T_2 ; \ldots = \text{att } A \ x_1 : T_1 ; \text{att } B \ x_2 : T_2 ; \ldots
\]

Our notation allows interleaving concerning with the order of class, state component, and operation declaration. For example, the sequence below is allowed.

\[
\text{class } A ; \text{att } A \ x : \mathbb{Z} ; \text{class } B \text{ extends } A ; \text{att } A \ y : \mathbb{B} ; \text{att } B \ z : A
\]

In this case, the state component \(y\) of class \(A\) is declared after the declaration of class \(B\). In fact, if we have recursive classes, the required order of the declaration is different from that adopted in languages where classes are blocks. For example, if a class \(A\) has a state component \(x\) whose type is a subclass \(B\) of \(A\), then the following order of declaration is required.

\[
\text{class } A ; \text{class } B \text{ extends } A ; \text{att } A \ x : B
\]

Transforming the class-based declarations of an object-oriented language into an appropriate sequence of class and state component declarations is a simple task. For operations, similar considerations apply; mutual recursion, however, is further discussed in Section 1.7.4.
Example 1.4.2 (State Component declaration) This example adds some state components to the classes of Example 1.4.1.

\[
\begin{align*}
\text{atts} & \text{ Account number : } Z, \text{ balance : } Z, \text{ contact : Contact} = \\
\text{atts} & \text{ BAccount bonus : } Z \\
\text{atts} & \text{ Contact phone : } Z \\
\text{atts} & \text{ EContact icq : } Z
\end{align*}
\]

\[
\begin{align*}
\forall \text{atts} \in \text{cls} & \left( \begin{array}{l}
\text{number} \notin \text{dom}(\text{atts, cls}) \\
\text{Z} \in \{B, Z\} \cup \text{cls}
\end{array} \right) ; \\
\forall \text{atts} \in \text{cls} & \left( \begin{array}{l}
\text{balance} \notin \text{dom}(\text{atts, cls}) \\
\text{Z} \in \{B, Z\} \cup \text{cls}
\end{array} \right) ; \\
\forall \text{atts} \in \text{cls} & \left( \begin{array}{l}
\text{contact} \notin \text{dom}(\text{atts, cls}) \\
\text{Contact} \in \{B, Z\} \cup \text{cls}
\end{array} \right) ; \\
\forall \text{atts} \in \text{cls} & \left( \begin{array}{l}
\text{bonus} \notin \text{dom}(\text{atts, cls}) \\
\text{Z} \in \{B, Z\} \cup \text{cls}
\end{array} \right) ; \\
\forall \text{atts} \in \text{cls} & \left( \begin{array}{l}
\text{phone} \notin \text{dom}(\text{atts, cls}) \\
\text{Z} \in \{B, Z\} \cup \text{cls}
\end{array} \right) ; \\
\forall \text{atts} \in \text{cls} & \left( \begin{array}{l}
\text{icq} \notin \text{dom}(\text{atts, cls}) \\
\text{Z} \in \{B, Z\} \cup \text{cls}
\end{array} \right)
\end{align*}
\]

We use the definition of state component declaration for each element of the sequence, starting with the state component number, and ending with icq. If we suppose that the declaration above comes after the class declarations of Example 1.4.1, the expected final value of \text{atts} is as follows.

\[
\text{atts} = \begin{cases} 
\text{Object} & : \emptyset, \\
\text{Account} & : \{\text{number} : Z, \text{balance} : Z, \text{contact} : \text{Contact}\}, \\
\text{BAccount} & : \{\text{bonus} : Z\}, \\
\text{Contact} & : \{\text{phone} : Z\}, \\
\text{EContact} & : \{\text{icq} : Z\}
\end{cases}
\]

For a given class \(N\), we define \(U(\text{amap}, \text{smap}, N)\) to be a mapping that records all the state components of \(N\), including those declared in its superclasses, considering a state component mapping \(\text{amap}\), and a subclass relation \(\text{smap}\) defined with the same types of \text{atts}, and \text{sc}, respectively. We define this closure as:

\[
U(\text{amap}, \text{smap}, N) \triangleq \bigcup(\text{amap}\{\text{smap}^*(\{N\})\})
\]
In words, $U(amap, smap, N)$ contains all the state component declarations of all classes related to $N$ by the reflexive and transitive closure of the superclass relation, considering the current state components in $atts$ and subclass relation $sc$. This function is useful to define object creation and also to check if an instance of an object is well-defined.

### 1.4.3 Operations

For an operation declaration to succeed, the class to which it is associated must have been introduced before, and all formal parameters, passed by value ($val$), result ($res$) or value-result ($vres$), must have primitive types or those introduced in $cls$. The result depends on whether the operation is being declared for the first time or not. If it is ($m \notin \alpha(meth \ A \ m = pds \bullet p)$), then the definition below applies. The new name $m$ is introduced in the alphabet using a variable declaration.

**Definition 1.4.3 (New operation introduction)** For new operations, the declaration is defined as follows.

\[
\begin{align*}
\text{meth } A \ m (Pdsatt \ p) & \equiv \\
\text{var } m : & \\
& \left( A \in cls \\
& \quad \forall t \in \text{types}(pds) \cdot t \in \text{Type} \\
& \quad \vdash (m' = vres \ self: Object ; pds \circledast \ p \gets \text{abort}) \\
& \quad \land (w' = w) \\
\text{provided} & \quad m \notin \alpha(\text{meth } A \ m = (pds \bullet p)) \\
\text{where} & \quad (w = \text{in} \alpha(\text{meth } A \ m = (pds \bullet p))
\end{align*}
\]

The value of $m'$ is the text of a parametrised program. Operations are higher-order, parametrised program-valued variables, much in the same way as in the theory of higher-order procedures and parameters of the UTP. The parameters of $m'$ are those in $pds$ and an extra parameter $self$ to represent the target of a call; its type is $Object$. Just as with $var \ x$, which introduces the new alphabetic variables $x$ and $x'$, for $meth \ A \ m$, we introduce the new alphabetic variables $m$ and $m'$, and use a design to define the value of $m'$.

For the case of a redefinition of an operation $m$ ($m \in \alpha(meth \ A \ m = pds \bullet p)$) we have another definition.

**Definition 1.4.4 (Operation redefinition)** If the operation name declared is not new, the corresponding definition is the following.

\[
\begin{align*}
\text{meth } A \ m = (pds @ p) & \equiv \\
\text{OO} & \\
& \left( A \in cls \\
& \quad \exists q \bullet m = pds_e \bullet q \\
& \quad \vdash \exists q \bullet \\
& \quad (m = pds_e \bullet q) \\
& \quad \land (m' = pds_e \bullet \text{join}(A, p, q)) \\
& \quad \land (w' = w) \\
\text{provided} & \quad m \in \alpha(\text{meth } A \ m = (pds \bullet p)) \\
\text{where} & \quad pds_e = vres \ self: Object ; pds @ p
\end{align*}
\]
If the operation declaration is a redefinition, the operation signature must be exactly the same as that of the existing operation, and a new conditional is built to take into account the class hierarchy. The definition of the syntactic function \( \text{join} \) deals with redefinition of \( m \) both in superclasses and in subclasses of the class where the original definition is placed. The use of \( \text{join} \) allows us to introduce the operation values as (parametrised) programs in a form where dynamic binding is already resolved, as in algebraic operations [3, 4] and in the weakest precondition approach [5]. As already said, the special variable \( \text{self} \) denotes the target of the operation call. All references to state components in operation bodies must be prefixed with \( \text{self} \); variables without this prefix are formal parameters or local variables.

We give the meaning of a parametrised program as a function from a value or a variable name to a program (or predicate). We consider each of the mechanisms of parameter passing individually; the definitions reflect the standard way of implementing them.

For a value parameter, the semantics is a higher-order function that takes the value of the argument and gives the program that declares the formal parameter as a local variable and initializes it with the argument.

\[
\text{val } v : T \bullet p \quad \overset{\Delta}{=} \text{(\lambda w : T \bullet (var } v : T \quad ; \quad v := w \quad ; \quad p \quad ; \quad end \quad v))}
\]

A function that models a parametrised program with a parameter passed by result takes as argument the name of a variable: an element of the syntactic category \( N \). This is the argument in an operation call.

\[
\text{res } v : T \bullet p \quad \overset{\Delta}{=} \text{(\lambda w : N \bullet (var } v : T \quad ; \quad p \quad ; \quad w := v \quad ; \quad end \quad v))}
\]

In this case, the local variable corresponding to the formal parameter is not initialized; its value is assigned to the argument.

For a value-result parameter, the definition is as expected: the local variable is initialized and then assigned to the argument in the end.

\[
\text{vres } v : T \bullet p \quad \overset{\Delta}{=} \text{(\lambda w : N \bullet (var } v : T \quad ; \quad v := w \quad ; \quad p \quad ; \quad w := v \quad ; \quad end \quad v))}
\]

The parameter of the function is again a program variable.

Lambda-reduction is extended to cope with variable parameters: elements of the syntactic category \( N \). It is an abstraction over four arguments: a variable, the corresponding type variable, and their dashed counterparts. A similar semantics for parametrisation was presented in [6].

\[
(\lambda x : N \bullet p)(y) \quad \overset{\Delta}{=} \text{p[y, y', y_{0}/x, x', x_{0}, x_{0}']}
\]
Example 1.4.3 (Operation declaration) In this example we show the semantics of operation declarations, given that \(cls\) is the one defined in Example 1.4.1, and \(atts\) is that defined in Example 1.4.2. We introduce an operation \(credit\) for \(Account\), and we redefine it for class \(BAccount\) to also increase the value of a bonus variable.

\[
\begin{align*}
\text{meth} & \text{Account} \ \text{credit} = (\text{val } x : \mathbb{Z} \bullet \text{self}.balance := \text{self}.balance + x) ; \\
\text{meth} & \text{BAccount} \ \text{credit} \\
& = (\text{val } x : \mathbb{Z} \bullet \text{self}.bonus := \text{self}.bonus + x \; \text{self}.balance := \text{self}.balance + x)
\end{align*}
\]

We observe that, in the body of the redefinition of \(credit\) for \(BAccount\), we have a repetition of the code in the body of \(credit\) as defined for \(Account\). In a programming language, this is likely to be written as \(\text{super} \ . \text{credit} (x)\) or using some other similar notation that avoids code repetition. As we explained in Section 1.2, however, semantically, these constructs can be removed using a copy rule. For this reason, we do not consider this issue here. The meaning of the two operation declarations is given by the sequence below.

\[
\begin{align*}
\var & \text{credit} ; \\
\begin{array}{l}
\text{Account} \in \text{cls} \Rightarrow \\
\exists q \bullet (\text{credit} = (\text{vres self: Object ; val } x: \text{int} @ q)) \\
\end{array} \\
\begin{array}{l}
\text{BAccount} \in \text{cls} \\
\exists q \bullet (\text{credit} = (\text{vres self: Object ; val } x: \text{int} @ q)) \\
(\text{credit}' = (\text{vres self: Object ; val } x: \text{int} @ q) \\
\text{join}(\text{BAccount, (self.bonus := \cdots)}, q)) \\
\end{array} \\
\end{align*}
\]

If we eliminate the sequential composition, the value of the variable \(q\) existentially quantified in the second design is determined to be the body of \(credit\) as defined in the first design. With that, we can calculate the result of \(\text{join}\). The final value of \(credit\) is of the following form.

\[
\begin{align*}
\var & \text{self: Object ; val } x: \text{int} @ \\
(\text{self.bonus := self.bonus + 1}; \cdots) \\
\begin{array}{l}
\text{<self is BAccount >} \\
(\text{self.balance := self.balance + x}) \\
\text{<self is Account >} \\
\text{abort}
\end{array}
\end{align*}
\]

The conditional generated by \(\text{join}\) selects the appropriate command depending on the type of \(\text{self}\). This is the expected behaviour in the presence of dynamic binding. \(\square\)

1.5 Variables

In [7], type information is not explicitly recorded for variables. In an object-oriented language, where types play a central role, this is not appropriate. In our theory, the values
of the variables are pairs, whose first element is the (runtime) type of the variable, and the second is the value itself. We define the construct \texttt{var} \( x : T \) for typed declaration of variables, where \( T \) is the static type of the variable \( x \).

**Definition 1.5.1 (Variable declaration)**

\[
\texttt{var} \ x : T \ \cong \ \texttt{OO} \left( \{ T \in \text{Type} \} \land \var x, x : (\text{true} \land (x'_t = T) \land x' \in \mathcal{V}(T) \land (w' = w)) \right)
\]

provided \( x \notin \text{ina}(\texttt{var} \ x : T) \) and \( w = \text{ina}(\texttt{var} \ x : T) \)

We use the existing \texttt{var} construct to introduce both \( x \) and \( x_t \) in the alphabet. In the definition, we assert that \( T \) is a valid type, if it is invalid the sequential composition aborts. Otherwise, the type \( x_t \) of \( x \) is defined to be \( T \), and an arbitrary element of \( T \) is chosen as its initial value. The other variables remain unchanged. In assignments to \( x \), its value, which is a pair \((x_t, x_v)\), may change, but \( x_t \) does not.

To complete this definition, we need to define the set of elements \( \mathcal{V} \) of a type \( C \). These are pairs in which the first element is a subclass \( A \) of \( C \), possibly \( C \) itself, and the second element is either the special value \texttt{null} or a mapping from each of the state components of \( A \) to a value, in the case of classes. For primitives types the second value is a primitive value, such as \texttt{1} for integer, or \texttt{true} for booleans. A formal definition is a function that takes \( sc \) and \( atts \) as parameters; a similar function is specified in [5]. As with \( \texttt{var} \ x \), our typed declaration is a non-homogeneous relation: the alphabet of \( \texttt{var} \ x : T \) does not include \( x \) or \( x_t \).

The definition of \( \texttt{end} \ x : T \) (the construct used to finalise the scope of \( x \)) is similar to that in the UTP for \( \texttt{end} \ x \). There are no concerns with type at the end of the scope of a variable, but we need to close the scope of both \( x \) and \( x_t \).

**Definition 1.5.2 (Variable removal)**

\[
\texttt{end} \ x : T \ \cong \ \texttt{OO}(\texttt{end} \ x, x_t)
\]

The discussion about the structure of values is extremely important to the definition of value of an object and the correctness of assignments and operation calls. We have made explicit the representation of values in order to handle the concepts of object-orientation.

### 1.6 Expressions

In this section, we specify well-definedness rules for expressions, and the semantics of object creation, type test, type cast, and state component accesses.

#### 1.6.1 Well-definedness

Our theory includes new forms of expression \( e \) characterised by the BNF-like definition in Table 1.2. In this syntax, \( v \) is a primitive or object value. The expressions \( le \), left expressions, are ordinary variable names or the special variable named \texttt{self}, followed by a (possibly empty) sequence of dot-separated names. The expression \( \texttt{new} \ N \) is an object.
Table 1.2: BNF for object-oriented expressions.

creation, \( e \texttt{ is} \ N \) is a type test, and \((N)e\) is a type cast. There is also a group of built-in operations over expressions, like, for instance, arithmetic and relational operators, denoted by \(f(e)\).

An expression \(e\) is represented by a pair: \((e_t,e_v)\), where the first element of the pair \(e_t\) is the type of \(e\) and the second element \(e_v\) is its value. The construct \texttt{null} stands for a family of values, one for each class. The type held by \(e_t\) in this case is inferred from the context. For instance, in an assignment \(x := \texttt{null}\), we have that \(e_t = x_t\); this means that the runtime type of \texttt{null} is the declared type of \(x\).

The well-definedness of expressions, and commands, is specified by a function named \(D\). If an expression has a primitive value, it is well-defined if the value belong to the set of possible values of the type. For objects, we must check if the type belongs to \(\texttt{cls}\), and if the value belongs to the set of values of type \(T, \mathcal{V}(T)\). For primitive types the test is simpler.

**Primitive values**
\[
\begin{align*}
D(\mathbb{B}, v) &= v \in \mathbb{B} \\
D(\mathbb{Z}, v) &= v \in \mathbb{Z}
\end{align*}
\]

**Objects**
\[
\begin{align*}
D(T, \texttt{null}) &= T \in \texttt{cls} \\
D(T, v) &= T \in \texttt{cls} \land (T, v) \in \mathcal{V}(T)
\end{align*}
\]

Variables are well-defined if their types are either primitive or present in \(\texttt{cls}\). If a variable has the special name \(\texttt{self}\), it cannot be of primitive type.

**Variables**
\[
\begin{align*}
D(x) &= x_t \in \texttt{Type} \\
D(\texttt{self}) &= \texttt{self}_t \in \texttt{Type}
\end{align*}
\]

a state component access \(le.x\) is valid only if \(le\) is well-defined, the type of \(le\) is not primitive, the value of \(le\) is different from \texttt{null}, and \(x\) is in the domain of the value of \(le\).

**State Component Accesses**
\[
D(le.x) = D(le) \land le_t \in \texttt{cls} \land (le_v \neq \texttt{null}) \land x \in \text{dom \, le_v}
\]

A \texttt{new} \(N\) declaration is valid only if the class \(N\) is recorded in \(\texttt{cls}\). A type test \(e \texttt{ is} \ N\) or casting \((N)e\) can be done only if \(e\) is a well-defined expression and \(N\) is not primitive. For a type cast, the expression has to be of a valid subtype of \(N\).

**Typing**
\[
\begin{align*}
D(\texttt{new} \ N) &= N \in \texttt{cls} \\
D(e \texttt{ is} \ N) &= D(e) \land N \in \texttt{cls} \\
D((N)e) &= D(e) \land N \in \texttt{cls} \land e_t \leq N
\end{align*}
\]

The well-definedness restrictions for built-in operations for primitive types, \(f(e)\), are defined individually and are very similar. We show the example of the remainder of a division
operator, “mod”.

**Remainder**
\[ D(x \mod y) = D(x) \land D(y) \land x = \mathbb{Z} \land y = \mathbb{Z} \land y \neq 0 \]

In Section 1.7.1, we use the function \( D \) on expressions to define well-definedness rules for commands.

### 1.6.2 Object Creation

An object value is a pair \((\text{type}, \text{value})\): the **type** is a class name and the **value** is a mapping from names to state component values. Using \( sc \) and \( atts \) to recover state components and inheritance information, we provide a definition for \texttt{new} as follows.

\[
\texttt{new} N \triangleq \left( \begin{array}{l}
  x : \text{dom map}; t : \text{Type}; v : \{ T : \text{Type}; i : T \bullet i \} | \\
  N, \left( \begin{array}{l}
    ( (\text{map}(x) = \mathbb{B}) \land (t = \mathbb{B}) \land (v = \text{false}) ) \\
    \lor ( (\text{map}(x) = \mathbb{Z}) \land (t = \mathbb{Z}) \land (v = 0)) \\
    \lor ( \exists T : \text{cls} \bullet (\text{map}(x) = T) \land (t = T) \land (v = \text{null})) \bullet \\
    x \mapsto (t, v)
  \end{array} \right)
\end{array} \right)
\]

where \( \text{map} = \mathcal{U}(atts, sc, N) \)

This definition says that the value of a newly created object is a mapping from state component names \( x \) to values \((t, v)\) that associate all boolean state components to \texttt{false}, all integer state components to \texttt{0}, and all class-typed state components to \texttt{null}. For example, the value of \texttt{new BAccount}, considering the values of \( sc \) and \( atts \) obtained after the declarations of Examples 1.4.1 and 1.4.2, is

\[
(\texttt{new BAccount}, \\
\{ \texttt{number} \mapsto (\mathbb{Z}, 0), \texttt{balance} \mapsto (\mathbb{Z}, 0), \texttt{contact} \mapsto (\texttt{Contact, null}), \texttt{bonus} \mapsto (\mathbb{Z}, 0) \})
\]

In this example, all state components from class \textit{Account} (\texttt{number, balance, contact}), as well as those from \texttt{BAccount} (\texttt{bonus}), are included.

### 1.6.3 Type Test

The expression \( e \texttt{ is } N \) is a boolean that indicates whether the value of \( e \) belongs to the class \( N \) or to one of its subclasses. The result yielded by such an expression is

\[
e \texttt{ is } N \triangleq (\mathbb{B}, e_t \leq N)
\]

For example,

\[
(\texttt{new BAccount}) \texttt{ is Account} \\
= (\texttt{BAccount, ...}) \texttt{ is Account} \\
= (\mathbb{B}, \texttt{BAccount} \leq \texttt{Account}) \\
= (\mathbb{B}, \texttt{true})
\]

This is justified by the definitions of \texttt{new}, type test, and \( \leq \), if we assume that \( \texttt{cls} \) and \( sc \) are as defined in Example 1.4.1.
1.6.4 Type Cast

The result of a cast \( (N)e \) is the expression \( e \) itself, if the casting is well defined. Since we are only defining the meaning of well-defined expressions, our specification is surprisingly trivial.

\[
(N)e \equiv e
\]

For example, provided that \( B\text{Account} \preceq \text{Account} \)

\[
(\text{Account}) \text{new } B\text{Account} \\
= (\text{Account})(B\text{Account}, \{\ldots\}) \\
= (B\text{Account}, \{\ldots\})
\]

Well-definedness is checked in the semantics of assignments and conditionals.

1.6.5 State Component Access

A state component access \( le.x \) recovers from the object value mapping \( (le_v) \) the state component named \( x \).

\[
le.x \equiv le_v(x)
\]

Again, we have a very simple definition, because we are considering only well-defined state component accesses. For example, suppose that we have an instance of \( B\text{Account} \) such as in the following program.

```plaintext
var x : BAccount ;
x := new BAccount ;
```

The result of the expression \( x\text{.bonus} \) is given by

\[
x\text{.bonus} \\
= (x_l, x_v)\text{.bonus} \\
= (B\text{Account}, \{\ldots, \text{bonus} \mapsto (Z, 0)\})\text{.bonus} \\
= \{\ldots, \text{bonus} \mapsto (Z, 0)\}(\text{bonus}) \\
= (Z, 0)
\]

As expected, we select the value associated to \( \text{bonus} \) in the mapping of state component values for \( x \). If we have a composite name like \( le.x.y \), we successively apply the lookup \((le_v(x)).y\) to select the expected value.

1.7 Commands

Our theory includes assignments \( le := e \) of a value \( e \) to a left expression \( le \), and operation calls \( le.m(a) \) with target \( le \) and list of arguments \( a \). Moreover, since expressions have changed, we need to consider well-definedness for some commands. We also introduce mutual recursion. Sequence remains unchanged. The syntax is described in Table 1.3.
\[ c ::= \text{le} := e | \Pi | \text{var} x : T | \text{end} x : T | e_1 \triangleleft e \triangleright e_2 | e_1 ; e_2 | \mu X \bullet F(X) | \text{le.m}(e) \]

Table 1.3: BNF for object-oriented commands.

### 1.7.1 Well-definedness

In this section, we specify well-definedness for assignments, conditionals, and operation calls. We consider two forms of assignment: assignments to variables, and assignments to object state components. An assignment of an expression \( e \) to a variable \( x \) is considered well-defined if \( x \) is well-defined, \( e \) is well-defined, and the type of \( e \) is a subtype of the type \( x_t \) of \( x \).

**Assignment to variables**
\[ D(x := e) = D(x) \land D(e) \land e_t \leq x_t \]

For an assignment of an expression \( e \) to a state component \( x \) of \( \text{le} \) to be well-defined, the expression \( \text{le}.x \) must be well-defined, \( e \) must be well-defined, and the type of \( e \) must be a subtype of \( U(\text{atts}, sc, le_t)(x) \), the type of the state component \( x \) in the class \( le_t \) (defined on page 14).

**Assignment to state components**
\[ D(\text{le}.x := e) = D(\text{le}.x) \land D(e) \land e_t \leq U(\text{atts}, sc, le_t)(x) \]

For a conditional to be well-defined, the condition must be well-defined and yield a boolean value.

**Conditional**
\[ D(P \triangleleft e \triangleright Q) = D(e) \land (e_t = \mathbb{B}) \]

The definition of well-definedness for operation calls is the most extensive. An operation call of the form \( \text{le}.m(a) \) is valid if:
- The left-hand expression \( \text{le} \) is well-defined.
- The value of \( \text{le} \) is not \text{null}.
- The operation \( m \) is defined for the type of \( \text{le} \).
- To avoid aliasing, \( \text{le} \) is not passed as an argument and is not involved in any argument. For further details about this restriction, see [5].
- The types of the arguments in the list \( a \) must be compatible with the formal parameters list of \( m \).

We present well-definedness definitions according to the parameter passing mechanism. Starting with value parameters, we have the definition below.

**Operation call**
\[ D(\text{le}.m(e)) \equiv D(\text{le}) \land (\text{le}_v \neq \text{null}) \land \text{compatible}(\text{le}, m) \land e_t \leq T \]

**provided** \( \exists p \bullet m = (\text{val} x : T @ p) \)

**where** \( \text{compatible}(\text{le}, m) = \exists \text{pds}, p \bullet m = (\text{pds} @ p) \land \text{le}_t \in \text{scan}(p) \)

**with** \( \text{scan}(\bot_{OO}) = \emptyset \)

**and** \( \text{scan}(\text{p}_l \triangleleft \text{self is} \triangleleft \text{p}_r) = \{ B : \text{cls} | B \leq A \} \cup \text{scan}(\text{p}_r) \)
The *scan* function yields the set of class names for which the operation \( m \) may have a definition different from \( \bot_{\text{OO}} \). For result and value-result parameters, we use the function \( \text{sdisjoint} \) [5], which verifies if \( le \) is involved in any of the arguments.

\[
\begin{align*}
\mathcal{D}(\text{le}.m(y)) & \triangleq \mathcal{D}(\text{le}) \land (\text{le}_v \neq \text{null}) \land \text{compatible}(\text{le}, m) \land \text{sdisjoint}(\text{le}, y) \land T \leq y_t \\
& \text{provided } \exists p \cdot m = (\text{res} \ x : T @ p) \\
\mathcal{D}(\text{le}.m(z)) & \triangleq \mathcal{D}(\text{le}) \land (\text{le}_v \neq \text{null}) \land \text{compatible}(\text{le}, m) \land \text{sdisjoint}(\text{le}, z) \land (T = z_t) \\
& \text{provided } \exists p \cdot m = (\text{vres} \ c : T)
\end{align*}
\]

The definition for an operation call with multiple arguments is a straightforward extension of these definitions.

### 1.7.2 Assignments

Now we define assignments to variables, and assignments to state components of object variables. In our theory, we observe that modifying the value of operation variables, type variables \( x_t \), \( \text{cls} \), \( \text{sc} \), \( \text{atts} \), or \( \text{ok} \) is not allowed, in much the same way that assignments to \( \text{ok} \) are not allowed in the theory of designs as well.

If we establish the well-definedness of an assignment, we can update the value of the variable to be that of the expression.

\[
\begin{align*}
x := e & \triangleq \text{OO}(\mathcal{D}(x := e) \models (x' = e) \land (w' = w)) \\
\text{where } (w = \text{in}(x := e) \setminus \{x\})
\end{align*}
\]

For example, given a variable \( x \) of type \( \text{Account} \ ((x_t = \text{Account})) \), we can calculate the meaning of the assignment \( x := \text{new} \ \text{BAccount} \) as follows, provided that \( y \) is the list of undashed variables in the alphabet, other than \( x \), and that \( \text{cls} \), \( \text{sc} \), and \( \text{atts} \) are as in Examples 1.4.1 and 1.4.2.

\[
\begin{align*}
x := \text{new} \ \text{BAccount} \\
\text{=} \{ \text{ assignment } \} \\
\text{OO} \left( \begin{align*}
\mathcal{D}(x := (\text{BAccount}, \{\text{number} \mapsto \ldots\})) \\
\models \\
(x' = (\text{BAccount}, \{\text{number} \mapsto \ldots\})) \land (y' = y)
\end{align*} \right) \\
\text{=} \{ \mathcal{D} \text{ for variable assignments } \} \\
\text{OO} \left( \begin{align*}
\mathcal{D}(x) \land \mathcal{D}(\text{BAccount}, \{\text{number} \mapsto \ldots\}) \land \text{BAccount} \preceq \text{Account} \\
\models \\
(x' = (\text{BAccount}, \{\text{number} \mapsto \ldots\})) \land (y' = y)
\end{align*} \right) \\
\text{=} \{ \mathcal{D} \text{ for variables and instances, assumptions on } \text{cls} \text{ and subtyping } \} \\
\text{OO} \left( \begin{align*}
\text{Account} \in \{\mathbb{B}, \mathbb{Z}\} \\
\land \text{BAccount} \in \text{cls} \\
\text{\models (BAccount, \{number} \mapsto \ldots\}) \in \mathcal{V}(\text{BAccount}) \\
\models \\
(x' = (\text{BAccount}, \{\text{number} \mapsto \ldots\})) \land (y' = y)
\end{align*} \right) \\
\text{=} \{ \text{ assumptions on } \text{cls}, \text{ definition of } \mathcal{V} \} \\
\text{OO} (\text{true} \models (x' = (\text{BAccount}, \{\text{number} \mapsto \ldots\})) \land (y' = y))
\end{align*}
\]
To update a state component of an object-valued expression, we check the well-definedness of the assignment, and if it is valid, then we update the mapping that records the state component value, maintaining the left expression type unchanged.

\[
le.x := e \Rightarrow \text{OO (D(le.x := e) ⊨ (le' = (le_l, le_v ⊕ \{x ↦ e\})) ∧ (w' = w))}
\]

where \( (w = \text{in}(le.x := e) \setminus \alpha(le)) \)

We use \( \alpha(le) \) to denote a variable in the alphabet whose value is being inspected by the left-expression \( le \). If \( le \) is a variable, then \( \alpha(le) \) is the variable itself. On the other hand, for \( x, y \) and \( x.y.z \), for example, the variable is \( x \). The equality \( (le' = (le_l, le_v ⊕ \{x ↦ e\})) \) for the case in which \( le \) is itself a state component access \( y.z \), for instance, is an abbreviation for the equality \( (y' = (y, y_v ⊕ \{z ↦ y.z ⊕ \{x ↦ e\})).) \)

For example, given a variable \( x \) of type \( \text{Account} \) \( (x_i = \text{Account}) \), which has been initialised with \( \text{new BAccount} \) \( (x = (\text{BAccount}, \{\text{number} ↦ (\mathbb{Z}, 0), \ldots\})) \), we can describe the state component update \( x \cdot \text{number} := 1 \) as follows, provided that \( y \) is the list of undashed variables in the alphabet, other than \( x \), and cls, sc, atts are as in Examples 1.4.1 and 1.4.2.

\[
x \cdot \text{number} := 1 = \{ \text{assignment to state components} \}
\]

\[
\text{OO (D((\text{BAccount}, \{\text{number} ↦ (\mathbb{Z}, 0), \ldots\}) \cdot \text{number} := (\mathbb{Z}, 1)))}
\]

\[\text{D for state component assignments, mapping replacement} \}

\[
\text{OO (D((\text{BAccount}, \{\text{number} ↦ (\mathbb{Z}, 0), \ldots\}) \cdot \text{number} := (\mathbb{Z}, 1)))}
\]

\[\text{for state component accesses and primitive instances, definition of} \ U \}

\[
\text{OO (D((\text{BAccount}, \{\text{number} ↦ (\mathbb{Z}, 0), \ldots\}))}
\]

\[\text{∧ BAccount ∈ cls}
\]

\[\text{∧ \{number ↦ (Z, 0), \ldots\} ≠ null}
\]

\[\text{∧ number ∈ dom\{number ↦ (Z, 0), \ldots\}}
\]

\[\text{∧ Z ≤ Z}
\]

\[\text{∧ (x' = (BAccount, \{number ↦ (Z, 1), \ldots\})) ∧ (y' = y)}
\]

\[\text{D for object instance, set properties, assumptions on} \ cls, \ subtyping} \}

\[
\text{OO (BAccount ∈ cls ∧ (BAccount, \{number ↦ \ldots\}) ∈ \mathcal{V}(BAccount)}
\]

\[\text{definition of} \ \mathcal{V}, \ \text{set properties} \}

\[
\text{OO (true} \text{∀ (x' = (BAccount, \{number ↦ (Z, 1), \ldots\}) ∧ (y' = y)}
\]

If we had not initialised the variable \( x \), the assignment would not be well-defined and would abort.
1.7.3 Conditional

We need to redefine the conditional to consider the well-definedness of the condition.

\[ P \triangleleft e \triangleright Q \triangleq \text{OO}(D(P \triangleleft e \triangleright Q) \land ((e_v \land P) \lor (\neg e_v \land Q))) \]

For example, suppose we have the variables cls, sc, and atts as in the Examples 1.4.1 and 1.4.2. If we declare a variable \texttt{var self : Account}, and initialise it as \texttt{self := new BAccount}, the result of the conditional \( P \triangleleft \text{self is BAccount} \triangleright Q \), with arbitrary \( P \) and \( Q \), is \( P \), as shown below.

\[
\begin{align*}
\text{OO}(P \triangleleft \text{self is BAccount} \triangleright Q) &= \{ \text{conditional, type test} \} \\
&= \{ \text{type selection} \} \\
&= \text{D for conditional, value selection, assumptions on cls, subtyping} \\
&= \{ \text{D for type test, type selection} \} \\
&= \{ \text{D for self, assumptions on cls} \} \\
&= \text{OO}(P \triangleleft \text{self is BAccount} \triangleright Q) \land ((\text{true} \land P) \lor (\text{false} \land Q)) \\
&= \{ \text{assumptions on cls} \}
\end{align*}
\]

If the type test were false, then the branch selected would be \( Q \). Moreover, according to the well-definedness rules for the variable \texttt{self}, it cannot be an instance of a primitive type. If this were the case, the meaning of the conditional would be to abort. It may be the case that \( P \) is not well-defined; in this case, abortion arises from the definition of \( P \).

1.7.4 Recursion

Recursion is as the least fixed-point in the complete lattice of parametrised programs partially ordered by refinement. For each parameter declaration, the set of programs with those parameters is a complete lattice; refinement is defined pointwise [2].

**Definition 1.7.1 (Recursive Operation)** The general form for the declaration of a recursive operation \( m \) of class \( A \) is the following.

\[
\text{meth } A \ m = \mu X \bullet (pds \bullet F(X))
\]

For example, an operation to calculate factorials could be added to a class \( A \) as follows

\[
\text{meth } A \ m = \mu X \bullet (\text{val } n : \mathbb{Z} ; \text{res } r : \mathbb{Z} \bullet r := 1 \triangleleft n \leq 0 \triangleright r := n \times X(n - 1, r))
\]
This defines a recursive method using the least fixed-point operator $\mu$. The body of the recursion has a value parameter $n$ and a result parameter $r$. The body itself is a conditional that checks if the parameter $n$ is non-positive. If it is, then the recursion terminates with the result parameter being set to 1; otherwise, the result is set to $n \cdot X(n-1, r)$.

This is not in conflict with the expected form of an operation declaration $meth A m = (pds \cdot p)$, since, of course, the least fixed-point operator results in a parametrised program. In particular, the parameters are the same as those in the body of the recursion. For each parameter declaration, we take the fixed point in the lattice of parametrised programs with those parameters.

**Definition 1.7.2 (Mutually Recursive Operations)** The general form for the declaration of mutually recursive operations $m$ of class $A$, and $n$ of class $B$ is the following.

$$meth A m, B n = \mu X, Y \cdot (pds_m \cdot F(X, Y), pds_n \cdot G(X, Y))$$

Mutual recursion is easily addressed in our theory. In this case, since $m$ and $n$ are mutually recursive, they are defined together, even though they are operations of different classes. This follows the standard approach to the definition of mutually recursive procedures. The vector of programs $m, n$ is defined as the least fixed point of the function from vectors of predicates to vectors of predicates defined by the bodies of $m$ and $n$: $(pds_m \cdot F(X, Y))$ and $(pds_n \cdot G(X, Y))$. As an example, calling the operations $m$ or $n$ defined below, with a variable $a$ as the result parameter, leads to the assignment of 0 to $a$.

$$meth A m, B n = \mu X, Y \cdot \left( \begin{array}{l} val \ x : \mathbb{Z} ; \ res \ i : \mathbb{Z} \cdot i := x < (x = 0) \Rightarrow Y(-x, i), \\
res \ j : \mathbb{Z} \cdot X(y + 1, j) \triangleleft (x > 0) \Rightarrow X(y + 1, j) \end{array} \right)$$

Once the recursion is resolved and the fixed-point operators are eliminated, the description of a multiple operation declaration like $meth A m, B n = ((pds_m \cdot mb), (pds_n \cdot nb))$ is a trivial extension of the definition of simple operation declarations presented in Section 1.4. In many theories of object-orientation, mutual recursion is a difficulty. The complication is really attached to the fact that the mutually recursive operations may be declared in an independent way in separate classes. By splitting the block structure of a class into its basic semantic blocks, we trivially overcome this difficulty.

### 1.7.5 Operation Call

Since we have already solved the problem of dynamic binding when dealing with the semantics of operation declaration in Section 1.4.3, the semantics of operation call is just a call to the value of the operation. In other words, we have isolated the several aspects involved in an operation call, so that dynamic binding is captured in the definition of the value of the operation variable, which holds a parametrised program, and an operation call is given mainly by the copy rule.

$$le.m(args) \doteq OO((D(le.m(args)))) \perp ; (pds_e \cdot p)(le, args)$$

**where** $m = (pds_e @ p)$


Example 1.7.1 Suppose that \((sc = sc_0)\), \((cls = cls_0)\), and \((atts = atts_0)\), and after the declaration of classes, state components and operations in the Examples 1.4.1 and 1.4.2, we have the program fragment below.

\[
\begin{align*}
\text{var} & \quad a : \text{Account} \\
& \quad a := \text{new } \text{BAccount} \\
& \quad a.\text{credit}(10)
\end{align*}
\]

Due to dynamic binding, \(a.\text{credit}(10)\) must execute the body of the operation \(\text{credit}\) defined for the subclass \(\text{BAccount}\). As described in Section 1.4, the value of \(\text{credit}\) is a conditional over the special variable named \(\text{self}\). Below, we show how the program associated to the variable \(\text{credit}\) resolves the dynamic binding. The meaning of \(a.\text{credit}(10)\) is defined in terms of \(\text{credit}(a, 10)\), which we consider below.

\[
\begin{align*}
\text{credit}(a, 10) &= \{ \text{operation expansion} \} \\
&= \{ \text{semantics of } \text{valres} \} \\
&\begin{align*}
\text{var} & \quad \text{self} : \text{Object} \\
& \quad \text{self} := a \\
& \quad \begin{align*}
& \quad \text{val} \quad x : \mathbb{Z} \\
& \quad \begin{align*}
& \quad \text{self}.\text{bonus} \ldots \text{self is } \text{BAccount} \\
& \quad \begin{align*}
& \quad \text{self is } \text{Account} \\
& \quad \downarrow \text{OO}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{semantics of } \text{vres} \} \\
&\begin{align*}
\text{var} & \quad \text{self} : \text{Object} \\
& \quad \text{self} := a \\
& \quad \begin{align*}
& \quad \text{val} \quad x : \mathbb{Z} \\
& \quad \begin{align*}
& \quad \text{self}.\text{bonus} \ldots \text{self is } \text{BAccount} \\
& \quad \begin{align*}
& \quad \text{self is } \text{Account} \\
& \quad \downarrow \text{OO}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{end self : Object} \\
= \{ \text{semantics of } \text{val} \} \\
&\begin{align*}
\text{var} & \quad \text{self} : \text{Object} \\
& \quad \text{self} := a \\
& \quad \begin{align*}
& \quad \text{var} \quad x : \mathbb{Z} \\
& \quad \text{x} := 10 \\
& \quad \begin{align*}
& \quad \text{self}.\text{bonus} \ldots \text{self is } \text{BAccount} \\
& \quad \begin{align*}
& \quad \text{self is } \text{Account} \\
& \quad \downarrow \text{OO}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\]

\[
\text{end self : Object} \\
= \{ \text{self is } \text{BAccount} \} \\
&\begin{align*}
\text{var} & \quad \text{self} : \text{Object} \\
& \quad \text{self} := a \\
& \quad \begin{align*}
& \quad \text{var} \quad x : \mathbb{Z} \\
& \quad \text{x} := 10 \\
& \quad \begin{align*}
& \quad \text{self}.\text{bonus} := \text{self}.\text{bonus} + 1 \\
& \quad \text{end} \quad x : \mathbb{Z} \\
& \quad \text{self}.\text{balance} := \text{self}.\text{balance} + x \\
\end{align*}
\end{align*}
\end{align*}
\]

This can be expanded to a predicate that establishes the final value of \(a\) to be its initial value with state components updated by assignments.
1.8 Conclusions

We have presented a stepwise introduction to object-oriented concepts in the Unifying Theories of Programming. We started with the definition of observational variables and healthiness conditions, that restrict the values of these variables. The closedness properties of the healthiness conditions were stated and proved, to show, for example, that any two programs (or specifications) which are independently valid for a given healthiness condition can be combined by conjunction, disjunction or sequential composition, and the resulting program is still part of the lattice of predicates.

The declarations of classes, state components and operations are defined in terms of the theory of designs, which itself filters the subset of terminating programs, combined with higher-order programming, which allows variables to record the behaviour associated to operations (abstractions), including dynamic binding resolution. We saw that each of these declarations preserves the defined healthiness conditions.

Type checking is an important issue in object-oriented languages. To record typing information, special variables were introduced and well-definedness conditions were specified for the new object-oriented commands and expressions. Those forms of commands and expressions already introduced by Hoare and He for a sequential programming language, however, had to be revised to cope with typing information, including operation calls. We allow operations co- and contra-variance of arguments and return types.

We have seen also that the separation of declarations in different blocks has allowed the definition of (mutually) recursive operations in a straightforward manner. Furthermore, this has allowed a compositional approach which focuses on each feature in isolation. Another facility is specially related to dynamic binding resolution. When processing a operation declaration, the observational variable responsible for that operation is updated to reflect the new operation’s meaning. This might introduce a new observational variable for the operation (when processing the first definition of an operation), or updating the value of an existing variable to take into account the dynamic binding resulting from an operation redefinition.

By all presented, we have a theory of object-orientation that handles inheritance, recursive data types, dynamic binding, polymorphism, and mutually recursive operations. Some considerations about verification are discussed in the sequel.

1.8.1 Verification

As we have already presented, the concept of refinement for the UTP is universal implication. If we are expected to introduce pre- and postconditions, in Hoare style, we have a straightforward manner to perform verification. As an example, suppose we have the following specification for the operation debit: informally, we cannot perform a debit without enough funds.

\[\text{pre } \text{self}.\text{balance} > \text{value}\]
\[\text{post } \text{self}.\text{balance}' = \text{self}.\text{balance} - \text{value}\]
To check this specification against its operation definition, at any operation call $le.m(args)$ the semantics could be extended to

$$OO ((D(le.m(args)) \land p)(le, args) ; \mathit{post}(args))$$

where the precondition is extended with the copy-by-value variables ($val$ and $vres$) and the postcondition is extended with the result variables ($res$ and $vres$). Notice that if the precondition is violated, the operation call behaves like abort ($true ; P = true$, from designs), and if the postcondition is violated, it aborts the remaining program.

Moreover, there is a simpler alternative. Remember that in the UTP programs and specifications are interchangeable; the verifications thus would be part of the operation body itself, and in this case the operation call semantics remains the same. The expression

$$OO ((D(le.m(args))) \land (pds_e \bullet p)(le, args) ; \mathit{post}(args))$$

is the same of declaring a lengthier operation body with pre- and post-conditions.
Bibliography


CML Definition 2 — Operational Semantics

Deliverable Number: D23.3-4
Version: 0.2
Date: 22 February 2013
Public Document

http://www.compass-research.eu
Contributors:

Ana Cavalcanti, York
Jim Woodcock, York

Editors:

Jim Woodcock, York

Reviewers:

Zoe Andrews, Newcastle
Joey Coleman, Aarhus
Marcel Oliveira, UFPE
Richard Payne, Newcastle
## Document History

<table>
<thead>
<tr>
<th>Ver</th>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>21-02-2013</td>
<td>Jim Woodcock</td>
<td>Initial document version</td>
</tr>
<tr>
<td>0.2</td>
<td>21-02-2013</td>
<td>Jim Woodcock</td>
<td>Update following internal review</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>3.2.3.4</td>
<td><code>rule::cml-transition::Div</code></td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>3.2.3.5</td>
<td><code>rule::cml-transition::input</code></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>3.2.3.6</td>
<td><code>rule::cml-transition::output</code></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>3.2.3.7</td>
<td><code>rule::cml-transition::variable-block::begin</code></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>3.2.3.8</td>
<td><code>rule::cml-transition::variable-block::visible</code></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>3.2.3.9</td>
<td><code>rule::cml-transition::variable-block::end</code></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>3.2.3.10</td>
<td><code>rule::cml-transition::sequence::progress</code></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>3.2.3.11</td>
<td><code>rule::cml-transition::sequence::end</code></td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>3.2.3.12</td>
<td><code>rule::cml-transition::nondeterministic-choice</code></td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>3.2.3.13</td>
<td><code>rule::cml-transition::guard</code></td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>3.2.3.14</td>
<td><code>rule::cml-transition::external-choice::begin</code></td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>3.2.3.15</td>
<td><code>rule::cml-transition::external-choice::end</code></td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>3.2.3.16</td>
<td><code>rule::cml-transition::external-choice::silent</code></td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>3.2.3.17</td>
<td><code>rule::cml-transition::external-choice::Skip</code></td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3.2.3.18</td>
<td><code>rule::cml-transition::parallel::begin</code></td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3.2.3.19</td>
<td><code>rule::cml-transition::parallel::independent</code></td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3.2.3.20</td>
<td><code>rule::cml-transition::parallel::synchronised</code></td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3.2.3.21</td>
<td><code>rule::cml-transition::parallel::end</code></td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3.2.4</td>
<td><code>lift</code></td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>3.2.4.1</td>
<td><code>definition::lift</code></td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>3.2.4.2</td>
<td><code>law::lift::composition</code></td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>3.2.4.3</td>
<td><code>law::lift::CSP4</code></td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>3.2.4.4</td>
<td><code>law::lift::external-choice</code></td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>3.2.4.5</td>
<td><code>law::lift::left-unit</code></td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>3.2.4.6</td>
<td><code>law::lift::leading::conjunctive</code></td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>3.2.4.7</td>
<td><code>law::lift::leading::substitution</code></td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>3.2.4.8</td>
<td><code>law::lift::leading::reactive-design</code></td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>3.2.4.9</td>
<td><code>law::lift::merge</code></td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>3.2.4.10</td>
<td><code>law::lift::parallel::distributivity</code></td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>3.2.4.11</td>
<td><code>law::lift::semi-idempotence</code></td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>3.2.4.12</td>
<td><code>law::lift::var</code></td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Soundness</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>3.3.1</td>
<td><code>theorem::CML::assignment-rule-sound</code></td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>3.3.2</td>
<td><code>theorem::CML::input::sound</code></td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>3.3.3</td>
<td><code>theorem::CML::nondeterministic-choice-rule-sound</code></td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>3.3.4</td>
<td><code>theorem::CML::output-rule-sound</code></td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>3.3.5</td>
<td><code>theorem::CML::external-choice::begin-sound</code></td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>3.3.6</td>
<td><code>theorem::CML::external-choice::Skip-rule-sound</code></td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>3.3.7</td>
<td><code>theorem::CML::external-choice::end-sound</code></td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>3.3.8</td>
<td><code>theorem::CML::parallel::begin-sound</code></td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>3.3.9</td>
<td><code>theorem::CML::parallel::independent-sound</code></td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>3.3.10</td>
<td><code>theorem::CML::parallel::end-sound</code></td>
<td>38</td>
<td></td>
</tr>
</tbody>
</table>

### A Algebraic Laws for CML

#### A.1 propositional-calculus

- A.1.1 `propositional-calculus::and`
  - A.1.1.1 `law::propositional-calculus::and::associativity` | 40
  - A.1.1.2 `law::propositional-calculus::and::commutativity` | 40
  - A.1.1.3 `law::propositional-calculus::and::elimination` | 40
A.1.1.4 law::propositional-calculus:-and:-idempotence .......... 40
A.1.1.5 law::propositional-calculus:-and:-or-distributivity .... 40
A.1.1.6 law::propositional-calculus:-and:-unit ............... 41
A.1.1.7 law::propositional-calculus:-and:-zero .............. 41

A.1.2 propositional-calculus:-or .................................. 41
A.1.2.1 law::propositional-calculus:-or:-absorption .......... 41
A.1.2.2 law::propositional-calculus:-or:-elimination ....... 41
A.1.2.3 law::propositional-calculus:-or:-idempotence ........ 41
A.1.2.4 law::propositional-calculus:-or:-introduction ....... 41
A.1.2.5 law::propositional-calculus:-or:-subsumption ....... 41
A.1.2.6 law::propositional-calculus:-or:-unit ............... 42
A.1.2.7 law::propositional-calculus:-or:-zero ............... 42

A.1.3 propositional-calculus:-negation ......................... 42
A.1.3.1 law::propositional-calculus:-negation:-false ........ 42
A.1.3.2 law::propositional-calculus:-negation:-true .......... 42
A.1.3.3 law::propositional-calculus:-negation:-contradiction 42
A.1.3.4 law::propositional-calculus:-negation:-contradiction:-rewrite 42
A.1.3.5 law::propositional-calculus:-negation:-De-Morgan ... 42
A.1.3.6 law::propositional-calculus:-negation:-double-negation 43
A.1.3.7 law::propositional-calculus:-negation:-excluded-middle 43
A.1.3.8 law::propositional-calculus:-negation:-absorption .... 43

A.1.4 propositional-calculus:-implies .......................... 43
A.1.4.1 law::propositional-calculus:-implies .................. 43
A.1.4.2 law::propositional-calculus:-implies:-absorption .... 43
A.1.4.3 law::propositional-calculus:-implies:-accumulation 43
A.1.4.4 law::propositional-calculus:-implies:-and-antecedent 43
A.1.4.5 law::propositional-calculus:-implies:-and-consequent 44
A.1.4.6 law::propositional-calculus:-implies:-contradiction 44
A.1.4.7 law::propositional-calculus:-implies:-export ......... 44
A.1.4.8 law::propositional-calculus:-implies:-identity ....... 44
A.1.4.9 law::propositional-calculus:-implies:-negation ...... 44

A.1.5 propositional-calculus:-equivalence ....................... 44
A.1.5.1 law::propositional-calculus:-equivalence:-boolean ... 44

A.1.6 propositional-calculus:-conditional ....................... 44
A.1.6.1 definition::propositional-calculus:-conditional ....... 44
A.1.6.2 law::propositional-calculus:-conditional:-assumption-else 45
A.1.6.3 law::propositional-calculus:-conditional:-assumption-then 45
A.1.6.4 law::propositional-calculus:-conditional:-constant:-false 45
A.1.6.5 law::propositional-calculus:-conditional:-constant:-then 46
A.1.6.6 law::propositional-calculus:-conditional:-and-distributivity 46
A.1.6.7 law::propositional-calculus:-conditional:-exchange:-or 46
A.1.6.8 law::propositional-calculus:-conditional:-export:-then 47
A.1.6.9 law::propositional-calculus:-conditional:-export:-else 47
A.1.6.10 law::propositional-calculus:-conditional:-idempotence 47
A.1.6.11 law::propositional-calculus:-conditional:-negation .. 47
A.1.6.12 law::propositional-calculus:-conditional:-simplification-1 47

A.2 predicate-calculus ................................................. 48
A.2.1 predicate-calculus:-exists ................................. 48
A.2.1.1 law::predicate-calculus:-exists:-and:-non-free ...... 48
A.2.1.2 law::predicate-calculus::exists::De-Morgan 48
A.2.1.3 law::predicate-calculus::exists::detach 48
A.2.1.4 law::predicate-calculus::exists::introduction 48
A.2.1.5 law::predicate-calculus::exists::nested 48
A.2.1.6 law::predicate-calculus::exists::one-point 48
A.2.1.7 law::predicate-calculus::exists::or-distributivity 48
A.2.1.8 law::predicate-calculus::exists::rename-bound-var 49
A.2.2 predicate-calculus::forall 49
A.2.2.1 law::predicate-calculus::forall::detach 49
A.2.2.2 law::predicate-calculus::forall::tautology 49
A.2.2.3 law::predicate-calculus::forall::boolean 49
A.3 equals 50
A.3.1 law::equals::cartesian-pair 50
A.3.2 law::equals::Leibniz 50
A.3.3 law::equals::reflection 50
A.4 alphabet 51
A.4.1 definition::alphabet::lifting 51
A.4.2 law::alphabet::lifting::disjunctivity 51
A.4.3 law::alphabet::lifting::assignment 51
A.4.4 law::alphabet::and 51
A.4.5 law::alphabet::change-of-variable 51
A.4.6 law::alphabet::exists 51
A.4.7 law::alphabet::lifting::conjunctivity 51
A.4.8 law::alphabet::lifting::sequence 52
A.5 relational-calculus 53
A.5.1 relational-calculus::assignment 53
A.5.1.1 definition::relational-calculus::assignment 53
A.5.1.2 definition::relational-calculus::assignment::declaration-init 53
A.5.1.3 law::relational-calculus::assignment::alphabet-extension 53
A.5.1.4 law::relational-calculus::assignment::cancel-by-end-var 53
A.5.1.5 law::relational-calculus::assignment::lifted-identity 54
A.5.1.6 law::relational-calculus::assignment::sequence::identity 54
A.5.1.7 law::relational-calculus::assignment::sequence 54
A.5.1.8 law::relational-calculus::assignment::unwinding 55
A.5.2 relational-calculus::end 55
A.5.2.1 definition::relational-calculus::end::alphabetised-equation 55
A.5.2.2 definition::relational-calculus::end::split 55
A.5.2.3 definition::relational-calculus::end 55
A.5.2.4 law::relational-calculus::end::alphabetised 56
A.5.2.5 law::relational-calculus::end::assignment 56
A.5.2.6 law::relational-calculus::end::sequence 56
A.5.2.7 law::relational-calculus::end::var 57
A.5.3 relational-calculus::II 57
A.5.3.1 definition::relational-calculus::II 57
A.5.3.2 law::relational-calculus::II::unwinding 58
A.5.3.3 definition::relational-calculus::II::heterogeneous 58
A.5.4 relational-calculus::refinement 58
A.5.4.1 definition::relational-calculus::refinement 58
A.5.4.2 law::relational-calculus::refinement::nondeterministic-choice
A.5.4.3 law::relational-calculus::refinement::subsumption
A.5.5 relational-calculus::sequence
A.5.5.1 definition::relational-calculus::sequence
A.5.5.2 law::relational-calculus::sequence::condition-swing
A.5.5.3 law::relational-calculus::sequence::detach-post
A.5.5.4 law::relational-calculus::sequence::disjunctivity
A.5.5.5 law::relational-calculus::sequence::identity
A.5.5.6 law::relational-calculus::sequence::introduction
A.5.5.7 law::relational-calculus::sequence::left-unit
A.5.5.8 law::relational-calculus::sequence::left-zero
A.5.5.9 law::relational-calculus::sequence::monotonic-2
A.5.5.10 law::relational-calculus::sequence::one-point::left
A.5.5.11 law::relational-calculus::sequence::one-point::right
A.5.5.12 law::relational-calculus::sequence::post-separation
A.5.5.13 law::relational-calculus::sequence::pre-separation
A.5.5.14 law::relational-calculus::sequence::right-zero
A.5.6 relational-calculus::var
A.5.6.1 definition::relational-calculus::var
A.5.6.2 definition::relational-calculus::var::initialised-declaration
A.5.6.3 law::relational-calculus::var::alphabetised-equation
A.5.6.4 law::relational-calculus::var::disjoint-parallel
A.5.6.5 law::relational-calculus::var::identity
A.5.6.6 law::relational-calculus::var::pre-separation
A.5.6.7 law::relational-calculus::var::scope::left
A.5.6.8 law::relational-calculus::var::parallel
A.5.6.9 law::relational-calculus::var::sequence
A.5.6.10 law::relational-calculus::var::split
A.6 set-theory
A.6.1 law::set-theory::union::left-subset
A.7 substitution
A.7.1 law::substitution::alphabet-restrict
A.7.2 law::relational-calculus::end::substitution
A.7.3 law::relational-calculus::II::ok''substitution
A.7.4 law::relational-calculus::assignment::substitution
A.7.5 law::relational-calculus::var::substitution
A.7.6 law::substitution
A.7.7 law::substitution-2
A.8 concurrency
A.8.1 definition::concurrency::disjoint-parallel
A.8.2 law::concurrency::disjoint-parallel::monotonic
A.8.3 law::concurrency::disjoint-parallel::substitution::plain
A.8.4 law::concurrency::disjoint-parallel::substitution::separation
A.9 design
A.9.1 definition::design
A.9.2 definition::design::ok''substitution
A.9.3 definition::design::wait::substitution
A.9.4 law::design::exists
A.9.5 law::design::ok''substitution::true
A.9.6 law::design:-ok'-substitution:-false .......................... 74
A.9.7 law::design:-post-ok:-Leibniz .............................. 74
A.9.8 law::design:-post-ok .......................................... 75
A.9.9 law::design:-post:-simplification ......................... 75
A.9.10 law::design:-pre-ok:-Leibniz .............................. 75
A.9.11 law::design:-pre-ok .......................................... 75
A.9.12 law::design:-refinement:-strengthen-post ............. 76
A.9.13 law::design:-refinement:-weaken-pre .................. 76
A.10 reactive ....................................................... 77
A.10.1 reactive::II-R ............................................... 77
  A.10.1.1 definition::reactive::R1 ............................... 77
  A.10.1.2 definition::reactive::II-R::R1 ..................... 77
  A.10.1.3 definition::reactive::II-R ......................... 77
  A.10.1.4 definition::reactive::R3 ............................. 77
  A.10.1.5 definition::reactive::R3post ....................... 77
  A.10.1.6 definition::reactive::R3pre ....................... 77
  A.10.1.7 definition::reactive::R .............................. 77
  A.10.1.8 law::reactive::R1::design::pre-cancellation .... 78
  A.10.1.9 law::reactive::R1::extension ..................... 78
  A.10.1.10 law::reactive::J::splitting ...................... 78
  A.10.1.11 law::reactive::R1::R3::commutative ............ 79
  A.10.1.12 law::reactive::R1::R3post::commutative ....... 79
  A.10.1.13 law::reactive::R1::conditional ................. 79
  A.10.1.14 law::reactive::R1::conjunctive-cancellation ... 80
  A.10.1.15 law::reactive::R1::constant ..................... 80
  A.10.1.16 law::reactive::R1::detach ....................... 80
  A.10.1.17 law::reactive::R1::disjunctivity ............... 80
  A.10.1.18 law::reactive::R1::exists ....................... 81
  A.10.1.19 law::reactive::R1::false ......................... 81
  A.10.1.20 law::reactive::R1::idempotence ................. 81
  A.10.1.21 law::reactive::R1::II ............................ 81
  A.10.1.22 law::reactive::R1::ok' substitution ........... 81
  A.10.1.23 law::reactive::R1::sequence::closure .......... 82
  A.10.1.24 law::reactive::R1::sequence::left-zero ....... 82
  A.10.1.25 law::reactive::R1::substitution::wait' ....... 82
  A.10.1.26 law::reactive::R1::substitution ................. 82
  A.10.1.27 law::reactive::R1::wait-substitution .......... 82
  A.10.1.28 law::reactive::R3::design-split ............... 83
  A.10.1.29 law::reactive::R3::not-wait::substitution .... 83
  A.10.1.30 law::reactive::R3::not-wait .................... 83
  A.10.1.31 law::reactive::R3::ok' substitution ........... 84
  A.10.1.32 law::reactive::R3::sequence::closure .......... 84
  A.10.1.33 law::reactive::R3post::wait::false ............ 84
  A.10.1.34 law::reactive::R3post::wait::true ............. 85
  A.10.1.35 law::reactive::R3pre::wait::false ............. 85
  A.10.1.36 law::reactive::R3pre::wait::true ............. 85
A.10.2 wp-R1 ..................................................... 85
  A.10.2.1 definition::wp-R1 .................................. 85
  A.10.2.2 law::reactive::wp-R1::simplification-1 ....... 85
A.13.4.3 law::-CML::Skip::not-wait::diverging .......................... 113
A.13.4.4 law::-CML::Skip::not-wait::not-diverging ................. 113
A.13.4.5 law::-CML::Skip::not-wait .................................. 114
A.13.4.6 law::-CML::Skip::ok'::substitution ........................ 115
A.13.4.7 law::-CML::Skip::R1 ........................................ 117
A.13.4.8 law::-CML::Skip::reactive-design ......................... 118
A.13.5 assignment ....................................................... 118
A.13.5.1 law::-CML::assignment::composition ........................ 118
A.13.5.2 law::-CML::assignment::restriction ........................ 118
A.13.5.3 law::-CML::assignment::substitution::one-point-rule .... 119
A.13.5.4 definition::-CML::assignment::substitution ............. 119
A.13.5.5 law::-CML::assignment::substitution ..................... 119
A.13.5.6 definition::-CML::assignment ....................... 120
A.13.6 input ............................................................ 120
A.13.6.1 definition::-CML::input .................................. 120
A.13.6.2 law::-CML::input::absorption ............................ 120
A.13.7 external-choice ................................................. 121
A.13.7.1 definition::-CML::external-choice::distributed .......... 121
A.13.7.2 law::-CML::external-choice::assignment .................. 121
A.13.7.3 law::-reactive-design::external-choice::distributed::not-wait ................................. 122
A.13.7.4 law::-reactive-design::external-choice::distributed::post .................................. 122
A.13.7.5 law::-reactive-design::external-choice::distributed::pre .......... 123
A.13.7.6 law::-CML::external-choice::distribution::union ........ 123
A.13.7.7 law::-CML::external-choice::idempotence ................ 125
A.13.7.8 law::-CML::external-choice::lift-distributive .......... 125
A.13.7.9 law::-CML::external-choice::monotonic .................. 126
A.13.7.10 law::-CML::external-choice::Skip ........................ 126
A.13.8 extra-choice ..................................................... 127
A.13.8.1 definition::-CML::extrachoice ............................ 127
A.13.8.2 law::-CML::extrachoice::monotonic .................... 127
A.13.9 parallel-composition .......................................... 127
A.13.9.1 definition::-CML::parallel-by-merge ........................ 127
A.13.9.2 law::-CML::parallel::cotermination ...................... 127
A.13.9.3 law::-CML::parallel::disjunctivity ...................... 127
A.13.9.4 law::-CML::parallel::post ................................ 128
A.13.9.5 law::-CML::parallel::pre .................................. 128
A.13.9.6 law::-CML::parallel::CSP2 ............................... 132
A.13.9.7 law::-CML::parallel::distribution-external-choice .... 133
A.13.9.8 law::-CML::parallel::step-law ........................... 133
A.13.9.9 law::-CML::parallel::substitution ...................... 133
A.13.9.10 law::-CML::parallel::wait-substitution ............. 133
Chapter 1

Introduction

In this part of the deliverable we describe the operational semantics for the CML kernel language; that is, the CML Definition 1, but without time. A preliminary version of the operational semantics was presented in deliverable D23.2.

We assume familiarity with Unifying Theories of Programming (UTP) and the CML language itself. However, to widen the accessibility of the material, we provide an introduction to the technique of operational semantics and try to explain notation as we go along.

In Chapter 2, we describe Plotkin-style structured operational semantics (SOS) in the UTP setting. To motivate the SOS for CML, we describe individual SOSs for its two constituent parts: the abstract nondeterministic programs described by VDM (Section 2.2) and the behavioural process algebra of CSP (Section 2.4). We also describe examples of the use of the different SOSs. In Section 2.5, we give a detailed overview of the SOS for CML, motivating the key ideas, including its novel symbolic nature.

The following chapters are more technical. Chapter 3 contains all the rules for the kernel language, the auxiliary definitions required for the rules, and theorems for the soundness of the majority of the rules. Although not required for this deliverable, it is intended that the operational semantics should be mechanised in Isabelle/HOL, complete with soundness proofs. The proofs of the soundness theorems are detailed enough to serve as the basis for this mechanisation. Appendix A contains an indexed collection of algebraic laws for CML.
Chapter 2

Operational Semantics in UTP

We present a Plotkin-style structured operational semantics (SOS) for the CML kernel language. This will be extended in future deliverables to include time and object-orientation.

The operational semantics is described by a transition relation for the language constructs. These rules can be used to define an abstract interpreter for the language, giving possible execution steps for CML processes. This description can then be used as a guide for implementing model checking, refinement checking, animation, and testing.

An operational semantics is only a partial language definition, leaving many important questions unanswered. For example, it does not define a notion of program refinement, which is at the heart of the denotational semantics. It does not give a method for deciding which transition rules should be selected when several are applicable. Finally, it tells only part of the story for the semantics of recursion and iteration: the transition rule for recursion is based on the copy rule for unfolding the definition; but there is no position on whether this is the least or greatest fixed-point.

2.1 Transition relations

The operational meaning of a CML action is a computation: a sequence of individual steps that the action can make as it executes. These steps are represented by a transition relation between individual machine states: \((s, P)\). Here, \(s\) is a text assigning constants to alphabetical variables and \(P\) is an action text. The pair represents the current state of the computation, \(s\), and the action yet to be executed, \(P\). It is important to note that the transition relation relates *syntactic* objects, not semantic ones. When we need to relate syntax to semantics, we write \(P\) to describe the syntax of a action \(P\) with semantics \(P\).

If we allow \(\text{Skip}\) to represent program termination, then \((s, \text{Skip})\) is a terminal execution state, where \(s\) is the final value of the computation.

To understand the meaning of the transition relation, suppose we start in state \((s, P)\) and transit to state \((t, Q)\). We write this as

\[(s, P) \rightarrow (t, Q)\]
The transition relation will be given inductively over the syntax of the language. Here, the after-state \((t, Q)\) is one possible outcome of executing \(P\) starting from \(s\). Of course, if \(P\) is deterministic, then there will be a unique outcome; but if \(P\) is nondeterministic, then there may be many different outcomes, of which \((t, Q)\) may represent one or more, but not necessarily all. For example, suppose that we have the following action:

\[
x := x + 1 | \sim | x := x + 2
\]

and we start from the state \(x := 0\). The first step taken by the action will be to resolve the nondeterministic choice, perhaps to \(x := x + 1\). In this precise sense, the after state must be a refinement of the starting state. So we might see the following computation leading to a terminal execution state:

\[
(x := 0, x := x + 1 | \sim | x := x + 2) \longrightarrow (x := 0, x := x + 1) \longrightarrow (x := 1, \text{Skip})
\]

Of course, another perfectly valid computation sequence would be

\[
(x := 0, x := x + 1 | \sim | x := x + 2) \longrightarrow (x := 0, x := x + 2) \longrightarrow (x := 2, \text{Skip})
\]

This observation about the nature of the execution of a nondeterministic program gives us a neat connection between syntax and semantics that we can exploit as a definition of the transition relation:

\[
(s, P) \longrightarrow (t, Q) \equiv (s ; P) \sqsubseteq (t ; Q)
\]

This definition connects the operational and denotational semantics and allows us to verify the soundness of the transition relation with respect to the denotational semantics, as we demonstrate in the next section.

### 2.2 Transition Relation for Programming Language

Before describing the complexities of the operational semantics for CML, we describe the operational semantics for its principal components. In this section, we describe the transition relation for a simple nondeterministic sequential programming language. The transition relation is given in full in Figure 2.1. Some points to note:

- The rule for recursion is simply unfolding. Definitions of function names can also be expanded. In fact, the rule is equivalent to

\[
(s, \mu F) \longrightarrow (s, F(\mu F)) \quad \text{where } F = \lambda X @ P(X)
\]

- There is no transition rule for \(\text{Skip}\). This means that a program can progress no further than any state \((s, \text{Skip})\); but fortunately this is an accepting state, since it is a terminal execution state.

- The aborting program \(\text{abort}\) is stuck: it will engage in an endless sequence of transitions, none of which results in a terminal execution state.
<table>
<thead>
<tr>
<th>Assignment</th>
<th>$(s, v := e) \rightarrow (s ; v := e, \text{Skip})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip</td>
<td>$(s, (\text{Skip} ; Q)) \rightarrow (s, Q)$</td>
</tr>
<tr>
<td>Sequential composition</td>
<td>$(s, (P ; R)) \rightarrow (t, (Q ; R))$ whenever $(s, P) \rightarrow (t, Q)$</td>
</tr>
<tr>
<td>Nondeterministic choice</td>
<td>$(s, (P \mid Q)) \rightarrow (s, P)$</td>
</tr>
<tr>
<td>Nondeterministic choice</td>
<td>$(s, (P \mid Q)) \rightarrow (s, Q)$</td>
</tr>
<tr>
<td>Conditional</td>
<td>$(s, (\text{if} \ b \ \text{then} \ P \ \text{else} \ Q)) \rightarrow (s, P)$ whenever $(s ; b)$</td>
</tr>
<tr>
<td>Nondeterministic choice</td>
<td>$(s, (\text{if} \ b \ \text{then} \ P \ \text{else} \ Q)) \rightarrow (s, Q)$ whenever $(s ; \neg b)$</td>
</tr>
<tr>
<td>Recursion</td>
<td>$(s, (\mu X @ P (X))) \rightarrow (s, P (\mu X @ P (X)))$</td>
</tr>
<tr>
<td>Abort</td>
<td>$(s, \text{abort}) \rightarrow (s, \text{abort})$</td>
</tr>
<tr>
<td>Skip</td>
<td>no transition</td>
</tr>
</tbody>
</table>

Figure 2.1: Transition relation for sequential programming language

Consider the soundness of the rule for sequential composition:

$$(s, (P ; R)) \rightarrow (t, (Q ; R)) \quad \text{whenever } (s, P) \rightarrow (t, Q)$$

To prove the soundness of this transition step, we need to show that

$$(s ; (P ; R)) \sqsubseteq (t ; (Q ; R))$$

We reason as follows. First, assume that $(s ; P) \sqsubseteq (t ; Q)$. Then,

\[
(s ; (P ; R)) = \begin{cases} \text{associativity of sequential composition} \\ (s ; P) ; R \end{cases}
\sqsubseteq \begin{cases} \text{monotonicity of sequential composition} \\ ((t ; Q) ; R) \end{cases}
= \begin{cases} \text{associativity of sequential composition} \\ (t ; (Q ; R)) \end{cases}
\]

This is a attractively simple proof: the transition’s correctness depends on just two algebraic laws:

- Sequential composition is associative.
- Sequential composition is monotonic with respect to refinement.

This illustrates the connection between three presentations of the semantics of CML: denotational, algebraic, and operational (we haven’t presented a complete algebraic semantics; we merely refer to particular algebraic laws). The three presentations are intertwined.
in an interesting way: the soundness of the operational semantics depends on the denotational semantics and particular algebraic laws. There are further relationships between the three presentations, which we shall not explore here.

### 2.2.1 Example execution sequence

To illustrate the use of the operational semantics, consider the following simple program:

\[
i := 1; \text{while} i < 3 \text{ do } i := i + 1
\]

Here, the while-loop is syntactically equivalent to the program \( \mu F \), which computes the least fixed-point of \( F \), where

\[
F(X) = (\text{if } i < 3 \text{ then } (i := i + 1; X) \text{ else } \text{Skip})
\]

We now execute this code according to the operational semantics, starting from the initial state \( i := 0 \).

\[
(i := 0, (i := 1; \text{while} i < 3 \text{ do } i := i + 1))
\]

\[
\rightarrow \{ \text{assignment, sequence } \}
\]

\[
((i := 0; i := 1), (\text{while} i < 3 \text{ do } i := i + 1))
\]

\[
= \{ \text{rewrite state assignment } \}
\]

\[
(i := 1, (\text{while} i < 3 \text{ do } i := i + 1))
\]

\[
= \{ \text{local definition, rewrite while } \}
\]

\[
(i := 1, \mu F) \quad \text{where } F(X) = (\text{if } i < 3 \text{ then } (i := i + 1; X) \text{ else } \text{Skip})
\]

\[
\rightarrow \{ \text{recursion } \}
\]

\[
(i := 1, F(\mu F))
\]

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

\[
(i := 1, (\text{if } i < 3 \text{ then } (i := i + 1; \mu F) \text{ else } \text{Skip}))
\]

\[
\rightarrow \{ \text{conditional } \}
\]

\[
(i := 1, (i := i + i; \mu F))
\]

\[
\rightarrow \{ \text{assignment, sequence } \}
\]

\[
((i := 1; i := i + i), \mu F)
\]

\[
= \{ \text{rewrite state assignment } \}
\]

\[
(i := 2, \mu F)
\]

\[
\rightarrow \{ \text{recursion } \}
\]

\[
(i := 2, F(\mu F))
\]

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

\[
(i := 2, (\text{if } i < 3 \text{ then } (i := i + 1; \mu F) \text{ else } \text{Skip}))
\]

\[
\rightarrow \{ \text{conditional } \}
\]

\[
(i := 2, (i := i + i; \mu F))
\]

\[
\rightarrow \{ \text{assignment, sequence } \}
\]
\[(i := 2 ; i := i + 1, \mu F)\]
\[= \{ \text{rewrite state assignment} \}\]
\[(i := 4, \mu F)\]
\[\rightarrow \{ \text{recursion} \}\]
\[(i := 4, F(\mu F))\]
\[= \{ \text{definition of} \ F \} \]
\[(i := 4, (\text{if} \ i < 3 \ \text{then} \ (i := i + 1 ; \mu F) \ \text{else} \ \text{Skip}))\]
\[\rightarrow \{ \text{conditional} \}\]
\[(i := 4, \text{Skip})\]

We leave the program in its terminal execution state with the final value of the sole state variable. Provided that we have verified the soundness of the transition system, then this is a correct computation. The operational semantics establishes the following (semantic) theorem about the correctness of the program:

\[(i := 0 ; (i := 1 ; \textbf{while} \ i < 3 \ \textbf{do} \ i := i + 1)) \subseteq (i := 4 ; \text{Skip})\]

Of course, since the program is entirely deterministic, this theorem could be strengthened to an equality.

### 2.3 Operational Semantics: Reflections

In UTP, we derive the algebraic semantics of the language from the denotational semantics; we then derive the operational semantics from the algebraic semantics. There are many choices to be made in collecting the transition rules for the operational semantics. For example, we have included two transitions for nondeterministic choice, but we have excluded any transition for \textit{Skip} as a complete program and we could have omitted the stuck transition for the aborting program. All these decisions would still have led to a sound operational semantics.

These are acknowledged dangers of the operational approach: too many transitions, correct because of the reflexivity of refinement; too few transitions, correct by default.

These are acknowledged dangers of the operational approach: there can be both too many and too few transitions. For example, adding a silent identity transition for \textit{Skip} would be correct, since refinement is reflexive, but it would introduce an artificial divergence. Omitting any transition would be correct by default. The solution is to reconstruct both the algebraic and the denotational semantics from the operational semantics, but that is beyond the scope of the project. For now, we must accept that this is just one possible operational semantics for the language.

### 2.4 Operational Semantics for CSP

We now describe the operational semantics for pure CSP. This extends our previous example of the nondeterministic sequential programming language by requiring labelled transitions,
but there is no need for describing imperative state. Similar accounts of the operational semantics of CSP may be found in [?, ?, ?>.

So now there are two kinds of transitions:

1. silent transitions
   \[ P \rightarrow Q \equiv P \subseteq Q \]

2. action transitions
   \[ P \xrightarrow{a} Q \equiv P \subseteq (a \rightarrow Q) \square P \]

Notice that the definition of the labelled transition is slightly more complicated than that for the unlabelled transition. An action transition is triggered by an event \( a \) and the subsequent behaviour is \( Q \), so \( a \rightarrow Q \) is one of the possible behaviours of \( P \). For example, suppose that we have \( P = (a \rightarrow Q) \) \( b \rightarrow R \). Clearly, \( P \xrightarrow{a} Q \), but (by analogy with an unlabelled transition) we do not have \( P \subseteq a \rightarrow Q \).

The complete operational semantics for pure CSP is contained in Figure 2.2. Clearly, the most complicated operator is parallel composition, with five transition rules. Of course, if we remove symmetries, then there are only three rules: one of the processes makes an internal, silent transition; one of the processes makes an independent, visible transition; or both processes synchronise on a common event (one in the synchronisation set, \( cs \)).

### 2.4.1 Example: Two-place buffer

To demonstrate the use of the operational semantics, consider the execution of a simple two-place buffer defined by the recursive equations:

\[
\begin{align*}
\text{BUF1} &= \text{in}.0 \rightarrow \text{mid}.0 \rightarrow \text{BUF1}[], \text{in}.1 \rightarrow \text{mid}.1 \rightarrow \text{BUF1}[] \\
\text{BUF2} &= \text{mid}.0 \rightarrow \text{out}.0 \rightarrow \text{BUF2}[], \text{mid}.1 \rightarrow \text{out}.1 \rightarrow \text{BUF2}[]
\end{align*}
\]

Both processes are one-place buffers, communicating across the \( \text{mid} \) channel. We can see that their parallel composition is a two-place buffer by executing it and trying to force as many \text{in} events as possible.

\[
\begin{align*}
\text{BUF1}[] &\parallel \{ \parallel \text{mid} \} || \text{BUF2}[] \\
&= \{ \text{definitions} \} \\
&\parallel \{ \parallel \text{in} \} || \text{BUF1}[\parallel \{ \parallel \text{mid} \} || \text{in}.0 \rightarrow \text{mid}.0 \rightarrow \text{BUF1}[\parallel \{ \parallel \text{mid} \} || \text{in}.1 \rightarrow \text{mid}.1 \rightarrow \text{BUF1}] \\
&\parallel \{ \parallel \text{mid} \} || \text{BUF2}[\parallel \{ \parallel \text{mid} \} || \text{mid}.0 \rightarrow \text{out}.0 \rightarrow \text{BUF2}[\parallel \{ \parallel \text{mid} \} || \text{mid}.1 \rightarrow \text{out}.1 \rightarrow \text{BUF2}] \\
&\parallel \{ \parallel \text{mid} \} || \text{BUF1}[\parallel \{ \parallel \text{mid} \} || \text{in}.0 \rightarrow \text{mid}.0 \rightarrow \text{BUF1}] \\
&\parallel \{ \parallel \text{mid} \} || \text{BUF2}[\parallel \{ \parallel \text{mid} \} || \text{mid}.0 \rightarrow \text{out}.0 \rightarrow \text{BUF2}[\parallel \{ \parallel \text{mid} \} || \text{mid}.1 \rightarrow \text{out}.1 \rightarrow \text{BUF2}] \\
&\parallel \{ \parallel \text{mid} \} || \text{BUF1}[\parallel \{ \parallel \text{mid} \} || \text{in}.0 \rightarrow \text{mid}.0 \rightarrow \text{BUF1}] \\
&\parallel \{ \parallel \text{mid} \} || \text{BUF2}[\parallel \{ \parallel \text{mid} \} || \text{mid}.0 \rightarrow \text{out}.0 \rightarrow \text{BUF2}[\parallel \{ \parallel \text{mid} \} || \text{mid}.1 \rightarrow \text{out}.1 \rightarrow \text{BUF2}]
\end{align*}
\]
Simple prefixing
\[(a \rightarrow P) \xrightarrow{a} P\]

Generalised choice
\[(x: E \rightarrow Q(x)) \xrightarrow{a} Q(a) \quad \text{if } a \in E\]

Internal choice
\[P \mid \sim \mid Q \rightarrow P \quad P \mid \sim \mid Q \rightarrow Q\]

External choice

<table>
<thead>
<tr>
<th>Silent transitions</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P [] Q \rightarrow P' [] Q \quad \text{if } P \rightarrow P'</td>
<td>P [] Q \xrightarrow{a} P' \quad \text{if } P \xrightarrow{a} P'</td>
</tr>
<tr>
<td>P [] Q \rightarrow P' [] Q' \quad \text{if } Q \rightarrow Q'</td>
<td>P [] Q \xrightarrow{a} Q' \quad \text{if } Q \xrightarrow{a} Q'</td>
</tr>
</tbody>
</table>

Sequential composition

<table>
<thead>
<tr>
<th>Left silent</th>
<th>Left action</th>
<th>Left termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>P ; Q \rightarrow P' ; Q \quad \text{if } P \rightarrow P'</td>
<td>P ; Q \xrightarrow{a} P' ; Q \quad \text{if } P \xrightarrow{a} P'</td>
<td>SKIP ; Q \rightarrow Q</td>
</tr>
</tbody>
</table>

Hiding

Silent transition or hidden action
\[P \setminus E \rightarrow P' \setminus E \quad \text{if } P \rightarrow P' \text{ or } (P \xrightarrow{a} P' \text{ and } a \in E)\]

Visible action
\[P \setminus E \xrightarrow{a} P' \setminus E \quad \text{if } P \xrightarrow{a} P' \text{ and } a \notin E\]

Parallel composition

<table>
<thead>
<tr>
<th>Left silent</th>
<th>Right silent</th>
<th>Left independent</th>
<th>Right independent</th>
<th>Synchronised action</th>
<th>Recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>P [</td>
<td>cs</td>
<td>] Q \rightarrow P' [</td>
<td>cs</td>
<td>] Q \quad \text{if } P \rightarrow P'</td>
<td>P [</td>
</tr>
</tbody>
</table>

Figure 2.2: Operational Semantics for CSP
The rules have allowed us to execute the parallel composition to obtain the following trace: \((\text{in}.0 \rightarrow \text{mid}.0 \rightarrow \text{BUF1}) \mid \mid (\text{in}.1 \rightarrow \text{mid}.1 \rightarrow \text{BUF1}) \mid \mid (\text{out}.0 \rightarrow \text{BUF2})\). We have forced two in events and now there is only one rule that can be applied, which will result in the event \((\text{out}.0 \rightarrow \text{BUF2})\). This demonstrates that the parallel composition can hold at least a two values. By considering the not just a trace, but a tree of transitions, it is not difficult to see that it cannot hold more than two values, nor can values be reordered or invented. It is truly a two-place buffer. The point is that the operational semantics gives an abstract way of executing the parallel composition to see what can happen.

### 2.5 Operational Semantics for CML—Overview

CML is essentially a compound of abstract imperative programming and CSP, and so its operational semantics will contain both sets of transition rules. We augment this with a new idea: symbolic transitions. In CSP, the action \(c?x:\{0,\ldots,9\} \rightarrow P(x)\) has 10 possible transitions:

\[
\begin{align*}
(c?x:\{0,\ldots,9\} \rightarrow P(x)) & \xrightarrow{c.0} P(0) \\
(c?x:\{0,\ldots,9\} \rightarrow P(x)) & \xrightarrow{c.1} P(1) \\
& \vdots \\
(c?x:\{0,\ldots,9\} \rightarrow P(x)) & \xrightarrow{c.9} P(9)
\end{align*}
\]

In the operational semantics for CML we restrict this to one symbolic transition:

\[
(c?x:\{0,\ldots,9\} \rightarrow P(x)) \xrightarrow{c.w_0} P(w_0) \quad \text{for loose constant } w_0 \in \{0,\ldots,9\}
\]

Here, the fresh constant \(w_0\) records which of the 10 transitions actually takes place, and its value is loosely defined: it ranges over the set \(\{0,\ldots,9\}\). We can view these loose constants simply as values supplied when the rule is applied. In this example, \((w_0 = 5)\) would be a suitable choice. Or, more interestingly, we can see them as a collection of loose constants satisfying certain constraints. For example, consider the action

\[
c?x:\{0,\ldots,9\} \rightarrow d?y:\{0,\ldots,9\} \rightarrow (x=y) \& \text{Skip}
\]

This inputs a value through the channel \(c\), then inputs another value through the channel \(d\); if the values are equal, then the action terminates, otherwise it deadlocks. This has the following maximal trace, \(\langle c.w_0, d.w_1 \rangle\), which leads to a terminating execution state exactly when \((w_0 = w_1)\). CML’s symbolic traces and their constraints are a concise way of comprehending execution histories.
To understand the nature of a transition in CML, we need to understand a bit of the
denotational semantics of CML, which we quickly summarise here. The UTP semantics
embeds the theory of designs in the theory of reactive processes. Designs are relations with
preconditions and postconditions. They allow important observations: \( ok \) is true exactly
when the program has started and \( ok' \) is true exactly when the program has terminated.
Formally, a design is defined as

\[
P \vdash Q \equiv ok \land P \Rightarrow ok' \land Q
\]

Here, \( P \) is the precondition and \( Q \) is the postcondition. The definition is a statement of
total correctness: if the program is started in a state where \( P \) is true, then it will terminate
and when it does, \( Q \) will be true.

Reactive processes have three observations:

- \( wait' \) is true exactly when the process is waiting for interaction with its environment.
- \( tr' \) is the trace of visible events.
- \( ref' \) is the set of events being refused by the process.

The observations \( wait, tr, \) and \( ref \) are similar observations that may be made of a process’s
immediate predecessor.

There are two important healthiness conditions that link some of these variables

\[
\begin{align*}
R1(P) &= P \land tr \leq tr' \\
R3(P) &= (\exists \preceq wait \triangleright P)
\end{align*}
\]

The first says that the event trace is monotonic: once a event has occurred it cannot be
undone. The second makes sequential composition work: if a process is in its predecessor’s
wait-state, then it behaves like the identity; otherwise it has its own behaviour.

Now we can describe the states of a CML process. As additional to the operational seman-
tics for CSP, there is an imperative component, just like in the operational semantics for
the nondeterministic sequential programming language. This is a total assignment of con-
stants to the program variables (not the auxiliary observational variables). The constants
may include some of the loose constants mentioned above, and these may previously have
been constrained. A state configuration, therefore, is a triple: \((c \mid s \models A)\), where

- \( c \) is a text describing a constraint on loose constants.
- \( s \) is text describing a configuration state assignment.
- \( A \) is a text describing a CML action.

Note that this requires all variables to have a value.

When we compose the assignment and the action we must be careful to balance the alphabets: the assignment knows only about the program variables. We need to add the
observational variables and their healthiness conditions, both \( R1 \) and \( R3 \) and the healthi-
ness constraints on \( ok \) and \( ok' \) implicit in the nature of a design (see D23.3-1, Chapter 2).
We lift the syntactic assignment \( s \) to a reactive design as follows:

\[
\text{lift}(s) \equiv R1 \circ R3 (true \vdash (s_{+tr})_{ref} \land \neg wait')
\]
The alphabet of the assignment \( s \) has been lifted in two ways:

\[
(s_{+tr})^{\text{ref}} = s \land (tr' = tr)
\]
\[
\alpha((s_{+tr})^{\text{ref}}) = \alpha(s) \cup \{tr, ref, tr', ref'\}
\]

Now we are ready to define a transition corresponding to an invisible action.

\[
(c_1 | s_1 = A_1) \xrightarrow{\text{ref}} (c_2 | s_2 = A_2) \equiv \forall w \cdot c_1 \land c_2 \Rightarrow (\text{lift}(s_1) ; A_1 \subseteq \text{lift}(s_2) ; A_2)
\]

Here, \( w \) is the vector of loose constants. Note the similarity with the unlabelled transition in both the previous operational semantics (Sections 2.2 and 2.4). This definition says that however we satisfy the loose constants, \((\text{lift}(s_2) ; A_2)\) is an improvement (a refinement) on \((\text{lift}(s_1) ; A_1)\). Notice that we cannot ‘improve’ the loose constants themselves: in verifying the transition, we cannot choose values just to suit ourselves. Loose constants are not subject to refinement, otherwise we would be able to prove sound transitions that lose important behaviours.

The definition of a labelled transition follows this principle and is otherwise similar to that in CSP:

\[
(c_1 | s_1 = A_1) \xrightarrow{\text{ref}} (c_2 | s_2 = A_2)
\]

\[
\equiv \forall w \cdot c_1 \land c_2 \Rightarrow (\text{lift}(s_1) ; A_1 \subseteq (\text{lift}(s_2) ; c.w_1 \rightarrow A_2) \sqcap (\text{lift}(s_1) ; A_1))
\]

The use of loose constants raises the possibility of infeasibility if the constraints cannot be satisfied, so we require that all rules must be feasible in the following sense:

\[
\begin{array}{c}
\text{P} \\
(c_1 | s_1 = A_1) \xrightarrow{\text{ref}} (c_2 | s_2 = A_2)
\end{array}
\]

is feasible, providing that \((\exists w \cdot P \land c_1 \land c_2)\).

We illustrate the idea of feasibility by describing two of the imperative rules in CML: the ones for VDM operations defined by a precondition/postcondition pair. Suppose we have a VDM operation represented by the precondition \( p \) and the postcondition \( Q \) (here, \( p \) is a condition on the before state, whilst \( Q \) is a relation between before and after states). The first transition is applicable when the precondition \( p \) holds:

\[
(c | s = [p, Q]) \xrightarrow{\text{ref}} (c \land (c ; Q[w_0/v']) | s ; v := w_0 \Rightarrow \text{Skip})
\]

Suppose that we are in the configuration \((c | s = [p, Q])\). Here, the action to be executed is the design \([p, Q]\), the current state is \( s \) and the loose constants have to satisfy \( c \). There are four antecedents to the rule:

1. The constraint \( c \) must be valid.
2. The precondition must hold in the current state: \( s ; p \). This is the relational composition of the assignment \( s \) and the condition \( p \) and so it is a truth value. For example, \( x, y := 2, 4 ; (y = 2 * x) \) is true.
3. It must be possible to reach a final state satisfying \( Q \) starting from \( s \); that is, \( \exists v' \cdot s ; Q \). 
4. The vector \( v \) is the list of program variables: \((v = \text{out}(s))\).
So, providing that these four provisos hold, we can transit to a new state:

\[(c \text{ and } (c ; Q[w_0/x']) | s ; v := w_0) \models \text{Skip}\]

The postcondition may be nondeterministic, so the transition selects a fresh, loose constant \(w_0\) that it then uses as the target for the new assignment to the state: \((s ; v := w_0)\). If some of the program variables were declared during execution, then \(s\) will contain declarations and cannot be discarded. Having executed the design, the action is completed and becomes \(\text{Skip}\). The loose constant \(w_0\) must satisfy the postcondition, evaluated in the current state: \((c ; Q[w_0/x'])\). The previous constraint on the constants must continue to hold.

The second rule handles the case where the design is not applicable because the precondition cannot be satisfied.

\[
\begin{align*}
&c \not\models (s ; p) \\
\implies & (c | s \models [p, Q]) \overset{r}{\rightarrow} (c | s \models \text{Div})
\end{align*}
\]

The result of this execution is that the action now behaves like \(\text{Div}\).

To see the application of these rules consider the following example. Suppose that we have the VDM operation \(\text{Dec}\) on the sole programming variable \(x\), \(\text{nat}\) defined as \([x > 0, x' < x]\).

\[
\begin{align*}
&\text{true} | x := 2 \models \text{Dec} ; \text{Dec} ; \text{Dec} \\
&\overset{\rightarrow}{\rightarrow} (\text{true and } (x := 2 ; (x' < x) [w_0/x']) | x := w_0 \models \text{Skip} ; \text{Dec} ; \text{Dec}) \\
&= (w_0 \in \{0, 1\} | x := w_0 \models \text{Skip} ; \text{Dec} ; \text{Dec}) \\
&\overset{\rightarrow}{\rightarrow} (w_0 \in \{0, 1\} | x := w_0 \models \text{Dec} ; \text{Dec}) \\
&= (w_0 \in \{0, 1\} \text{ and } x := w_0 | x := w_1 \models \text{Skip} ; \text{Dec}) \\
&= (w_0 = 1 \text{ and } w_1 = 0 | x := w_1 \models \text{Skip} ; \text{Dec}) \\
&\overset{\rightarrow}{\rightarrow} (w_0 = 1 \text{ and } w_1 = 0 | x := w_1 \models \text{Dec}) \\
&\overset{\rightarrow}{\rightarrow} (w_0 = 1 \text{ and } w_1 = 0 | x := w_1 \models \text{Div})
\end{align*}
\]

We need to check the feasibility of this chain of transitions. Consider the design non-diverge transition:

\[
(w_0 \in \{0, 1\} | x := w_0 \models \text{Dec} ; \text{Dec}) \\
\overset{\rightarrow}{\rightarrow} (w_0 \in \{0, 1\} \text{ and } x := w_0 | x := w_1 \models \text{Skip} ; \text{Dec})
\]

with the antecedent

\[
w_0 \in \{0, 1\} \land (x := w_0 ; (x > 0)) \land (\exists x' \cdot x := w_0 ; x' < x)
\]

\[
= w_0 \in \{0, 1\} \land w_0 > 0 \land (\exists x' \cdot x' < w_0)
\]

\[
= w_0 \in \{0, 1\} \land w_0 > 0 \land (\exists x' \cdot x' < w_0)
\]

\[
= w_0 > 0 \land (\exists x' \cdot x' < w_0)
\]

\[
= w_0 > 0
\]

Feasibility requires that we can find values for both constants that make the before and after constraints true and that then satisfy the antecedent:

\[
\exists w_0, w_1 : \mathbb{N} \cdot (w_0 > 0) \land (x := w_0 ; (x' < x)[w_1/x'])
\]
\( \exists w_0, w_1 : \mathbb{N} \cdot (w_0 > 0) \land (w_1 < w_0) \)

\( \iff ((w_0 > 0) \land (w_1 < w_0))[1, 0/w_0, w_1] \)

\( = (1 > 0) \land (0 < 1) \)

\( = \text{true} \)

The design diverge transition is also feasible:

\[
(\begin{align*}
(\omega_0 \in \{0, 1\}) & \mid x := \omega_0 \models \text{Dec} \sqcup \text{Dec} \\
\rightarrow (\omega_0 \in \{0, 1\}) & \land (x := \omega_0 \mid \neg \text{pre Dec}) \mid x := \omega_0 \models \text{Div} \sqcup \text{Dec}
\end{align*})
\]

We finish this section by describing the transition rules for the external choice operator. The operational interpretation of external choice is that both operands are run in parallel until something observable occurs to choose between them; this is either an observable event or when one or the other terminates. We use a syntactic device in the process text to mark the fact that the parallel execution is taking place. This purely syntactic device is called “extra choice” and is denoted by \( P [+] Q \). It allows us to record two parallel copies of the state evolving separately until the choice is resolved. There are four rules for external choice, the first duplicates the state and initiates the extra choice operator:

\[
(c \mid s \models A_1 \{\} A_2) \xrightarrow{\tau} (c \mid s \models (\text{loc } c \mid s @ A_1) [+] (\text{loc } c \mid s @ A_2))
\]

The second transition rules allows independent invisible actions to take place without resolving the external choice.

\[
\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{\tau} (c_2 \mid s_3 \models A_3)}{(c_1 \mid s \models (\text{loc } s_1 @ A_1) [+] (\text{loc } s_2 @ A_2)) \xrightarrow{\tau} (c_2 \mid s \models (\text{loc } s_3 @ A_3) [+] (\text{loc } s_2 @ A_2))}
\]

The third transition rule takes place when one of the operands terminates.

\[
(c \mid s \models (\text{loc } s_1 @ \text{Skip}) [+] (\text{loc } s_2 @ A)) \xrightarrow{\tau} (c \mid s \models \text{Skip})
\]

Finally, the choice may be resolved by a visible event.

\[
\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{\tau} (c_3 \mid s_3 \models A_3) \quad l \neq \tau}{(c \mid s \models (\text{loc } s_1 @ A_1) [+] (\text{loc } s_2 @ A_2)) \xrightarrow{\tau} (c_3 \mid s_3 \models A_3)}
\]

The same kind of syntactic device is used in the transition rule for parallel composition to maintain two state partitions that get merged once the two processes have terminated.
Chapter 3

Operational Semantics for CML

3.1 Denotational Semantics

We present an indexed collection of laws that were used by hand to prove the soundness of the operational semantics. The naming scheme to index the laws is:

\[
\begin{align*}
\text{index} & ::= \langle \text{kind} \rangle : -(\text{theory}) : -(\text{subindex}) \\
\text{kind} & ::= \text{definition} \mid \text{law} \mid \text{rule} \mid \text{theorem} \\
\text{theory} & ::= \text{propositional-calculus} \mid \text{predicate-calculus} \mid \text{relational-calculus} \mid \ldots
\end{align*}
\]

3.1.1 definition::-reactive-design::Skip

Law 3.1.1 (definition::-reactive-design::Skip)

\[
\text{Skip } = R(\text{true} \vdash (v' = v)_+ \land \lnot \text{wait'})
\]

3.1.2 definition::-reactive-design::assignment

Definition 3.1.1 (definition::-reactive-design::assignment)

\[
v :=_{RD} e \triangleq \text{lift}(v := e) ; \text{Skip}
\]

3.1.3 definition::-reactive-design::sequence

Definition 3.1.2 (definition::-reactive-design::sequence)

\[
R_1 \circ R_3(P \vdash Q) ; R_1 \circ R_3(R \vdash S) = R_1 \circ R_3(P_{wp}R_1R_3_{\text{pre}}(R) \vdash Q ; R_3_{\text{post}}(S))
\]

3.1.4 definition::-reactive-design::simple-prefix

Definition 3.1.3 (definition::-reactive-design::simple-prefix)

\[
l \rightarrow \text{Skip } = R_1 \circ R_3(\text{true} \vdash ((tr' = tr) \land a \notin \text{ref'} \land \text{wait'} \supset tr' = tr \odot (a))_+)
\]
3.1.5 **definition:::-reactive-design::prefix**

Definition 3.1.4 (**definition:::-reactive-design::prefix**)

\[ a \rightarrow P \triangleq (a \rightarrow \text{Skip}) \land P \]

3.1.6 **definition:::-reactive-design::nondeterministic-choice**

Definition 3.1.5 (**definition:::-reactive-design::nondeterministic-choice**)

\[ P \sqcap Q \triangleq P \lor Q \]

3.1.7 **definition:::-reactive-design::external-choice**

Definition 3.1.6 (**definition:::-reactive-design::external-choice**)

\[ P \sqcup Q \triangleq R1 \circ R3(\neg P_i \land Q_j) \land P_i \land \neg Q_j \land (tr' = tr) \land \text{wait} \lor P_i \lor Q_j \]

3.1.8 **definition:::-reactive-design::external-choice::distributed**

Law 3.1.2 (**definition:::-reactive-design::external-choice::distributed**)

\[ \Box S \triangleq R1 \circ R3(\forall X : S \lor X \lor X : S \bullet X \bullet X \land (tr' = tr) \land \text{wait} \lor \exists X : S \bullet X) \]

3.1.9 **definition:::-reactive-design::parallel::alphabetised-partitioned**

Law 3.1.3 (**definition:::-reactive-design::parallel::alphabetised-partitioned**)

\[
\begin{align*}
P \parallel [x_1 | cs | x_2] & \parallel Q \\
\Rightarrow & \circ R1 \circ R3(\neg (P_i \parallel Q_j) \land \neg (P_i \parallel Q_j) \\
\lor & (P_i \parallel Q_j))
\end{align*}
\]

3.1.10 **theorem:::-parallel-reactive-design**

Theorem 3.1.1 (**theorem:::-parallel-reactive-design**)

\[
\begin{align*}
P \parallel_{\text{CSP}} Q & = (P_i ; \text{end ok}) \parallel M_{\text{CSP}2} (Q_j ; \text{end ok}) \land \neg ((P_i ; \text{end ok}) \parallel M_{\text{CSP}2} (Q_j ; \text{end ok})) \\
& \lor (P_i \parallel M_{\text{ok}} Q_j))
\end{align*}
\]

*Proof*
\[ \begin{align*}
P \parallel Q \\
= \{ \text{A.11.13 (law::reactive-design)} \} \\
R(\neg (P \parallel Q)^f_f \vdash (P \parallel Q)^f_f) \\
= \{ \text{A.13.34 (law::CML::parallel-pre)} \} \\
R(\neg ((P^f_f \parallel_{M, \text{CSP} \neg \text{ok}} Q^f_f) \lor (P^t_t \parallel_{M, \text{CSP} \neg \text{ok}} Q^t_t)) \\
\vdash (P \parallel Q)^f_f \\
\} \\
= \{ \text{A.13.33 (law::CML::parallel-post)} \} \\
R(\neg ((P^f_f \parallel_{M, \text{CSP} \neg \text{ok}} Q^f_f) \lor (P^t_t \parallel_{M, \text{CSP} \neg \text{ok}} Q^t_t)) \\
\vdash (P^f_f \parallel_{M, \text{ok}} Q^f_f) \\
\} \\
= \{ \text{A.1.19 (law::propositional-calculus::negation::De-Morgan)} \} \\
R(\neg ((P^f_f \parallel_{M, \text{CSP} f} Q^f_f) \land \neg (P^t_t \parallel_{M, \text{CSP} f} Q^t_t)) \\
\vdash (P^f_f \parallel_{M, \text{ok}} Q^f_f) \\
\}
\]
3.2 Operational Semantics

3.2.1 configuration

3.2.1.1 definition::-configuration::-state-assignment

Definition 3.2.1 (definition::-configuration::-state-assignment) A configuration state assignment \( s \) is a total function mapping the programming variables in the alphabet to (possibly loose) constants. Thus, for programming variables \( v \), although \( (\alpha(s) = \{v, v'\}) \), \( v \) is not free in \( s \).

3.2.1.2 law::-configuration::-assignment::-conjunctive

Law 3.2.1 (law::-configuration::-assignment::-conjunctive)

\[
s ; P \land Q = (s ; P) \land (s ; Q)
\]

3.2.1.3 law::-configuration::-leading-assignment

Law 3.2.2 (law::-configuration::-leading-assignment)

\[
s_{+\text{tr@ref}} ; P = P[s]
\]

3.2.1.4 law::-configuration::-state-idempotence::-inner

Law 3.2.3 (law::-configuration::-state-idempotence::-inner)

\[
s ; (s ; (w_1 = e)) \land P = s ; (w_1 = e) \land P
\]

3.2.1.5 law::-configuration::-state-idempotence

Law 3.2.4 (law::-configuration::-state-idempotence)

\[
s ; t = t
\]

3.2.2 transition

3.2.2.1 definition::-transition::-tau

Definition 3.2.2 (definition::-transition::-tau)

\[
(c_1 \mid s_1 \models A_1) \tau\ (c_2 \mid s_2 \models A_2) \equiv \forall w \cdot c_1 \land c_2 \Rightarrow (\text{lift}(s_1) ; A_1 \subseteq \text{lift}(s_2) ; A_2)
\]
3.2.2.2  definition::-transition:-labelled

Definition 3.2.3 (definition::-transition:-labelled)

\[ (c_1 \mid s_1 \models A_1) \xrightarrow{c.w_1} (c_2 \mid s_2 \models A_2) \]
\[ \equiv \forall w \cdot c_1 \land c_2 \Rightarrow lift(s_1) ; A_1 \sqsubseteq (lift(s_2) ; c.w_1 \to A_2) \square (lift(s_1) ; A_1) \]

3.2.2.3  definition::-transition:-feasibility

Definition 3.2.4 (definition::-transition:-feasibility)

\[ (c_1 \mid s_1 \models A_1) \xrightarrow{\lambda} (c_2 \mid s_2 \models A_2) \]

is feasible, providing that there exists \( w \) such that \( c_1 \) and \( c_2 \)

3.2.3  CML-transition

3.2.3.1  rule::-cml-transition:-assignment

Rule 3.2.1 (rule::-cml-transition:-assignment)

\[ \frac{c \cdot s ; (w_0 = e)}{(c \mid s \models v := e) \xrightarrow{\tau} (c \text{ and } (s ; (w_0 = e)) \mid s ; v := w_0 \models \text{Skip})} \]

3.2.3.2  rule::-cml-transition:-design:-nondiverge

Rule 3.2.2 (rule::-cml-transition:-design:-nondiverge)

\[ \frac{c \cdot s ; p \quad \exists v' \cdot s ; Q \quad v = out\alpha(s)}{(c \mid s \models [p, Q]) \xrightarrow{\tau} (c \text{ and } (s ; Q[w_0 / v']) \mid s ; v := w_0 \models \text{Skip})} \]

3.2.3.3  rule::-cml-transition:-design:-diverge

Rule 3.2.3 (rule::-cml-transition:-design:-diverge)

\[ \frac{c \not\vdash (s ; p)}{(c \mid s \models [p, Q]) \xrightarrow{\tau} (c \mid s \models \text{Div})} \]

3.2.3.4  rule::-cml-transition:-Div

Rule 3.2.4 (rule::-cml-transition:-Div)

\[ \frac{c}{(c \mid s \models \text{Div}) \xrightarrow{\tau} (c \mid s \models \text{Div})} \]
3.2.3.5 rule::-cml-transition::input

Rule 3.2.5 (rule::-cml-transition::input)

\[ T \neq \emptyset \quad x \notin \alpha(s) \]

\[(c \mid s \models d?x, T \rightarrow A) \xrightarrow{\text{d}x_0} (c \land w_0 \in T \mid s ; \text{var} \ x := w_0 \models \text{let} \ x @ A)\]

3.2.3.6 rule::-cml-transition::output

Rule 3.2.6 (rule::-cml-transition::output) for \( w_0 \) not occurring in \( c, s, e, \) or \( A \)

\[ c \quad s ; (w_0 = e) \]

\[(c \mid s \models d!e \rightarrow A) \xrightarrow{\text{d}w_0} (c \land (s ; (w_0 = e) \mid s \models A)\]

3.2.3.7 rule::-cml-transition::variable-block::begin

Rule 3.2.7 (rule::-cml-transition::variable-block::begin)

\[ c \quad T \neq \emptyset \quad x \notin \alpha(s) \]

\[(c \mid s \models \text{var} \ x, T \rightarrow A) \xrightarrow{T} (c \land w_0 \in T \mid s ; \text{var} \ x := w_0 \models \text{let} \ x @ A)\]

3.2.3.8 rule::-cml-transition::variable-block::visible

Rule 3.2.8 (rule::-cml-transition::variable-block::visible)

\[(c_1 \mid s_1 \models A_1) \xrightarrow{1} (c_2 \mid s_2 \models A_2) \]

\[(c_1 \mid s_1 \models \text{let} \ x @ A_1) \xrightarrow{1} (c_2 \mid s_2 \models \text{let} \ x @ A_2)\]

3.2.3.9 rule::-cml-transition::variable-block::end

Rule 3.2.9 (rule::-cml-transition::variable-block::end)

\[ c \]

\[(c \mid s \models \text{let} \ x @ \text{Skip}) \xrightarrow{T} (c \mid s ; \text{end}x \models \text{Skip})\]

3.2.3.10 rule::-cml-transition::sequence::progress

Rule 3.2.10 (rule::-cml-transition::sequence::progress)

\[(c_1 \mid s_1 \models A_1) \xrightarrow{1} (c_2 \mid s_2 \models A_2) \]

\[(c_1 \mid s_1 \models A_1 \; B) \xrightarrow{1} (c_2 \mid s_2 \models A_2 \; B)\]
3.2.3.11  \textit{rule::-cml-transition:-sequence:-end}

Rule 3.2.11 (\textit{rule::-cml-transition:-sequence:-end})

\[
\frac{c}{(c \mid s \vdash \text{Skip} ; A) \xrightarrow{\tau} (c \mid s \vdash A)}
\]

3.2.3.12  \textit{rule::-cml-transition:-nondeterministic-choice}

Rule 3.2.12 (\textit{rule::-cml-transition:-nondeterministic-choice})

\[
\frac{c}{(c \mid s \mid = A_1 \mid \sim A_2) \xrightarrow{\tau} (c \mid s \vdash A_1)}
\]

3.2.3.13  \textit{rule::-cml-transition:-guard}

Rule 3.2.13 (\textit{rule::-cml-transition:-guard})

\[
\frac{c \quad s ; g}{(c \mid s \vdash g \& A) \xrightarrow{\tau} (c \text{ and } (s ; g) \mid s \vdash A)}
\]

3.2.3.14  \textit{rule::-cml-transition:-external-choice:-begin}

Rule 3.2.14 (\textit{rule::-cml-transition:-external-choice:-begin})

\[
\frac{(c \mid s \vdash A_1 \mid A_2) \xrightarrow{\tau} (c \mid s \vdash (\text{loc } c \mid s @ A_1) [+] (\text{loc } c \mid s @ A_2))}{(c \mid s \vdash A_1 \mid A_2) \xrightarrow{\tau} (c \mid s \vdash (\text{loc } c \mid s @ A_1) [+] (\text{loc } c \mid s @ A_2))}
\]

3.2.3.15  \textit{rule::-cml-transition:-external-choice:-end}

Rule 3.2.15 (\textit{rule::-cml-transition:-external-choice:-end})

\[
\frac{(c_1 \mid s_1 \vdash A_1) \xrightarrow{1} (c_3 \mid s_3 \vdash A_3) \quad l \neq \tau}{(c \mid s \vdash (\text{loc } s_1 @ A_1) [+] (\text{loc } s_2 @ A_2)) \xrightarrow{l} (c_3 \mid s_3 \vdash A_3)}
\]

3.2.3.16  \textit{rule::-cml-transition:-external-choice:-silent}

Rule 3.2.16 (\textit{rule::-cml-transition:-external-choice:-silent})

\[
\frac{(c_1 \mid s_1 \vdash A_1) \xrightarrow{1} (c_2 \mid s_3 \vdash A_3)}{(c_1 \mid s \vdash (\text{loc } s_1 @ A_1) [+] (\text{loc } s_2 @ A_2)) \xrightarrow{l} (c_2 \mid s \vdash (\text{loc } s_3 @ A_3) [+] (\text{loc } s_2 @ A_2))}
\]
3.2.3.17  **rule::cml-transition::external-choice::Skip**

Rule 3.2.17 (**rule::cml-transition::external-choice::Skip**)

\[(c \mid s \models ( \text{loc} \ s_1 @ \text{Skip}) \mathbin{[+]} ( \text{loc} \ s_2 @ \text{A}) ) \xrightarrow{\tau} (c \mid s_1 \models \text{Skip})\]

3.2.3.18  **rule::cml-transition::parallel::begin**

Rule 3.2.18 (**rule::cml-transition::parallel::begin**)

\[(\text{out}(\alpha(s)) = \{ x'_1, x'_2, x'_3 \})\]

\[(c \mid s \models A_1 [ \mid x_1 \mid cs \mid x_2 \mid] A_2) \xrightarrow{\tau} (c \mid s \models (\text{loc} \ (s \mid x_1) + x_2 @ A_1) [ \mid x_1 \mid cs \mid x_2 \mid] (\text{loc} \ (s \mid x_2) + x_1 @ A_2))\]

3.2.3.19  **rule::cml-transition::parallel::independent**

Rule 3.2.19 (**rule::cml-transition::parallel::independent**)

\[(c \mid s \models A_1 \mid x_1 \mid cs \mid x_2 \mid A_2) \xrightarrow{\tau} (c_3 \mid s \models (\text{loc} \ s_3 \bullet A_3) [ \mid x_1 \mid cs \mid x_2 \mid] (\text{loc} \ s_2 @ A_2))\]

3.2.3.20  **rule::cml-transition::parallel::synchronised**

Rule 3.2.20 (**rule::cml-transition::parallel::synchronised**)

\[d \in cs \quad c_1 \quad c_2 \quad c_3 \quad c_4 \quad (w_1 = w_2) \quad (\alpha, \omega) \in \{ (\varnothing, \varnothing), (\varnothing, \omega), (\omega, \varnothing) \}\]

\[(c_1 \mid s_1 \models A_1) \xrightarrow{d_{\omega_2}} (c_3 \mid s_3 \models A_3) \quad (c_2 \mid s_2 \models A_2) \xrightarrow{d_{\omega_2}} (c_4 \mid s_4 \models A_4)\]

\[(c_1 \text{ and } c_2 \mid s \models (\text{loc} \ c_1 \mid s_1 @ A_1) [ \mid x_1 \mid cs \mid x_2 \mid] (\text{loc} \ c_2 \mid s_2 @ A_2))\]

\[\xrightarrow{d_{\omega_2}} (c_3 \wedge c_4 \wedge (w_1 = w_2) \mid s \models (\text{loc} \ c_3 \text{ and } (w_1 = w_2) \mid s_3 @ A_3) [ \mid x_1 \mid cs \mid x_2 \mid] (\text{loc} \ c_4 \text{ and } (w_1 = w_2) \mid s_4 @ A_4))\]

3.2.3.21  **rule::cml-transition::parallel::end**

Rule 3.2.21 (**rule::cml-transition::parallel::end**)

\[(\text{out}(\alpha(s)) = \{ x'_1, x'_2, m' \})\]

\[(c \mid s \models (\text{loc} \ s_1 @ \text{Skip}) [ \mid x_1 \mid cs \mid x_2 \mid] (\text{loc} \ s_2 @ \text{Skip}))\]

\[(c \mid (s_1 \mid x_1 \text{ and } (s_2 \mid x_2) \models \text{Skip})\]

32
3.2.4 lift

3.2.4.1 definition:-lift

Definition 3.2.5 (definition:-lift)

\[
\text{lift}(s) \triangleq R1 \circ R3[\text{true} \vdash s_{+\text{tr@ref}} \land \neg \text{wait}']
\]

3.2.4.2 law:-lift:-composition

Law 3.2.5 (law:-lift:-composition)

\[
\text{lift}(s); \text{lift}(t) = \text{lift}(s; t)
\]

Proof

\[
lift(s); lift(t) \\
= \{ 3.2.5 (definition:-lift) \} \\
lift(s); R1 \circ R3[\text{true} \vdash t_{+\text{tr@ref}} \land \neg \text{wait}'] \\
= \{ 3.2.11 (law:-lift:-leading:-reactive-design) \} \\
R1 \circ R3[\text{true}[s] \vdash (t_{+\text{tr@ref}} \land \neg \text{wait'})[s]] \\
= \{ A.7.6 (law:-substitution) \} \\
R1 \circ R3[\text{true} \vdash t[s]_{+\text{tr@ref}} \land \neg \text{wait}'] \\
= \{ A.13.16 (law:-CML:-assignment:-composition) \} \\
R1 \circ R3[\text{true} \vdash (s + t)_{+\text{tr@ref}} \land \neg \text{wait}'] \\
= \{ 3.2.5 (definition:-lift), (s + t) is an assignment \} \\
lift(s; t)
\]

\]

3.2.4.3 law:-lift:-CSP4

Law 3.2.6 (law:-lift:-CSP4)

Proof

\[
lift(v := e); \text{Skip} \\
= \{ A.13.9 (law:-CML:-Skip:-lifted-assignment) \} \\
lift(v := e); lift(v := v) \\
= \{ 3.2.5 (definition:-lift) \} \\
R1 \circ R3[\text{true} \vdash (v := e)_{+\text{tr@ref}} \land \neg \text{wait'}]; R1 \circ R3[\text{true} \vdash (v := v)_{+\text{tr@ref}} \land \neg \text{wait'}] \\
= \{ A.11.11 (law:-reactive-design:-sequence:-R1-R3) \} \\
R1 \circ R3(\]

33
\[
\neg (R_1(\neg \text{true}) \land R_1(\text{true})) \land \neg (R_1((v := e)_{+\text{tr@ref}} \land \neg \text{wait}')) \land \neg \text{wait}'; R_1(\neg \text{true}))
\]
\[
\vdash R_1((v := e)_{+\text{tr@ref}} \land \neg \text{wait}'); (\Pi \triangleleft \text{wait} \triangleright R_1((v := e)_{+\text{tr@ref}} \land \neg \text{wait}'))
\]
\[
= \{ \text{A.1.16 (law::propositional-calculus::negation::true), twice } \}
\]
\[
R_1 \circ R_3(\neg (R_1(\text{false}) \land R_1(\text{true})) \land \neg (R_1((v := e)_{+\text{tr@ref}} \land \neg \text{wait}') \land \neg \text{wait}'; R_1(\text{false})))
\]
\[
\vdash R_1((v := e)_{+\text{tr@ref}} \land \neg \text{wait}'); (\Pi \triangleleft \text{wait} \triangleright R_1((v := e)_{+\text{tr@ref}} \land \neg \text{wait}'))
\]
\[
= \{ \text{A.10.12 (law::reactive::R1::false) } \}
\]
\[
R_1 \circ R_3(\neg \text{false} \land \neg \text{false})
\]
\[
\vdash R_1((v := e)_{+\text{tr@ref}} \land \neg \text{wait}'); (\Pi \triangleleft \text{wait} \triangleright R_1((v := e)_{+\text{tr@ref}} \land \neg \text{wait}'))
\]
\[
= \{ \text{A.1.15 (law::propositional-calculus::negation::false), twice } \}
\]
\[
R_1 \circ R_3(\text{true} \land \text{true})
\]
\[
\vdash R_1((v := e)_{+\text{tr@ref}} \land \neg \text{wait}'); (\Pi \triangleleft \text{wait} \triangleright R_1((v := e)_{+\text{tr@ref}} \land \neg \text{wait}'))
\]
\[
= \{ \text{A.1.6 (law::propositional-calculus::and::unit) } \}
\]
\[
R_1 \circ R_3(\text{true} \vdash R_1((v := e)_{+\text{tr@ref}} \land \neg \text{wait}'); (\Pi \triangleleft \text{wait} \triangleright R_1((v := e)_{+\text{tr@ref}} \land \neg \text{wait}'))
\]
\[
= \{ \text{A.10.9 (law::reactive::R1::detach), twice } \}
\]
\[
R_1 \circ R_3(\text{true} \vdash (v := e)_{+\text{tr@ref}} \land \neg \text{wait}'; (\Pi \triangleleft \text{wait} \triangleright ((v := e)_{+\text{tr@ref}} \land \neg \text{wait}')))
\]
\[
= \{ \text{A.5.22 (law::relational-calculus::sequence::one-point::left) } \}
\]
\[ \text{Proof} \]

\[ R1 \circ R3 \vdash_{\text{true}} \]

\[ (v := e)_{+tr_{\text{ref}}} [\text{false}/wait'] ; \]
\[ (\Pi \lhd \text{wait} \triangleright ((v := v)_{+tr_{\text{ref}}} \land \neg \text{wait}'))[false/\text{wait}] \]

\[ = \{ \text{A.7.6 (law::substitution), twice} \} \]

\[ \text{A.1.35 (law::propositional-calculus::conditional::constant::false)} \]

\[ R1 \circ R3 \vdash_{\text{true}} (v := e)_{+tr_{\text{ref}}} ; (v := v)_{+tr_{\text{ref}}} \land \neg \text{wait}') \]

\[ = \{ \text{A.5.15 (law::relational-calculus::sequence::detach-post)} \} \]

\[ R1 \circ R3 \vdash_{\text{true}} ((v := e)_{+tr_{\text{ref}}} ; (v := v)_{+tr_{\text{ref}}} \land \neg \text{wait}') \]

\[ = \{ \text{A.4.4 (law::alphabet::lifting::sequence)} \} \]

\[ R1 \circ R3 \vdash_{\text{true}} (v := e ; v := v)_{+tr_{\text{ref}}} \land \neg \text{wait}') \]

\[ = \{ \text{A.5.4 (law::relational-calculus::assignment::sequence::identity)} \} \]

\[ R1 \circ R3 \vdash_{\text{true}} (v := e)_{+tr_{\text{ref}}} \land \neg \text{wait}') \]

\[ = \{ \text{3.2.5 (definition::lift)} \} \]

\[ \text{lift}(v := e) \]

\[ \square \]

3.2.4.4 \textit{law::lift::external-choice} \]

Law 3.2.7 \textit{(law::lift::external-choice)}

\[ \text{lift}(s) ; (P \Box Q) = (\text{lift}(s) ; P) \Box (\text{lift}(s) ; Q) \]

\[ \text{Proof} \]

\[ \text{lift}(s) ; (P \Box Q) \]

\[ = \{ \text{3.1.6 (definition::reactive-design::external-choice)} \} \]

\[ \text{lift}(s) ; R1 \circ R3 \vdash_{\text{!P f} \land \neg Q f} P f \land Q f \land ((tr' = tr) \land \text{wait}' \triangleright P f \lor Q f) \]

\[ = \{ \text{3.2.10 (law::lift::leading::substitution)} \} \]

\[ R1 \circ R3 \vdash_{\text{!P f} \land \neg Q f[s]} (P f \land Q f[s]) \land ((tr' = tr) \land \text{wait}' \triangleright P f \lor Q f)[s] \]

\[ = \{ \text{A.7.6 (law::substitution)} \} \]

\[ R1 \circ R3 \vdash_{\text{!P f} \land \neg Q f[s]} P f[s] \land Q f[s] \land ((tr' = tr) \land \text{wait}' \triangleright P f \lor Q f)[s] \]

\[ = \{ \text{A.7.6 (law::substitution) (v not free in (tr' = tr) \land \text{wait}') \} \}

\[ R1 \circ R3 \vdash_{\text{!P f} \land \neg Q f[s]} P f[s] \land Q f[s] \land ((tr' = tr) \land \text{wait}' \triangleright P f \lor Q f)[s] \]

\[ = \{ \text{3.1.6 (definition::reactive-design::external-choice)} \} \]

\[ P f[s] \Box Q f[s] \]

\[ = \{ \text{3.2.10 (law::lift::leading::substitution), twice} \} \]

\[ (\text{lift}(s) ; P) \Box (\text{lift}(s) ; Q) \]

\[ \square \]

35
3.2.4.5  \textit{law::lift::left-unit}

Law 3.2.8 (\textit{law::lift::left-unit})
\[ \text{lift}(s) ; \text{lift}(t) = \text{lift}(t) \]

3.2.4.6  \textit{law::lift::leading::conjunctive}

Law 3.2.9 (\textit{law::lift::leading::conjunctive})
\[ \text{lift}(s) ; P \land Q = (\text{lift}(s) ; P) \land (\text{lift}(s) ; P) \]

Proof

\[ \text{lift}(s) ; P \land Q = \{ 3.2.11 \ (\textit{law::lift::leading::reactive-design}) \} (P \land Q)[s] = \{ A.7.6 \ (\textit{law::substitution}) \} P[s] \land Q[s] = \{ 3.2.11 \ (\textit{law::lift::leading::reactive-design}), \textit{twice} \} (\text{lift}(s) ; P) \land (\text{lift}(s) ; P) \]

\[
\]

3.2.4.7  \textit{law::lift::leading::substitution}

Law 3.2.10 (\textit{law::lift::leading::substitution})
\[ \text{lift}(s) ; P = \text{lift}(s) ; P[e/w] \quad \text{providing } (s ; (w = e)) \]

Proof

\[ \text{lift}(s) ; P = \{ 3.2.11 \ (\textit{law::lift::leading::reactive-design}) \} P[s] = \{ \textit{assumption}: (s ; (w = e)) \} (s ; (w = e)) \land P[s] = \{ A.13.19 \ (\textit{law::CML::assignment::substitution}) \} (w = e[s]) \land P[s] = \{ A.7.6 \ (\textit{law::substitution}) \} ((w = e) \land P)[s] = \{ A.3.2 \ (\textit{law::equals::Leibniz}) \} \]
\[(w = e) \land P[e/w][s] = \{\ A.7.6 \text{ (law::substitution)} \}\]

\[(w = e[s]) \land P[e/w][s] = \{\ A.13.19 \text{ (law::CML:-assignment:-substitution)} \}\]

\[(s ; (w = e)) \land P[e/w][s] = \{\ \text{assumption:} (s ; (w = e)) \}\]

\[P[e/w][s] = \{\ 3.2.11 \text{ (law::lift:-leading:-reactive-design)} \}\]

\[
\text{lift}(s) ; P[e/w]
\]

\[\checkmark\]

3.2.4.8 \textit{law::lift:-leading:-reactive-design}

Law 3.2.11 \textit{(law::lift:-leading:-reactive-design)}

\[
\text{lift}(s) ; R1 \circ R3(P \vdash Q) = R1 \circ R3(P[s] \vdash Q[s])
\]

Proof

\[
\text{lift}(s) ; R1 \circ R3(P \vdash Q)
\]

\[= \{\ 3.2.5 \text{ (definition::lift)} \}\]

\[R1 \circ R3(\text{true} \vdash s_{+tr\ref} \land \neg wait') ; R1 \circ R3(P \vdash Q) = \{\ 3.1.2 \text{ (definition::reactive-design:-sequence)} \}\]

\[R1 \circ R3((s_{+tr\ref} \land \neg wait') \text{wp} R1 \circ R3 \text{pre}(P) \vdash s_{+tr\ref} \land \neg wait' ; R3 \text{post}(Q)) = \{\ A.5.14 \text{ (law::relational-calculus:-sequence:-condition-swing), twice} \}\]

\[R1 \circ R3((s_{+tr\ref} \land \neg wait') \text{wp} R1 \circ R3 \text{pre}(P) \vdash s_{+tr\ref} ; Q_f) = \{\ A.5.14 \text{ (law::relational-calculus:-sequence:-condition-swing)} \}\]

\[R1 \circ R3((s_{+tr\ref} \land \neg wait') \text{wp} R1 \circ R3 \text{pre}(P) \vdash s_{+tr\ref} ; Q_f) = \{\ A.5.14 \text{ (law::relational-calculus:-sequence:-condition-swing)} \}\]

\[
R1 \circ R3((\neg (s_{+tr\ref} \land \neg wait') ; R1(\neg R3 \text{pre}(P))) \vdash s_{+tr\ref} ; Q_f) = \{\ A.5.14 \text{ (law::relational-calculus:-sequence:-condition-swing)} \}\]

\[R1 \circ R3((\neg (s_{+tr\ref} \land \neg wait') ; R1(\neg R3 \text{pre}(P))) \vdash s_{+tr\ref} ; Q_f) = \{\ A.5.14 \text{ (law::relational-calculus:-sequence:-condition-swing)} \}\]

\[R1 \circ R3((\neg (s_{+tr\ref} \land \neg wait') ; R1(\neg R3 \text{pre}(P))) \vdash s_{+tr\ref} ; Q_f) = \{\ A.5.14 \text{ (law::relational-calculus:-sequence:-condition-swing)} \}\]

\[R1 \circ R3((\neg (s_{+tr\ref} \land \neg wait') ; R1(\neg R3 \text{pre}(P))) \vdash s_{+tr\ref} ; Q_f) = \{\ A.5.14 \text{ (law::relational-calculus:-sequence:-condition-swing)} \}\]

\[R1 \circ R3(P[s] \vdash Q[s]) = \{\ 3.2.2 \text{ (law::configuration:-leading-assignment), twice} \}\]

\[R1 \circ R3(P[s] \vdash Q[s]) = \{\ A.10.1 \text{ (law::reactive:-R1:-design:-pre-cancellation)} \}\]

\[R1 \circ R3(P[s] \vdash Q[s]) = \{\ A.1.20 \text{ (law::propositional-calculus:-negation:-double-negation)} \}\]

\[R1 \circ R3(P[s] \vdash Q[s]) = \{\ A.10.1 \text{ (law::reactive:-R1:-design:-pre-cancellation)} \}\]

\[R1 \circ R3(P[s] \vdash Q[s]) = \{\ A.1.20 \text{ (law::propositional-calculus:-negation:-double-negation)} \}\]

\[R1 \circ R3(P[s] \vdash Q[s]) = \{\ A.10.1 \text{ (law::reactive:-R1:-design:-pre-cancellation)} \}\]

\[R1 \circ R3(P[s] \vdash Q[s]) = \{\ A.1.20 \text{ (law::propositional-calculus:-negation:-double-negation)} \}\]

\[R1 \circ R3(P[s] \vdash Q[s]) = \{\ A.10.1 \text{ (law::reactive:-R1:-design:-pre-cancellation)} \}\]

\[R1 \circ R3(P[s] \vdash Q[s]) = \{\ A.1.20 \text{ (law::propositional-calculus:-negation:-double-negation)} \}\]
3.2.4.9  \textit{law::-lift:-merge}

Law 3.2.12 (\textit{law::-lift:-merge})

\begin{align*}
lift(s_1) [ x_1 | cs | x_2 ] \ lift(s_2) &= \ lift((s_1 | x_1) \land (s_2 | x_2))
\end{align*}

3.2.4.10  \textit{law::-lift:-parallel:-distributivity}

Law 3.2.13 (\textit{law::-lift:-parallel:-distributivity})

\begin{align*}
lift(s) ; (P [ x_1 | cs | s_2 ] Q) = (lift(s) ; P) [ x_1 | cs | x_2 ] (lift(s) ; Q)
\end{align*}

\textbf{Proof}

\begin{align*}
lift(s) ; (P [ x_1 | cs | x_2 ] Q) \\
&= \{ \text{3.1.3 (definition::-reactive-design:-parallel:-alphabetised-partitioned)} \} \\
lift(s) ; \\
R1 \circ R3(\neg (P f_f \parallel_{M_f} Q_f) \land \neg (P f_f \parallel_{M_f} Q_f)) \\
\vdash \\
P_f_f \parallel_{M_f} Q_f) \\
&= \{ \text{A.11.6 (law::-reactive-design::lift)} \} \\
R1 \circ R3((\neg (P f_f \parallel_{M_f} Q_f) \land \neg (P f_f \parallel_{M_f} Q_f)))[s] \\
\vdash \\
(P_f_f \parallel_{M_f} Q_f)[s] \\
&= \{ \text{3.2.11 (law::-lift-leading:-reactive-design)} \} \\
R1 \circ R3(\neg (P f_f \parallel_{M_f} Q_f)[s] \land \neg (P f_f \parallel_{M_f} Q_f)[s]) \\
\vdash \\
(P_f_f \parallel_{M_f} Q_f)[s] \\
&= \{ \text{A.13.38 (law::-CML:-parallel:-substitution), three times} \} \\
R1 \circ R3(\neg (P[s]_f f_f \parallel_{M_f} Q[s]_f) \land \neg (P[s]_f f_f \parallel_{M_f} Q[s]_f)) \\
\vdash \\
(P[s]_f f_f \parallel_{M_f} Q[s]_f) \\
&= \{ \text{A.11.7 (law::-reactive-design::parallel-by-merge)} \} \\
R1 \circ R3(\neg (P[s]_f f_f) \vdash P[s]_f f_f) \parallel_M R1 \circ R3(\neg (Q[s]_f f_f) \vdash Q[s]_f f_f) \\
&= \{ \text{3.2.11 (law::-lift-leading:-reactive-design), twice} \} \\
lift(s) ; R1 \circ R3(\neg (P f_f) \vdash P f_f) \\
\parallel_M \\
lift(s) ; R1 \circ R3(\neg (Q f_f) \vdash Q f_f)
\end{align*}
\[
\begin{align*}
= \{ \text{A.11.13 (law::reactive-design)} \} \\
(\text{lift}(s) ; P) \|_{M} (\text{lift}(s) ; Q) \\
= \{ \text{3.1.3 (definition::reactive-design::parallel::alphabetised-partitioned)} \} \\
(\text{lift}(s) ; P) \| [x_1 | cs | x_2] (\text{lift}(s) ; Q)
\end{align*}
\]

\[\square\]

3.2.4.11 \textit{law::lift::semi-idempotence}

\textbf{Law 3.2.14 (law::lift::semi-idempotence)} Suppose that \( (\alpha(s) = \{x_1, x'_1, x_2, x'_2\}) \), and that \( x_1 \) and \( x_2 \) are not free in \( s \).

\[s ; (s | x_1)_{+x_2} = s\]

\textbf{Proof} Without loss of generality, assume that \( (s = (x_1, x_2 := c_1, c_2)) \), for constant expressions \( c_1 \) and \( c_2 \).

\[s ; (s | x_1)_{+x_2}\]
\[= \{ \text{assumption: (s = (x_1, x_2 := c_1, c_2))} \} \]
\[x_1, x_2 := c_1, c_2 ; ((x_1, x_2 := c_1, c_2) | x_1)_{+x_2}\]
\[= \{ \text{A.13.17 (law::CML::assignment::restriction)} \} \]
\[x_1, x_2 := c_1, c_2 ; (x_1 := c_1)_{+x_2}\]
\[= \{ \text{A.4.1 (definition::alphabet::lifting)} \} \]
\[x_1, x_2 := c_1, c_2 ; x_1, x_2 := c_1, x_2\]
\[= \{ \text{A.13.16 (law::CML::assignment::composition)} \} \]
\[x_1, x_2 := c_1, c_2\]
\[= \{ \text{assumption: (s = (x_1, x_2 := c_1, c_2))} \} \]
\[s\]

\[\square\]

3.2.4.12 \textit{law::lift::var}

\textbf{Law 3.2.15 (law::lift::var)}

\[\text{lift}(\text{var} x := w_1) = \text{var} x ; \text{lift}(x := w_1)\]
3.3 Soundness

3.3.1 theorem::-CML:-assignment-rule-sound

Theorem 3.3.1 (theorem::-CML:-assignment-rule-sound)

Proof  W.T.P.

\[ (c \mid s \models v := e) \rightarrow (c \text{ and } (s ; (w_1 = e)) \mid s ; v := w_1 \models \text{Skip}) \]

S.T.P. (tau-transition soundness)

\[ \forall w \bullet c \land (s ; (w_1 = e)) \Rightarrow \text{lift}(s) ; v := RD e \sqsubseteq \text{lift}(s ; v := w_1) ; \text{Skip} \]

assume c and (s ; (w_1 = e))

\[ R.H.S. = \]

\[ \text{lift}(s ; v := w_1) ; \text{Skip} \]

\[ = \{ 3.2.6 \text{ (law::-lift:-CSP4)} \} \]

\[ \text{lift}(s ; v := w_1) \]

\[ = \{ 3.2.5 \text{ (law::-lift:-composition)} \} \]

\[ \text{lift}(s) ; \text{lift}(v := w_1) \]

\[ = \{ 3.2.10 \text{ (law::-lift:-leading:-substitution) (assumption: } (s ; (w_1 = e))\} \} \]

\[ \text{lift}(s) ; \text{lift}(v := w_1)[e/w_1] \]

\[ = \{ A.7.6 \text{ (law::-substitution)} \} \]

\[ \text{lift}(s) ; \text{lift}(v := e) \]

\[ = \{ 3.1.1 \text{ (definition::-reactive-design:-assignment)} \} \]

\[ \text{lift}(s) ; v := RD e \]

\[ = L.H.S. \]

\[ \square \]

3.3.2 theorem::-CML:-input:-sound

Theorem 3.3.2 (theorem::-CML:-input:-sound)

Proof  S.T.P.

\[ \forall w \bullet c \land w_1 : T \Rightarrow \]

\[ \text{lift}(s) ; d?x : T \rightarrow A \]

\[ \sqsubseteq \]

\[ (\text{lift}(s) ; \text{var} x := w_1) ; d. w_1 \rightarrow \textbf{let} x \bullet A) \square (\text{lift}(s) ; d?x : T \rightarrow A) \]

\[ R.H.S. = \]
Theorem 3.3.3 (\textit{theorem::CML::nondeterministic-choice-rule-sound})

\textit{Proof} \quad \textit{S.T.P. (tau-transition soundness)}

\[ \forall w \cdot \exists c \Rightarrow \text{lift}(s) ; A_1 \sqcap A_2 \sqsubseteq \text{lift}(s) ; A_1 \]

\textit{assume} \, c\\
\hspace{1cm} L.H.S.

\[ = \]
\[ \text{lift}(s) ; (A_1 \sqcap A_2) \]
\[ \sqsubseteq \{ \ A.5.12 \ (\text{law::relational-calculus::refinement::nondeterministic-choice}) \ \} \]
\[ \text{lift}(s) ; A_1 \]
\[ = R.H.S \]
3.3.4 \textit{theorem::CML::output-rule-sound}

**Theorem 3.3.4** (\textit{theorem::CML::output-rule-sound})

**Proof**  assume \( w_1 \) does not occur in \( c, s, e, \) or \( A \) S.T.P.

\[ \forall w \cdot c \wedge (s ; (w_1 = e)) \Rightarrow \text{lift}(s) ; d.e \rightarrow A \sqsubseteq (\text{lift}(s) ; d.w_1 \rightarrow A) \triangleq (\text{lift}(s) ; d.e \rightarrow A) \]

assume \( c \wedge (s ; (w_1 = e)) \)

\begin{align*}
R.H.S. = \\
(\text{lift}(s) ; d.w_1 \rightarrow A) \triangleq (\text{lift}(s) ; d.e \rightarrow A) \\
= \{ \text{ 3.2.10 (law::lift::leading::substitution)} (\text{assumption:} (s ; (w_1 = e))) \} \\
(\text{lift}(s) ; (d.w_1 \rightarrow A)[e/w_1]) \triangleq (\text{lift}(s) ; d.e \rightarrow A) \\
= \{ \text{ 3.2.10 (law::lift::leading::substitution)} \} \\
(\text{lift}(s) ; d.e \rightarrow A)[e/w_1] \triangleq (\text{lift}(s) ; d.e \rightarrow A) \\
= \{ \text{ A.7.6 (law::substitution)} \} \\
(\text{lift}(s) ; d.e \rightarrow A) \triangleq (\text{lift}(s) ; d.e \rightarrow A) \\
= \{ \text{ A.13.26 (law::CML::external-choice::idempotence)} \} \\
\text{lift}(s) ; d.e \rightarrow A \\
= L.H.S.
\end{align*}

\[ \square \]

3.3.5 \textit{theorem::CML::external-choice::begin-sound}

**Theorem 3.3.5** (\textit{theorem::CML::external-choice::begin-sound})

**Proof**  S.T.P.

\[ \forall w \cdot cimplies\text{lift}(s) ; A_1 \triangleq A_2 \sqsubseteq \text{lift}(s) ; ((\text{loc}s \bullet A_1) \{+\} (\text{loc}s \bullet A_2)) \]

\begin{align*}
R.H.S. = \\
\text{lift}(s) ; ((\text{loc}s \bullet A_1) \{+\} (\text{loc}s \bullet A_2)) \\
= \{ \text{ A.13.10 (definition::CML::extrachoice)} \} \\
\text{lift}(s) ; ((\text{loc}s \bullet A_1) \triangleq (\text{loc}s \bullet A_2)) \\
= \{ \text{ A.13.2 (definition::CML::loc)} \} \\
\text{lift}(s) ; ((\text{lift}(s) ; A_1) \triangleq (\text{lift}(s) ; A_2)) \\
= \{ \text{ 3.2.7 (law::lift::external-choice)} \} \\
\text{lift}(s) ; \text{lift}(s) ; (A_1 \triangleq A_2) \\
= \{ \text{ 3.2.8 (law::lift::left-unit)} \} \\
\text{lift}(s) ; (A_1 \triangleq A_2)
\end{align*}
\[ L.H.S. \]

\[ \square \]

### 3.3.6 \textit{theorem}:::-CML:::-external-choice-Skip-rule-sound

**Theorem 3.3.6** (\textit{theorem}:::-CML:::-external-choice-Skip-rule-sound)

**Proof**  
S.T.P.

\[ \forall w \cdot c_1 \land c_2 \Rightarrow lift(s_1) ; ((\text{loc } s_1 \cdot \text{Skip}) \ [+] (\text{loc } s_2 \cdot A)) \sqsubseteq lift(s_1) ; \text{Skip} \]

\[ lift(s_1) ; ((\text{loc } s_1 \cdot \text{Skip}) \ [+] (\text{loc } s_2 \cdot A)) \]

\[ = \{ \ A.13.2 \ \text{(definition}:::-CML:::-loc) \ \} \]

\[ lift(s_1) ; (((lift(s_1) ; \text{Skip}) \ [+] (lift(s_2) \cdot A)) \]

\[ = \{ \ A.13.10 \ \text{(definition}:::-CML:::-extrachoice) \ \} \]

\[ lift(s_1) ; ((lift(s_1) ; \text{Skip}) \sqsubseteq (lift(s_2) \cdot A)) \]

\[ = \{ \ 3.2.6 \ \text{(law}:::-lift:-CSP4) \ \} \]

\[ lift(s_1) ; (lift(s_1) \land (lift(s_2) \cdot A)) \]

\[ \sqsubseteq \{ \ A.13.21 \ \text{(law}:::-CML:::-external-choice:-assignment) \ \} \]

\[ lift(s_1) ; lift(s_1) \]

\[ = \{ \ 3.2.8 \ \text{(law}:::-lift:-left-unit) \ \} \]

\[ lift(s_1) \]

\[ \square \]

### 3.3.7 \textit{theorem}:::-CML:::-external-choice:-end-sound

**Theorem 3.3.7** (\textit{theorem}:::-CML:::-external-choice:-end-sound)

**Proof**  
Assume \((c_1 \mid s_1 \models A_1) \Rightarrow (c_3 \mid s_3 \models A_3)\) which is equivalent to

\[ \forall w \cdot c_1 \land c_3 \Rightarrow lift(s_1) ; A_1 \sqsubseteq (lift(s_3) ; l \rightarrow A_3) \sqsubseteq (lift(s_1) ; A_1) \]

S.T.P.

\[ \forall w \cdot c \land c_3 \Rightarrow \]

\[ lift(s) ; ((\text{loc } s_1 \cdot A_1) \ [+] (\text{loc } s_2 \cdot A_2)) \]

\[ \sqsubseteq \]

\[ (lift(s_3) ; l \rightarrow A_3) \sqsubseteq (lift(s) ; ((\text{loc } s_1 \cdot A_1) \ [+] (\text{loc } s_2 \cdot A_2))) \]

\[ R.H.S. \]

\[ = \]

\[ (lift(s_3) ; l \rightarrow A_3) \sqsubseteq (lift(s) ; ((\text{loc } s_1 \cdot A_1) \ [+] (\text{loc } s_2 \cdot A_2))) \]

43
\[ = \{ \text{A.13.10 (definition::CML:-extrachoice)} \} \]
\[ (\text{lift}(s_1) \cdot l \rightarrow A_3) \Box (\text{lift}(s) ; ((\text{loc } s_1 \cdot A_1) \Box (\text{loc } s_2 \cdot A_2))) \]
\[ = \{ \text{A.13.2 (definition::CML:-loc)} \} \]
\[ (\text{lift}(s_1) \cdot l \rightarrow A_3) \Box (\text{lift}(s) ; ((\text{lift}(s_1) ; A_1) \Box (\text{lift}(s_2) ; A_2))) \]
\[ = \{ \text{A.13.27 (law::CML:-external-choice:-lift-distributive)} \} \]
\[ (\text{lift}(s_1) \cdot l \rightarrow A_3) \Box (\text{lift}(s) ; (\text{lift}(s_1) ; A_1) \Box (\text{lift}(s) ; \text{lift}(s_2) ; A_2)) \]
\[ = \{ \text{3.2.8 (law::lift:-left-unit)} \} \]
\[ (\text{lift}(s_1) \cdot l \rightarrow A_3) \Box (\text{lift}(s_1) ; A_1) \Box (\text{lift}(s_2) ; A_2) \]
\[ = \{ \text{assumption} \} \]
\[ (\text{lift}(s_1) ; A_1) \Box (\text{lift}(s_2) ; A_2) \]
\[ = \{ \text{3.2.8 (law::lift:-left-unit)} \} \]
\[ (\text{lift}(s) ; \text{lift}(s_1) ; A_1) \Box (\text{lift}(s) ; \text{lift}(s_2) ; A_2) \]
\[ = \{ \text{A.13.27 (law::CML:-external-choice:-lift-distributive)} \} \]
\[ \text{lift}(s) ; (((\text{lift}(s_1) ; A_1) \Box (\text{lift}(s_2) ; A_2))) \]
\[ = \{ \text{A.13.2 (definition::CML:-loc)} \} \]
\[ \text{lift}(s) ; (((\text{loc } s_1 \cdot A_1) \Box (\text{loc } s_2 \cdot A_2))) \]
\[ = \{ \text{A.13.10 (definition::CML:-extrachoice)} \} \]
\[ \text{lift}(s) ; (((\text{loc } s_1 \cdot A_1) \cdot (\text{loc } s_2 \cdot A_2))) \]
\[ = \text{L.H.S.} \]

\[ \blacksquare \]

3.3.8 \textit{theorem::CML:-parallel:-begin-sound}

Theorem 3.3.8 (\textit{theorem::CML:-parallel:-begin-sound})

\textbf{Proof} \quad S.T.P.

\[ \forall w \cdot c \Rightarrow \]
\[ \text{lift}(s) ; A_1 \parallel x_1 \parallel c \parallel x_2 \parallel A_2 \]
\[ \subseteq \]
\[ \text{lift}(s) ; ((\text{loc } s_{x_1+x_2} \cdot A_1) \parallel x_1 \parallel c \parallel x_2 \parallel (\text{loc } s_{x_2+x_1} \cdot A_2)) \]

\text{R.H.S.}
\[ \]
\[ = \]
\[ \text{lift}(s) ; ((\text{loc } s_{x_1+x_2} \cdot A_1) \parallel x_1 \parallel c \parallel x_2 \parallel (\text{loc } s_{x_2+x_1} \cdot A_2)) \]
\[ = \{ \text{A.13.2 (definition::CML:-loc)} \} \]
\[ \text{lift}(s) ; (\text{lift}((s \parallel x_{1+x_2}) ; A_1) \parallel x_1 \parallel c \parallel x_2 \parallel (\text{lift}((s \parallel x_{2+x_1}) ; A_2)) \]
\[ = \{ \text{3.2.13 (law::lift:-parallel:-distributivity)} \} \]
\[ (\text{lift}(s) ; (\text{lift}((s \parallel x_{1+x_2}) ; A_1) \parallel x_1 \parallel c \parallel x_2 \parallel (\text{lift}(S) ; \text{lift}((s \parallel x_{2+x_1}) ; A_2)) \]
\[ = \{ \text{3.2.5 (law::lift:-composition), twice} \} \]

\[44\]
3.3.10 \(\langle \text{lift}(s ; (s \mid x_1 ; x_2) \cdot A_1) \parallel x_1 \parallel x_2 \rangle \langle \text{lift}(s ; (s \mid x_2) \cdot x_1 + x_2) \cdot A_2 \rangle = \{ \text{3.2.14 (law::lift::semi-idempotence)} \} \)

\(\langle \text{lift}(s) \cdot A_1 \parallel x_1 \parallel x_2 \rangle \langle \text{lift}(s_2) \cdot A_2 \rangle = \{ \text{3.2.13 (law::lift::parallel::distributivity)} \} \)

\(\text{lift}(s) ; (A_1 \parallel x_1 \parallel x_2) A_2) = L.H.S.\)

\[\Box\]

3.3.9 \(\text{theorem::CML::parallel::independent-sound}\)

Theorem 3.3.9 (\(\text{theorem::CML::parallel::independent-sound}\))

Proof Assume

\(\forall w \cdot c \land c_3 \Rightarrow \text{lift}(s_1) ; A_1 \sqsubseteq (\text{lift}(s_3) ; l \rightarrow A_3) \sqcap (\text{lift}(s_1) ; A_1)\)

S.T.P.

\(\forall w \cdot c \land c_3 \Rightarrow\)

\(\text{lift}(s) ; ((\text{loc} s_1 \cdot A_1) \parallel x_1 \parallel x_2) (\text{loc} s_2 \cdot A_2)\)

\(\sqsubseteq\)

\( (\text{lift}(s) ; l \rightarrow ((\text{loc} s_3 \cdot A_3) \parallel x_1 \parallel x_2) (\text{loc} s_2 \cdot A_2))\)

\(\sqcap\)

\( (\text{lift}(s) ; ((\text{loc} s_1 \cdot A_1) \parallel x_1 \parallel x_2) (\text{loc} s_2 \cdot A_2))\)

\[\Box\]

3.3.10 \(\text{theorem::CML::parallel::end-sound}\)

Theorem 3.3.10 (\(\text{theorem::CML::parallel::end-sound}\))

Proof S.T.P.

\(\forall w \cdot c \Rightarrow\)

\(\text{lift}(s) ; (\text{loc} s_1 \cdot \text{Skip}) \parallel x_1 \parallel x_2) (\text{loc} s_2 \cdot \text{Skip})\)

\(\sqsubseteq\)

\(\text{lift}(s_1 \parallel x_1 \land s_2 \parallel x_2) \cdot \text{Skip}\)

L.H.S.

\(=\)

\(\text{lift}(s) ; (\text{loc} s_1 \cdot \text{Skip}) \parallel x_1 \parallel x_2) (\text{loc} s_2 \cdot \text{Skip})\)

\(=\{ \text{A.13.2 (definition::CML::loc)} \}\)

\(\text{lift}(s) ; (\text{lift}(s_1) \cdot \text{Skip}) \parallel x_1 \parallel x_2) (\text{lift}(s_2) \cdot \text{Skip})\)

\(=\{ \text{3.2.13 (law::lift::parallel::distributivity)} \}\)

\( (\text{lift}(s) ; (\text{lift}(s_1) \cdot \text{Skip}) \parallel x_1 \parallel x_2) (\text{lift}(s) ; \text{lift}(s_2) \cdot \text{Skip})\)
\[
\begin{align*}
= \{ \text{3.2.8 (law::lift:-left-unit), twice } \} \\
(lift(s_1) ; \text{Skip}) [ x_1 | cs | x_2 ] (lift(s_2) ; \text{Skip}) \\
= \{ \text{3.2.6 (law::lift:-CSP4), twice } \} \\
lift(s_1) [ x_1 | cs | x_2 ] lift(s_2) \\
= \{ \text{3.2.12 (law::lift:-merge) } \} \\
lift(s_1 | x_1 \land s_2 | x_2) \\
= \{ \text{3.2.6 (law::lift:-CSP4) } \} \\
lift(s_1 | x_1 \land s_2 | x_2) ; \text{Skip} \\
= R.H.S.
\end{align*}
\]
Appendix A

Algebraic Laws for CML

A.1  *propositional-calculus*

A.1.1  *propositional-calculus:-and*

A.1.1.1  *law::-propositional-calculus:-and:-associativity*

Law A.1.1  (*law::-propositional-calculus:-and:-associativity*)

\[(P \land Q) \land R = P \land (Q \land R)\]

A.1.1.2  *law::-propositional-calculus:-and:-commutativity*

Law A.1.2  (*law::-propositional-calculus:-and:-commutativity*)

\[P \land Q = Q \land P\]

A.1.1.3  *law::-propositional-calculus:-and:-elimination*

Law A.1.3  (*law::-propositional-calculus:-and:-elimination*)

\[P \land Q \Rightarrow P\]

A.1.1.4  *law::-propositional-calculus:-and:-idempotence*

Law A.1.4  (*law::-propositional-calculus:-and:-idempotence*)

\[P \land P = P\]

A.1.1.5  *law::-propositional-calculus:-and:-or-distributivity*

Law A.1.5  (*law::-propositional-calculus:-and:-or-distributivity*)

\[P \land (Q \lor R) = (P \land Q) \lor (P \land R)\]
A.1.1.6  \textit{law::propositional-calculus::and::unit}

Law A.1.6 (\textit{law::propositional-calculus::and::unit})

\[ P \land \text{true} = P \]

A.1.1.7  \textit{law::propositional-calculus::and::zero}

Law A.1.7 (\textit{law::propositional-calculus::and::zero})

\[ P \land \text{false} = \text{false} \]

A.1.2  \textit{propositional-calculus::or}

A.1.2.1  \textit{law::propositional-calculus::or::absorption}

Law A.1.8 (\textit{law::propositional-calculus::or::absorption})

\[ P \lor (P \land Q) = P \]

A.1.2.2  \textit{law::propositional-calculus::or::elimination}

Law A.1.9 (\textit{law::propositional-calculus::or::elimination})

\[ P \lor Q \Rightarrow R = (P \Rightarrow R) \land (Q \Rightarrow R) \]

A.1.2.3  \textit{law::propositional-calculus::or::idempotence}

Law A.1.10 (\textit{law::propositional-calculus::or::idempotence})

\[ P \lor P = P \]

A.1.2.4  \textit{law::propositional-calculus::or::introduction}

Law A.1.11 (\textit{law::propositional-calculus::or::introduction})

\[ P \Rightarrow P \lor Q \]

A.1.2.5  \textit{law::propositional-calculus::or::subsumption}

Law A.1.12 (\textit{law::propositional-calculus::or::subsumption})

\[ (P \Rightarrow Q) \Rightarrow (P \lor Q = Q) \]
A.1.2.6 \( \text{law::-propositional-calculus:-or:-unit} \)

Law A.1.13 (\( \text{law::-propositional-calculus:-or:-unit} \))

\[
P \lor \text{false} = P
\]

A.1.2.7 \( \text{law::-propositional-calculus:-or:-zero} \)

Law A.1.14 (\( \text{law::-propositional-calculus:-or:-zero} \))

\[
P \lor \text{true} = \text{true}
\]

A.1.3 \( \text{propositional-calculus:-negation} \)

A.1.3.1 \( \text{law::-propositional-calculus:-negation:-false} \)

Law A.1.15 (\( \text{law::-propositional-calculus:-negation:-false} \))

\[\neg \text{false} = \text{true}\]

A.1.3.2 \( \text{law::-propositional-calculus:-negation:-true} \)

Law A.1.16 (\( \text{law::-propositional-calculus:-negation:-true} \))

\[\neg \text{true} = \text{false}\]

A.1.3.3 \( \text{law::-propositional-calculus:-negation:-contradiction} \)

Law A.1.17 (\( \text{law::-propositional-calculus:-negation:-contradiction} \))

\[P \land \neg P = \text{false}\]

A.1.3.4 \( \text{law::-propositional-calculus:-negation:-contradiction:-rewrite} \)

Law A.1.18 (\( \text{law::-propositional-calculus:-negation:-contradiction:-rewrite} \))

\[(P = \text{false}) = \neg P\]

A.1.3.5 \( \text{law::-propositional-calculus:-negation:-De-Morgan} \)

Law A.1.19 (\( \text{law::-propositional-calculus:-negation:-De-Morgan} \))

\[\neg (P \land Q) = \neg P \lor \neg Q\]
A.1.3.6  \textit{law::-propositional-calculus:-negation:-double-negation}

Law A.1.20 (\textit{law::-propositional-calculus:-negation:-double-negation})

\[ \neg \neg P = P \]

A.1.3.7  \textit{law::-propositional-calculus:-negation:-excluded-middle}

Law A.1.21 (\textit{law::-propositional-calculus:-negation:-excluded-middle})

\[ P \lor \neg P \]

A.1.3.8  \textit{law::-propositional-calculus:-negation:-absorption}

Law A.1.22 (\textit{law::-propositional-calculus:-negation:-absorption})

\[ P \land \neg (\neg P \land Q) = P \]

A.1.4  \textit{propositional-calculus:-implies}

A.1.4.1  \textit{law::-propositional-calculus:-implies}

Law A.1.23 (\textit{law::-propositional-calculus:-implies})

\[ P \Rightarrow Q = \neg P \lor Q \]

A.1.4.2  \textit{law::-propositional-calculus:-implies:-absorption}

Law A.1.24 (\textit{law::-propositional-calculus:-implies:-absorption})

\[ (P \Rightarrow Q) \Rightarrow Q = \neg P \Rightarrow Q \]

A.1.4.3  \textit{law::-propositional-calculus:-implies:-accumulation}

Law A.1.25 (\textit{law::-propositional-calculus:-implies:-accumulation})

\[ P \Rightarrow (Q \Rightarrow R) = P \land Q \Rightarrow R \]

A.1.4.4  \textit{law::-propositional-calculus:-implies:-and-antecedent}

Law A.1.26 (\textit{law::-propositional-calculus:-implies:-and-antecedent})

\[ P \land Q \Rightarrow R = (P \Rightarrow R) \lor (Q \Rightarrow R) \]
A.1.4.5 law::propositional-calculus::implies::and-consequent

Law A.1.27 (law::propositional-calculus::implies::and-consequent)

\[ P \Rightarrow Q \land R = (P \Rightarrow Q) \land (P \Rightarrow R) \]

A.1.4.6 law::propositional-calculus::implies::contradiction

Law A.1.28 (law::propositional-calculus::implies::contradiction)

\( (\text{false} \Rightarrow P) = \text{true} \)

A.1.4.7 law::propositional-calculus::implies::export

Law A.1.29 (law::propositional-calculus::implies::export)

\[ P \land Q \Rightarrow R = P \land Q \Rightarrow P \land R \]

A.1.4.8 law::propositional-calculus::implies::identity

Law A.1.30 (law::propositional-calculus::implies::identity)

\[ R \Rightarrow R = \text{true} \]

A.1.4.9 law::propositional-calculus::implies::negation

Law A.1.31 (law::propositional-calculus::implies::negation)

\[ \neg (P \Rightarrow Q) = P \land \neg Q \]

A.1.5 propositional-calculus::equivalence

A.1.5.1 law::propositional-calculus::equivalence::boolean

Law A.1.32 (law::propositional-calculus::equivalence::boolean)

\( (x = \text{false}) = \neg x \)

A.1.6 propositional-calculus::conditional

A.1.6.1 definition::propositional-calculus::conditional

Definition A.1.1 (definition::propositional-calculus::conditional)

\[ P \Leftrightarrow b \Leftrightarrow Q \equiv (b \land P) \lor (\neg b \land Q) \]
A.1.6.2  \textit{law::-propositional-calculus:-conditional:-assumption-else}

Law A.1.33 (\textit{law::-propositional-calculus:-conditional:-assumption-else})

\[ P \triangleleft b \triangleright Q = P \triangleleft b \triangleright (\neg b \land Q) \]

\textbf{Proof}

\[ P \triangleleft b \triangleright Q \]
\[ = \{ \text{A.1.1 (definition::-propositional-calculus:-conditional) } \} \]
\[ (b \land P) \lor (\neg b \land Q) \]
\[ = \{ \text{A.1.4 (law::-propositional-calculus:-and:-idempotence) } \} \]
\[ (b \land P) \lor (\neg b \land \neg b \land Q) \]
\[ = \{ \text{A.1.1 (definition::-propositional-calculus:-conditional) } \} \]
\[ P \triangleleft b \triangleright (\neg b \land Q) \]

\[ \Box \]

A.1.6.3  \textit{law::-propositional-calculus:-conditional:-assumption-then}

Law A.1.34 (\textit{law::-propositional-calculus:-conditional:-assumption-then})

\[ P \triangleleft b \triangleright Q = (b \land P) \triangleleft b \triangleright Q \]

\textbf{Proof}

\[ P \triangleleft b \triangleright Q \]
\[ = \{ \text{A.1.1 (definition::-propositional-calculus:-conditional) } \} \]
\[ (b \land P) \lor (\neg b \land Q) \]
\[ = \{ \text{A.1.4 (law::-propositional-calculus:-and:-idempotence) } \} \]
\[ (b \land P) \lor (\neg b \land P) \lor (\neg b \land Q) \]
\[ = \{ \text{A.1.1 (definition::-propositional-calculus:-conditional) } \} \]
\[ (b \land P) \triangleleft b \triangleright Q \]

\[ \Box \]

A.1.6.4  \textit{law::-propositional-calculus:-conditional:-constant:-false}

Law A.1.35 (\textit{law::-propositional-calculus:-conditional:-constant:-false})

\[ (P \triangleleft \textit{false} \triangleright Q) = Q \]

\textbf{Proof}
\[(P \triangleleft false \triangleright Q)\]
\[= \{ \text{A.1.1 (definition::propositional-calculus::conditional)} \}\]
\[(false \land P) \lor (\neg false \land Q)\]
\[= \{ \text{A.1.7 (law::propositional-calculus::and::zero)} \}\]
\[false \lor (\neg false \land Q)\]
\[= \{ \text{A.1.13 (law::propositional-calculus::or::unit)} \}\]
\[\neg false \land Q\]
\[= \{ \text{A.1.15 (law::propositional-calculus::negation::false)} \}\]
\[true \land Q\]
\[= \{ \text{A.1.6 (law::propositional-calculus::and::unit)} \}\]
\[Q\]

\[\square\]

A.1.6.5 \textit{law::propositional-calculus::conditional::constant::then}

Law A.1.36 \textit{(law::propositional-calculus::conditional::constant::then)}
\[(P \triangleleft b \triangleright false) = b \land P\]

\textit{Proof}
\[(P \triangleleft b \triangleright false)\]
\[= \{ \text{A.1.1 (definition::propositional-calculus::conditional)} \}\]
\[(b \land P) \lor (\neg b \land false)\]
\[= \{ \text{A.1.7 (law::propositional-calculus::and::zero)} \}\]
\[(b \land P) \lor false\]
\[= \{ \text{A.1.13 (law::propositional-calculus::or::unit)} \}\]
\[b \land P\]

\[\square\]

A.1.6.6 \textit{law::propositional-calculus::conditional::and-distributivity}

Law A.1.37 \textit{(law::propositional-calculus::conditional::and-distributivity)}
\[(P \triangleleft b \triangleright Q) \land R = (P \land R) \triangleleft b \triangleright (Q \land R)\]

A.1.6.7 \textit{law::propositional-calculus::conditional::exchange::or}

Law A.1.38 \textit{(law::propositional-calculus::conditional::exchange::or)}
\[(P \triangleleft b \triangleright Q) \lor (R \triangleleft b \triangleright S) = (P \lor R) \triangleleft b \triangleright (Q \lor S)\]
A.1.6.8  \textit{law::propositional-calculus::conditional::export::then}

Law A.1.39 (\textit{law::propositional-calculus::conditional::export::then})
\[ P \land b \implies Q = P[true/b] \land b \implies Q \]

A.1.6.9  \textit{law::propositional-calculus::conditional::export::else}

Law A.1.40 (\textit{law::propositional-calculus::conditional::export::else})
\[ P \land b \implies Q = P \land b \implies Q[false/b] \]

A.1.6.10  \textit{law::propositional-calculus::conditional::idempotence}

Law A.1.41 (\textit{law::propositional-calculus::conditional::idempotence})
\[ P \land b \implies P = P \]

Proof

\[ P \land b \implies P \]
\[ = \{ \text{A.1.1 (definition::propositional-calculus::conditional)} \} \]
\[ (b \land P) \lor (\neg b \land P) \]
\[ = \{ \text{A.1.5 (law::propositional-calculus::and::or-distributivity)} \} \]
\[ (b \lor \neg b) \land P \]
\[ = \{ \text{A.1.21 (law::propositional-calculus::negation::excluded-middle)} \} \]
\[ \text{true} \land P \]
\[ = \{ \text{A.1.6 (law::propositional-calculus::and::unit)} \} \]
\[ P \]
\[ \square \]

A.1.6.11  \textit{law::propositional-calculus::conditional::negation}

Law A.1.42 (\textit{law::propositional-calculus::conditional::negation})
\[ \neg (P \land b \implies Q) = \neg P \land b \implies \neg Q \]

A.1.6.12  \textit{law::propositional-calculus::conditional::simplification-1}

Law A.1.43 (\textit{law::propositional-calculus::conditional::simplification-1})
\[ ((P \land b \implies Q) \land b \implies (R \land b \implies S)) = (P \land b \implies S) \]
A.2 **predicate-calculus**

A.2.1 **predicate-calculus:-exists**

A.2.1.1 **law::predicate-calculus:-exists:-and:-non-free**

Law A.2.1 (**law::predicate-calculus:-exists:-and:-non-free**)

\[ \exists x \bullet P \land N = (\exists x \bullet P) \land N \quad \text{for } x \text{ not free in } N \]

A.2.1.2 **law::predicate-calculus:-exists:-De-Morgan**

Law A.2.2 (**law::predicate-calculus:-exists:-De-Morgan**)

\[ \neg \forall x \bullet P = \exists x \bullet \neg P \]

A.2.1.3 **law::predicate-calculus:-exists:-detach**

Law A.2.3 (**law::predicate-calculus:-exists:-detach**)

\[ \exists X : S \cup \{P\} \bullet X = P \lor \exists X : S \bullet X \]

A.2.1.4 **law::predicate-calculus:-exists:-introduction**

Law A.2.4 (**law::predicate-calculus:-exists:-introduction**)

\[ P \Rightarrow \exists x \bullet P \]

A.2.1.5 **law::predicate-calculus:-exists:-nested**

Law A.2.5 (**law::predicate-calculus:-exists:-nested**)

\[ \exists x \bullet (\exists y \bullet P) = \exists y \bullet (\exists x \bullet P) \]

A.2.1.6 **law::predicate-calculus:-exists:-one-point**

Law A.2.6 (**law::predicate-calculus:-exists:-one-point**)

\[ \exists x \bullet (x = e) \land P = P[e/x] \]

\[ \text{for } x \text{ not free in } P \]

A.2.1.7 **law::predicate-calculus:-exists:-or-distributivity**

Law A.2.7 (**law::predicate-calculus:-exists:-or-distributivity**)

\[ \exists x \bullet (P \land Q) \lor (P \land R) = (\exists x \bullet P \land Q) \lor (\exists x \bullet P \land R) \]
A.2.1.8 \textit{law::-predicate-calculus:-exists:-rename-bound-var}

Law A.2.8 (\textit{law::-predicate-calculus:-exists:-rename-bound-var})

\[\exists x \cdot P = \exists y \cdot P[y/x] \quad \text{for fresh } y\]

A.2.2 \textit{predicate-calculus:-forall}

A.2.2.1 \textit{law::-predicate-calculus:-forall:-detach}

Law A.2.9 (\textit{law::-predicate-calculus:-forall:-detach})

\[\forall X: S \cup \{P\} \cdot X = P \wedge \forall X: S \cdot X\]

A.2.2.2 \textit{law::-predicate-calculus:-forall:-tautology}

Law A.2.10 (\textit{law::-predicate-calculus:-forall:-tautology})

\[[P] = (P = \text{true})\]

A.2.2.3 \textit{law::-predicate-calculus:-forall:-boolean}

Law A.2.11 (\textit{law::-predicate-calculus:-forall:-boolean})

\[[P] = [P[\text{true}/x] \wedge P[\text{false}/x]]_{\forall x: \alpha(P)}\]

A.2.2.4 \textit{law::-predicate-calculus:-forall:-universal}

Law A.2.12 (\textit{law::-predicate-calculus:-forall:-universal})

\[(P = Q) = (P \subseteq Q) \wedge (Q \subseteq P)\]
A.3  \textit{equals}

A.3.1  \textit{law::-equals:-cartesian-pair}

Law A.3.1 (\textit{law::-equals:-cartesian-pair})

\[(x_1, y_1) = (x_2, y_2) \Rightarrow (x_1 = x_2) \land (y_1 = y_2)\]

A.3.2  \textit{law::-equals:-Leibniz}

Law A.3.2 (\textit{law::-equals:-Leibniz})

\[(x = e) \land P = (x = e) \land P[e/x]\]

A.3.3  \textit{law::-equals:-reflection}

Law A.3.3 (\textit{law::-equals:-reflection})

\[(e = e)\]
A.4  alphabet

A.4.1  definition:::-alphabet:-lifting

Definition A.4.1  (definition:::-alphabet:-lifting)

\[ P_{+x} \triangleq P \land (x' = x) \]

A.4.2  law:::-alphabet:-lifting:-disjunctivity

Law A.4.1  (law:::-alphabet:-lifting:-disjunctivity)

\[ (P \lor Q)_{+x} = P_{+x} \lor Q_{+x} \]

A.4.3  law:::-alphabet:-lifting:-assignment

Law A.4.2  (law:::-alphabet:-lifting:-assignment)

\[ (x := e)_{+y} = x, y := e, y \]

A.4.4  law:::-alphabet:-and

Definition A.4.2  (law:::-alphabet:-and)

\[ (P \land Q)_A = P_A \land Q_A \]

A.4.5  law:::-alphabet:-change-of-variable

Definition A.4.3  (law:::-alphabet:-change-of-variable)

\[ (P_A)[y/x] = (P[y/x])_A[y/x] \quad \text{for } x \text{ in } A, y \text{ not in } A \]

A.4.6  law:::-alphabet:-exists

Definition A.4.4  (law:::-alphabet:-exists)

\[ \exists x \bullet P_A = (\exists x \bullet P)_A\setminus\{x\} \]

A.4.7  law:::-alphabet:-lifting:-conjunctivity

Law A.4.3  (law:::-alphabet:-lifting:-conjunctivity)

\[ (P \land Q)_{+x} = P_{+x} \land Q \]
Proof

\[(P \land Q)_{+x}\]
\[= \{ A.4.1 \ (definition::alphabet:-lifting) \} \]
\[P \land Q \land (x' = x)\]
\[= \{ A.1.2 \ (law::-propositional-calculus:-and:-commutativity) \} \]
\[\{ A.1.1 \ (law::-propositional-calculus:-and:-associativity) \} \]
\[P \land (x' = x) \land Q\]
\[= \{ A.4.1 \ (definition::alphabet:-lifting) \} \]
\[P_{+x} \land Q\]

\[\Box\]

A.4.8  law::alphabet::lifting::sequence

Law A.4.4  (law::alphabet::lifting::sequence)

\[(P ; Q)_{+x} = P_{+x} ; Q_{+x}\]

Proof

\[(P ; Q)_{+x}\]
\[= \{ A.4.1 \ (definition::alphabet:-lifting) \} \]
\[(P ; Q) \land (x' = x)\]
\[= \{ A.5.9 \ (definition::relational-calculus:-sequence) \} \]
\[\exists v_0 \bullet P[v_0/v'] \land Q[v_0/v] \land (x' = x)\]
\[= \{ A.2.1 \ (law::-predicate-calculus:-exists:-and:-non-free) \} \]
\[\exists v_0 \bullet P[v_0/v'] \land Q[v_0/v] \land (x' = x)\]
\[= \{ A.1.4 \ (law::-propositional-calculus:-and:-idempotence) \} \]
\[\exists v_0 \bullet P[v_0/v'] \land (x' = x) \land Q[v_0/v] \land (x' = x)\]
\[= \{ A.7.6 \ (law::-substitution) \} \]
\[\exists v_0 \bullet (P \land (x' = x))[v_0/v'] \land (Q \land (x' = x))[v_0/v]\]
\[= \{ A.5.9 \ (definition::relational-calculus:-sequence) \} \]
\[(P \land (x' = x)) ; (Q \land (x' = x))\]
\[= \{ A.4.1 \ (definition::alphabet:-lifting), \ twice \} \]
\[P_{+x} ; Q_{+x}\]

\[\Box\]
A.5 relational-calculus

A.5.1 relational-calculus::assignment

A.5.1.1 definition::relational-calculus::assignment

Definition A.5.1 (definition::relational-calculus::assignment)

\[ x :=_v e \equiv (x' = e) \land (v'_1 = v_1) \land \ldots \land (v'_n = v_n) \text{ for } v = v_1, \ldots, v_n \]

A.5.1.2 definition::relational-calculus::assignment::declaration-init

Definition A.5.2 (definition::relational-calculus::assignment::declaration-init)

\( (\text{var} x :=_v e) \equiv (\text{var} x ; x :=_v e) \)

A.5.1.3 law::relational-calculus::assignment::alphabet-extension

Law A.5.1 (law::relational-calculus::assignment::alphabet-extension)

\[ (y :=\{x,x',y,y',z,z'\} f) = (x, y :=\{y,y',z,z'\} f, x) \]

Proof

\[ y :=\{x,x',y,y',z,z'\} f = \{ A.5.1 \} \]
\[ (x' = x) \land (y' = f) \land (z' = z) = \{ A.5.1 \} \]
\[ x, y :=\{y,y',z,z'\} f, x \]

A.5.1.4 law::relational-calculus::assignment::cancel-by-end-var

Law A.5.2 (law::relational-calculus::assignment::cancel-by-end-var)

\[ y :=\{x,x',y,y'\} f ; \text{end} x ; \text{var} x = (y' = f) \]

Proof

\[ y :=\{x,x',y,y'\} f ; \text{end} x ; \text{var} x = \{ A.5.5 \} \]
\[ (\text{end} x ; \text{var} x)[f / y] \]
= \{ \text{A.7.6 (law::substitution)} \} \\
(\text{end } x)[f/y] \ ; \ \text{var } x

= \{ \text{A.5.3 (definition::relational-calculus::end::alphabetised-equation)} \} \\
((y' = y)_{x,y,y'})(f/y) \ ; \ \text{var } x

= \{ \text{A.7.6 (law::substitution)} \} \\
(y' = f)_{x,y,y'} \ ; \ \text{var } x

= \{ \text{A.5.27 (law::relational-calculus::var::alphabetised-equation)} \} \\
(y' = f)_{x,y,y'} \ ; \ (y' = y)_{x',y,y'}

= \{ \text{A.5.9 (definition::relational-calculus::sequence)} \} \\
\exists y_0 \bullet ((y' = f)_{x,y,y'})(y_0/y) \land ((y' = y)_{x',y,y'})(y_0/y)

= \{ \text{A.4.3 (law::alphabet::change-of-variable)} \} \\
\exists y_0 \bullet (y_0 = f)_{x,y,y'} \land ((y' = y)_{x',y,y'})(y_0/y)

= \{ \text{A.4.3 (law::alphabet::change-of-variable)} \} \\
\exists y_0 \bullet (y_0 = f)_{x,y,y'} \land (y' = y_0)_{x',y,y'}

= \{ \text{A.4.2 (law::alphabet::and)} \} \\
\exists y_0 \bullet ((y_0 = f) \land (y' = y_0))_{x,x',y,y',y'}

= \{ \text{A.4.4 (law::alphabet::exists)} \} \\
(\exists y_0 \bullet (y_0 = f) \land (y' = y_0))_{x,x',y,y'}

= \{ \text{A.2.6 (law::predicate-calculus::exists::one-point)} \} \\
(y_0 = f)_{y'/y_0} \ ; \ x,x',y,y'

= \{ \text{A.4.4 (law::alphabet::exists)} \} \\
(y' = f)_{x,x',y,y'}

\square

\text{A.5.1.5 law::relational-calculus::assignment::lifted-identity}

\text{Law A.5.3 (law::relational-calculus::assignment::lifted-identity)}

(x :=_{x,y,x'} e) = \Pi_{x,x'} \circ (y) \ ; \ x :=_{x,x'} e

\text{A.5.1.6 law::relational-calculus::assignment::sequence::identity}

\text{Law A.5.4 (law::relational-calculus::assignment::sequence::identity)}

v := e \ ; \ v := v = v := e

\text{A.5.1.7 law::relational-calculus::assignment::sequence}

\text{Law A.5.5 (law::relational-calculus::assignment::sequence)}

\text{for } x \text{ not free in } e

x := e \ ; \ P = P[e/x]
Proof

\[ x := e ; P \]
\[ = \{ A.5.9 \ (\text{definition::relational-calculus::sequence}) \} \]
\[ \exists x_0, y_0 \cdot (x := e)[x_0, y_0/x', y'] \land P[x_0, y_0/x, y] \]
\[ = \{ A.5.1 \ (\text{definition::relational-calculus::assignment}) \} \]
\[ \exists x_0, y_0 \cdot ((x' = e) \land (y' = y))[x_0, y_0/x', y'] \land P[x_0, y_0/x, y] \]
\[ = \{ A.7.6 \ (\text{law::substitution}) \} \]
\[ \exists x_0, y_0 \cdot (x_0 = e) \land (y_0 = y) \land P[x_0, y_0/x, y] \]
\[ = \{ A.2.6 \ (\text{law::predicate-calculus::exists::one-point}) \} \]
\[ P[e, y/x, y] \]
\[ = \{ A.7.6 \ (\text{law::substitution}) \} \]
\[ P[e/x] \]

\(\Box\)

A.5.1.8 law::relational-calculus::assignment::unwinding

Law A.5.6 (law::relational-calculus::assignment::unwinding)

\[ x := \{x, y, x', y'\} \ e = (x := \{x, y, x'\} \ e) \land (y' = y) \]

A.5.2 relational-calculus::end

A.5.2.1 definition::relational-calculus::end::alphabetised-equation

Definition A.5.3 (definition::relational-calculus::end::alphabetised-equation)

\[ (\text{end} x)\{x, x', y, y'\} \cong (y' = y)\{x, y, y'\} \]

A.5.2.2 definition::relational-calculus::end::split

Definition A.5.4 (definition::relational-calculus::end::split)

\[ \text{end} x, y = \text{end} x ; \text{end} y \]

A.5.2.3 definition::relational-calculus::end

Definition A.5.5 (definition::relational-calculus::end)

\[ \text{end} x \cong \exists x' \cdot \Pi \]
A.5.2.4  **law::-relational-calculus:-end:-alphabetised**

Law A.5.7  (**law::-relational-calculus:-end:-alphabetised**)  
\[
\text{end } x = (y' = y) \land (z' = z) \{x, y, y', z, z'\}
\]

A.5.2.5  **law::-relational-calculus:-end:-assignment**

Law A.5.8  (**law::-relational-calculus:-end:-assignment**)  
\[
\text{fora}(e) = \{x\} x := \{x, y, y'\} e ; \text{end } y = \end y ; x := \{x, x'\} e
\]

**Proof**

\[
x := \{x, y, y', y'\} e ; \end y
= \{ \text{A.5.9 (law::-relational-calculus:-end:-sequence)} \}
\exists y' \bullet x := \{x, y, y', y'\} e
= \{ \text{A.5.6 (law::-relational-calculus:-assignment:-unwinding)} \}
\exists y' \bullet (x := \{x, y, y'\} e) \land (y' = y)
= \{ \text{A.2.6 (law::-predicate-calculus:-exists:-one-point)} \}
(x := \{x, y, y'\} e)[y/y']
= \{ \text{A.7.6 (law::-substitution)} \}
(x := \{x, y, y'\} e)
= \{ \text{A.5.3 (law::-relational-calculus:-assignment:-lifted-identity)} \}
\Pi_{x'}(y); x := \{x, x'\} e
= \{ \text{A.5.7 (definition::-relational-calculus:-II:-heterogeneous)} \}
(\exists y' \bullet \Pi_{x, y, x', y'}; x := \{x, x'\} e
= \{ \text{A.5.5 (definition::-relational-calculus:-end)} \}
\end y ; x := \{x, x'\} e
\]

\[\square\]

A.5.2.6  **law::-relational-calculus:-end:-sequence**

Law A.5.9  (**law::-relational-calculus:-end:-sequence**)  
\[
\text{fora}(P) = y, y' \text{ end } x ; P = \exists x' \bullet P
\]

**Proof**

\[
\text{end } x ; P
= \{ \text{A.5.5 (definition::-relational-calculus:-end)} \}
\]
(∃ x' • II); P
= { A.5.9 (definition::relational-calculus::sequence) }
∃ y₀ • (∃ x' • II)[y₀/y'] ∧ P[y₀/y]
= { A.2.1 (law::predicate-calculus::exists::and::non-free) }
∃ y₀ • (∃ x' • II)[y₀/y'] ∧ P[y₀/y])
= { A.2.5 (law::predicate-calculus::exists::nested) }
∃ x' • (∃ y₀ • II)[y₀/y'] ∧ P[y₀/y])
= { A.5.9 (definition::relational-calculus::sequence) }
∃ x' • II; P
= { A.5.17 (law::relational-calculus::sequence::identity) }
∃ x' • P

□

A.5.2.7 law::relational-calculus::end::var

Law A.5.10 (law::relational-calculus::end::var)

\[ \text{var } x ; \text{end } y = \text{end } y ; \text{var } x \]

Proof

\[ \text{var } x ; \text{end } y \]
= { A.5.33 (law::relational-calculus::var::sequence) }
∃ x • end y
= { A.5.5 (definition::relational-calculus::end) }
∃ x • (∃ x' • II)
= { A.2.1 (law::predicate-calculus::exists::and::non-free) }
∃ x' • (∃ x • II)
= { A.5.10 (definition::relational-calculus::var) }
∃ x' • var x
= { A.5.9 (law::relational-calculus::end::sequence) }
end y ; var x

□

A.5.3 relational-calculus::II

A.5.3.1 definition::relational-calculus::II

Definition A.5.6 (definition::relational-calculus::II)

\[ II \{ x \} \triangleq (x' = x) \]
A.5.3.2 \textit{law::relational-calculus::II::unwinding}

Law A.5.11 \textit{(law::relational-calculus::II::unwinding)}

for \(x\) not in \(A\), and for bounded \(A\)

\[
\Pi(x,y) = \Pi_x \land (y' = y)
\]

\textbf{Proof}

\[
\begin{align*}
\Pi(x,y) \\
= \{ \textbf{A.5.6 (definition::relational-calculus::II) } \} \\
((x', y') = (x, y)) \\
= \{ \textbf{A.3.1 (law::equals::cartesian-pair) } \} \\
(x' = x) \land (y' = y) \\
= \{ \textbf{A.5.6 (definition::relational-calculus::II) } \} \\
\Pi_x \land (y' = y)
\end{align*}
\]

\(\square\)

A.5.3.3 \textit{definition::relational-calculus::II::heterogeneous}

Definition A.5.7 \textit{(definition::relational-calculus::II::heterogeneous)}

\[
\Pi\{x,x', y\} = (\exists y' \bullet \Pi(x,y,x',y'))
\]

A.5.4 \textit{relational-calculus::refinement}

A.5.4.1 \textit{definition::relational-calculus::refinement}

Definition A.5.8 \textit{(definition::relational-calculus::refinement)}

\[
P \sqsubseteq Q \equiv [Q \Rightarrow P]
\]

A.5.4.2 \textit{law::relational-calculus::refinement::nondeterministic-choice}

Law A.5.12 \textit{(law::relational-calculus::refinement::nondeterministic-choice)}

\[
A_1 \cap A_2 \subseteq A_1
\]

\textbf{Proof}

\[
\begin{align*}
A_1 \cap A_2 & \subseteq A_1 \\
= \{ \textbf{A.5.8 (definition::relational-calculus::refinement) } \}
\end{align*}
\]
[\(A_1 \Rightarrow A_1 \cap A_2\)]
= \{ 3.1.5 (definition::-reactive-design:-nondeterministic-choice) \}

[\(A_1 \Rightarrow A_1 \lor A_2\)]
= \{ A.1.11 (law::-propositional-calculus:-or:-introduction) \}
true

\[\square\]

A.5.4.3  \textit{law::-relational-calculus:-refinement:-subsumption}

\textbf{Law A.5.13 (law::-relational-calculus:-refinement:-subsumption)}

\[Q \subseteq R = (Q \cap R = Q)\]

A.5.5 \textit{relational-calculus:-sequence}

A.5.5.1  \textit{definition::-relational-calculus:-sequence}

\textbf{Definition A.5.9 (definition::-relational-calculus:-sequence)}

\[P ; Q \triangleq \exists v_0 \bullet P[v_0/v] \land Q[v_0/v]\text{for}(\text{out} \alpha(P) = \{v\})\text{and}(\text{in} \alpha(Q) = \{v\})\]

A.5.5.2  \textit{law::-relational-calculus:-sequence:-condition-swing}

\textbf{Law A.5.14 (law::-relational-calculus:-sequence:-condition-swing)}

\[P \land b'; Q = P ; b \land Q\]

\textbf{Proof}

\[P \land b'; Q\]
= \{ A.5.9 (definition::-relational-calculus:-sequence) \}
\[\exists x_0 \bullet (P \land b')[x_0/x'] \land Q[x_0/x]\]
= \{ A.7.6 (law::-substitution) \}
\[\exists x_0 \bullet P[x_0/x'] \land b'[x_0/x'] \land Q[x_0/x]\]
= \{ A.7.6 (law::-substitution) \}
\[\exists x_0 \bullet P[x_0/x'] \land b[x_0/x] \land Q[x_0/x]\]
= \{ A.7.6 (law::-substitution) \}
\[\exists x_0 \bullet P[x_0/x'] \land (b \land Q)[x_0/x]\]
= \{ A.5.9 (definition::-relational-calculus:-sequence) \}
\[P ; b \land Q\]

\[\square\]
A.5.5.3 \( \text{law::-relational-calculus:-sequence:-detach-post} \)

Law A.5.15 (\( \text{law::-relational-calculus:-sequence:-detach-post} \))
\[ P ; Q \land c' = (P ; Q) \land c' \]

A.5.5.4 \( \text{law::-relational-calculus:-sequence:-disjunctivity} \)

Law A.5.16 (\( \text{law::-relational-calculus:-sequence:-disjunctivity} \))
\[ P ; Q \lor R = (P ; Q) \lor (P ; R) \]

Proof
\[
P ; Q \lor R
= \begin{cases} 
\text{A.5.9 (definition::-relational-calculus:-sequence)} & \end{cases}
\begin{align*}
\exists x_0 \cdot P[x_0/x'] \land (Q \lor R)[x_0/x] \\
\exists x_0 \cdot (P[x_0/x'] \land Q[x_0/x]) \lor (P[x_0/x'] \land R[x_0/x]) \\
\exists x_0 \cdot (P[x_0/x'] \land Q[x_0/x]) \\
\lor (\exists x_0 \cdot P[x_0/x'] \land R[x_0/x]) \\
= \text{A.5.9 (definition::-relational-calculus:-sequence)}
\end{align*}
\]
\[ (P ; Q) \lor (P ; R) \]

\( \blacksquare \)

A.5.5.5 \( \text{law::-relational-calculus:-sequence:-identity} \)

Law A.5.17 (\( \text{law::-relational-calculus:-sequence:-identity} \))
\[ \Pi ; P = P \]

A.5.5.6 \( \text{law::-relational-calculus:-sequence:-introduction} \)

Law A.5.18 (\( \text{law::-relational-calculus:-sequence:-introduction} \))
\[ P \Rightarrow P ; \text{true} \]

A.5.5.7 \( \text{law::-relational-calculus:-sequence:-left-unit} \)

Law A.5.19 (\( \text{law::-relational-calculus:-sequence:-left-unit} \))
\[ \Pi ; P = P \]
A.5.5.8  \texttt{law::-relational-calculus:-sequence:-left-zero}

Law A.5.20 (\texttt{law::-relational-calculus:-sequence:-left-zero})

\begin{align*}
\textit{false} ; P &= \textit{false}
\end{align*}

A.5.5.9  \texttt{law::-relational-calculus:-sequence:-monotonic-2}

Law A.5.21 (\texttt{law::-relational-calculus:-sequence:-monotonic-2})

\begin{align*}
Q \subseteq R \Rightarrow P ; Q \subseteq P ; R
\end{align*}

\textbf{Proof}  \begin{align*}
\text{assume } Q \subseteq R \text{ first, note that} \\
Q \subseteq R \\
= \{ \text{A.5.13 (law::-relational-calculus:-refinement:-subsumption)} \} \\
Q \cap R = Q
\end{align*}

then,

\begin{align*}
P ; Q \subseteq P ; R \\
= \{ \text{A.5.13 (law::-relational-calculus:-refinement:-subsumption)} \} \\
(P ; Q) \cap (P ; R) = (P ; Q) \\
= \{ \text{A.5.16 (law::-relational-calculus:-sequence:-disjunctivity)} \} \\
P ; (Q \cap R) = (P ; Q) \\
= \{ \text{assumption: } Q \cap R = Q \} \\
P ; Q = P ; Q \\
= \{ \text{A.3.3 (law::-equals:-reflection)} \} \\
\text{true}
\end{align*}

\hfill \Box

A.5.5.10  \texttt{law::-relational-calculus:-sequence:-one-point:-left}

Law A.5.22 (\texttt{law::-relational-calculus:-sequence:-one-point:-left})

\begin{align*}
\text{for}(\alpha(e) = \{ y \}) P \land (x' = e') ; Q = P[e'/x'] ; Q[e/x]
\end{align*}

\textbf{Proof}

\begin{align*}
P \land (x' = e') ; Q \\
= \{ \text{A.5.9 (definition::-relational-calculus:-sequence)} \} \\
\exists x_0, y_0 \bullet (P \land (x' = e'))[x_0, y_0/x, y'] \land Q[x_0, y_0/x, y]
\end{align*}
\[\exists x_0, y_0 \in P[x_0, y_0/x', y'] \land (x' = e')[x_0, y_0/x', y'] \land Q[x_0, y_0/x, y]\]

\[\exists x_0, y_0 \in P[x_0, y_0/x', y'] \land x_0 = e_0 \land Q[x_0, y_0/x, y]\]

\[\exists x_0, y_0 \in P[x_0, y_0/x', y'] \land x_0 = e_0 \land Q[x_0, y_0/x, y]\]

\[\exists x_0, y_0 \in P[x_0, y_0/x', y'] \land x_0 = e_0 \land Q[x_0, y_0/x, y]\]

\[\exists x_0, y_0 \in P[x_0, y_0/x', y'][e_0/x_0] \land Q[x_0, y_0/x, y][e_0/x_0]\]

\[e_0 \mapsto e\]

A.5.5.11 \( \text{law:-relational-calculus:-sequence:-one-point:-right} \)

Law A.5.23 \( \text{law:-relational-calculus:-sequence:-one-point:-right} \)

\[P ; (x = e) \land Q = P[e'/x'] ; Q[e/x]\]

**Proof**

\[P ; (x' = e') \land Q\]

\[\{ A.5.9 \ (\text{definition::relational-calculus::sequence}) \}\]

\[\exists x_0, y_0 \in P[x_0, y_0/x', y'][Q \land (x = e)][x_0, y_0/x, y]\]

\[\{ A.7.6 \ (\text{law::substitution}) \}\]

\[\exists x_0, y_0 \in P[x_0, y_0/x', y'] \land Q[x_0, y_0/x, y] \land (x = e)[x_0, y_0/x, y]\]

\[\{ A.7.6 \ (\text{law::substitution}) \}\]

\[\exists x_0, y_0 \in P[x_0, y_0/x', y'] \land Q[x_0, y_0/x, y] \land x_0 = e_0\]

\[\{ A.2.6 \ (\text{law::predicate-calculus::exists::one-point}) \}\]

\[\exists y_0 \in P[x_0, y_0/x', y'][e_0/x_0] \land Q[x_0, y_0/x, y][e_0/x_0]\]

\[\{ A.7.6 \ (\text{law::substitution}) \}\]

\[\exists y_0 \in P[e_0/x', y'][Q[e_0/x, y]\]
\[
\begin{align*}
\exists y_0 \bullet P[e_0/x][y_0/y'] \land Q[e_0, y_0/x, y] & = \{ \text{A.7.6 (law::substitution)} \} \\
\exists y_0 \bullet P[e'/x'][y_0/y'] \land Q[e/x][y_0/y] & = \{ \text{A.5.9 (definition::relational-calculus::sequence)} \}
\end{align*}
\]

\[P[e'/x'] ; Q[e/x] = \{ \text{A.7.6 (law::substitution)} \}
\]

\[\exists y_0 \bullet P[e'/x'][y_0/y'] \land Q[e/x][y_0/y] = \{ \text{A.5.9 (definition::relational-calculus::sequence)} \}
\]

\[P[e'/x'] ; Q[e/x]
\]

\[\exists y_0 \bullet (b \land P)[x_0/x'] \land Q[x_0/x] = \{ \text{A.5.9 (definition::relational-calculus::sequence)} \}
\]

\[b \land (P ; Q)
\]

\[b \land P ; Q = \{ \text{A.5.9 (definition::relational-calculus::sequence)} \}
\]

\[\exists x_0 \bullet (b \land P)[x_0/x'] \land Q[x_0/x]
\]

\[\exists x_0 \bullet P[x_0/x'] \land (Q \land b')[x_0/x] = \{ \text{A.5.9 (definition::relational-calculus::sequence)} \}
\]

\[\exists x_0 \bullet P[x_0/x'] \land Q[x_0/x] \land b'[x_0/x] = \{ \text{A.7.6 (law::substitution)} \}
\]

\[\exists x_0 \bullet P[x_0/x'] \land Q[x_0/x] \land b' = \{ \text{A.7.6 (law::substitution)} \}
\]

\[\exists x_0 \bullet P[x_0/x'] \land Q[x_0/x] \land b' = \{ \text{A.2.1 (law::predicate-calculus::exists::and::non-free)} \}
\]

\[\exists x_0 \bullet P[x_0/x'] \land Q[x_0/x] \land b' = \{ \text{A.5.9 (definition::relational-calculus::sequence)} \}
\]

\[\exists x_0 \bullet P[x_0/x'] \land Q[x_0/x] \land b'
\]
\[
\exists x_0 \bullet b[x_0/x'] \land P[x_0/x'] \land Q[x_0/x] \\
= \{ \text{A.7.6 (law::substitution)} \} \\
\exists x_0 \bullet b \land P[x_0/x'] \land Q[x_0/x] \\
= \{ \text{A.7.6 (law::substitution)} \} \\
\exists x_0 \bullet b \land P[x_0/x'] \land Q[x_0/x] \\
= \{ \text{A.2.1 (law::predicate-calculus::exists::and::non-free)} \} \\
b \land (\exists x_0 \bullet P[x_0/x'] \land Q[x_0/x]) \\
= \{ \text{A.5.9 (definition::relational-calculus::sequence)} \} \\
b \land (P ; Q)
\]

\[\square\]

A.5.5.14 \textit{law::relational-calculus::sequence::right-zero}

Law A.5.26 \textit{(law::relational-calculus::sequence::right-zero)}

\[P ; \textit{false} = \textit{false}\]

A.5.6 \textit{relational-calculus::var}

A.5.6.1 \textit{definition::relational-calculus::var}

Definition A.5.10 \textit{(definition::relational-calculus::var)}

\[\text{var} \ x \ \triangleq \ \exists \ x \bullet II\]

A.5.6.2 \textit{definition::relational-calculus::var::initialised-declaration}

Definition A.5.11 \textit{(definition::relational-calculus::var::initialised-declaration)}

\[\text{var} \ x := \ e \ \triangleq \ \text{var} \ x ; \ x := \ e\]

A.5.6.3 \textit{law::relational-calculus::var::alphabetised-equation}

Law A.5.27 \textit{(law::relational-calculus::var::alphabetised-equation)}

\[\text{var} \ x = (y' = y)_{(x',y',y')}\]

A.5.6.4 \textit{law::relational-calculus::var::disjoint-parallel}

Law A.5.28 \textit{(law::relational-calculus::var::disjoint-parallel)}

\[(P ; \text{var} \ x) \parallel Q = (P \parallel Q) ; \text{var} \ x \quad \text{for disjoint} \ P, Q \text{ and for} \ x \text{ not in} \ \alpha(P) \text{ or} \ \alpha(Q)\]

\[\text{Proof}\]

71
\[(P ; \text{var} x) \parallel Q\]
\[
\{ \text{A.5.31 (law::relational-calculus::var::scope::left)} \}
\]
\[
(\text{var} x ; P_+x) \parallel Q
\]
\[
\{ \text{A.5.33 (law::relational-calculus::var::sequence)} \}
\]
\[
(\exists x \bullet P_+x) \parallel Q
\]
\[
\{ \text{A.8.1 (definition::concurrency::disjoint-parallel)} \}
\]
\[
(\exists x \bullet P_+x \land Q)
\]
\[
\{ \text{A.2.1 (law::predicate-calculus::exists::and::non-free)} \}
\]
\[
\exists x \bullet (P \land Q)_+x
\]
\[
\{ \text{A.5.33 (law::relational-calculus::var::sequence)} \}
\]
\[
\text{var} x ; (P \land Q)_+x
\]
\[
\{ \text{A.5.31 (law::relational-calculus::var::scope::left)} \}
\]
\[
P \land Q ; \text{var} x
\]
\[
\{ \text{A.8.1 (definition::concurrency::disjoint-parallel)} \}
\]
\[
(P \parallel Q) ; \text{var} x
\]

\[\square\]

A.5.6.5 \text{law::relational-calculus::var::identity}

Law A.5.29 \text{(law::relational-calculus::var::identity)}

\[
\text{var} x ; \text{end} x = \Pi
\]

Proof

\[
\text{var} x ; \text{end} x
\]
\[
\{ \text{A.5.33 (law::relational-calculus::var::sequence)} \}
\]
\[
\exists x \bullet \text{end} x
\]
\[
\{ \text{A.5.5 (definition::relational-calculus::end)} \}
\]
\[
\exists x \bullet (\exists x' \bullet \Pi_{y,x,y})
\]
\[
\{ \text{A.2.1 (law::predicate-calculus::exists::and::non-free)} \}
\]
\[
\exists x, x' \bullet \Pi_{y,x,y}
\]
\[
\{ \text{A.5.11 (law::relational-calculus::II::unwinding)} \}
\]
\[
\exists x, x' \bullet \Pi_y \land (x' = x)
\]
\[
\{ \text{A.2.6 (law::predicate-calculus::exists::one-point)} \}
\]
\[
\exists x \bullet \Pi_y
\]
\[
\{ \text{A.2.1 (law::predicate-calculus::exists::and::non-free)} \}
\]
A.5.6.6  \textit{law::relational-calculus::var::pre-separation}

Law A.5.30 (\textit{law::relational-calculus::var::pre-separation})

\begin{align*}
\text{var } x ; b \land P &= b \land (\text{var } x ; P) \quad \text{for } x \text{ not in } \alpha(b)
\end{align*}

\textbf{Proof}

\begin{align*}
\text{var } x ; b \land P &= \{ \text{A.5.33 (law::relational-calculus::var::sequence)} \} \\
\exists x \cdot b \land P &= \{ \text{A.2.1 (law::predicate-calculus::exists::and::non-free)} \} \\
b \land (\exists x \cdot P) &= \{ \text{A.5.33 (law::relational-calculus::var::sequence)} \} \\
b \land (\text{var } x ; P)
\end{align*}

\hfill \blacksquare

A.5.6.7  \textit{law::relational-calculus::var::scope::left}

Law A.5.31 (\textit{law::relational-calculus::var::scope::left})

\begin{align*}
P ; \text{var } x &= \text{var } x ; P + x
\end{align*}

\textbf{Proof}

\begin{align*}
P ; \text{var } x &= \{ \text{A.5.10 (definition::relational-calculus::var)} \} \\
P ; (\exists x \cdot \Pi (x,y)) &= \{ \text{A.5.9 (definition::relational-calculus::sequence)} \} \\
\exists y_0 \cdot P[y_0/y'] \land (\exists x \cdot \Pi (x,y))[y_0/y] &= \{ \text{A.7.6 (law::substitution)} \} \\
\exists y_0 \cdot P[y_0/y'] \land (\exists x \cdot \Pi (x,y_0)) &= \{ \text{A.2.1 (law::predicate-calculus::exists::and::non-free)} \} \\
\exists x, y_0 \cdot P[y_0/y'] \land \Pi (x,y_0) &= \{ \text{A.5.6 (definition::relational-calculus::II)} \} \\
\exists x, y_0 \cdot P[y_0/y'] \land (x' = x) \land (y' = y_0) &= \{ \text{A.2.6 (law::predicate-calculus::exists::one-point)} \}
\end{align*}
\[ \exists x \cdot P[y/y'] \land (x' = x) \]
\[ = \{ \text{A.7.6 (law::substitution)} \} \]
\[ \exists x \cdot P \land (x' = x) \]
\[ = \{ \text{A.4.1 (definition::alphabet::lifting)} \} \]
\[ \exists x \cdot P_{+x} \]
\[ = \{ \text{A.5.33 (law::relational-calculus::var::sequence)} \} \]
\[ \text{var} \, x \cdot P_{+x} \]

\[ \square \]

A.5.6.8  \textit{law::relational-calculus::var::parallel}

Law A.5.32  \textit{(law::relational-calculus::var::parallel)}

\[ (P \cdot \text{var} \, 0.x) \parallel (Q \cdot \text{var} \, 1.x) = (P \parallel Q) \cdot \text{var} \, 0.x, 1.x \]

A.5.6.9  \textit{law::relational-calculus::var::sequence}

Law A.5.33  \textit{(law::relational-calculus::var::sequence)}

\[ \text{var} \, x \cdot P = \exists x \cdot P \]

A.5.6.10  \textit{law::relational-calculus::var::split}

Law A.5.34  \textit{(law::relational-calculus::var::split)}

\[ \text{var} \, x, y = \text{var} \, x \cdot \text{var} \, y \]

\textbf{Proof}

\[ \text{var} \, x, y = \{ \text{A.5.10 (definition::relational-calculus::var)} \} \]
\[ \exists x, y \cdot \Pi \]
\[ = \{ \text{A.2.1 (law::predicate-calculus::exists::and::non-free)} \} \]
\[ \exists x \cdot (\exists y \cdot \Pi) \]
\[ = \{ \text{A.5.33 (law::relational-calculus::var::sequence)} \} \]
\[ \text{var} \, x \cdot (\exists y \cdot \Pi) \]
\[ = \{ \text{A.5.5 (definition::relational-calculus::end)} \} \]
\[ \text{var} \, x \cdot \text{var} \, y \]

\[ \square \]
A.6 set-theory

A.6.1 law::set-theory::union::left-subset

Law A.6.1 (law::set-theory::union::left-subset)

\[ f \in T \Rightarrow (\{e[f/x]\} \cup \{x : T \cdot e\}) = \{x : T \cdot e\} \]

A.7 substitution

A.7.1 law::substitution::alphabet-restrict

Law A.7.1 (law::substitution::alphabet-restrict)

\[ \text{var } x; P[w/x] = P(w/x) \]

A.7.2 law::relational-calculus::end::substitution

Law A.7.2 (law::relational-calculus::end::substitution)

\[(P \cdot \text{end})[e/y] = P[e'/y']; \text{end} \quad \text{for } \alpha(e) = \{y, z\} \]

Proof

\[
(P_{x,x',y,y',z,z'} \cdot \text{end}_x_{x',y,y',z,z'})[e/y'] \\
= \{ \text{A.5.7 (law::relational-calculus::end::alphabetised)} \} \\
(P_{x,x',y,y',z,z'})((y' = y) \land (z' = z))_{y,y',z,z'}[e/y'] \\
= \{ \text{A.7.6 (law::substitution)} \} \\
(P_{x,x',y,y',z,z'} ((y' = y)_{y,y'} \land (z' = z))_{x,x,z',z'})[e/y'] \\
= \{ \text{A.7.6 (law::substitution)} \} \\
P_{x,x',y,y',z,z'} (y' = y)_{y,y'}[e/y'] \land (z' = z)_{x,x,z',z'}[e/y'] \\
= \{ \text{A.7.6 (law::substitution)} \} \\
P_{x,x',y,y',z,z'} (y' = y)_{y,y'}[e/y'] \land (z' = z)_{x,x,z',z'}[e/y'] \\
= \{ \text{A.7.6 (law::substitution)} \} \\
P_{x,x',y,y',z,z'} (e = y)_{y,y'} \land (z' = z)_{x,x,z',z'}[e/y'] \\
= \{ \text{A.7.6 (law::substitution)} \} \\
P_{x,x',y,y',z,z'} (y = e)_{y,y'} \land (z' = z)_{x,x,z'} \\
= \{ \text{A.5.23 (law::relational-calculus::sequence::one-point::right)} \} \\
P_{x,x',y,y',z,z'}[e'/y'] \cdot (z' = z)_{x,x,z'}[e/y] \\
= \{ \text{A.7.6 (law::substitution)} \} \\
P[e'/y']_{x,x',y,y',z,z'} \cdot (z' = z)_{x,x,z'}[e/y] \]
= \{ \text{A.7.6 (law::-substitution)} \} \\
\text{P}[e'/y']_{\{x,x',y,z,z'\}} ; (z' = z)_{\{x,z,z'\}} \\
= \{ \text{A.7.6 (law::-substitution)} \} \\
\text{P}[e'/y']_{\{x,x',y,z,z'\}} ; (\text{end } x)_{\{x,z,z'\}}
\]

\[
\text{A.7.3 law::-relational-calculus:-II:-ok' substitution}
\]

\[\text{Law A.7.3 (law::-relational-calculus:-II:-ok’ substitution)}
\]

\[
\begin{align*}
&= \{ \text{A.5.11 (law::-relational-calculus:-II:-unwinding), assumption: ok } \notin A \} \\
&= \{ \text{A.7.6 (law::-substitution)} \} \\
&= \{ \text{A.1.3 (law::-propositional-calculus:-and:-elimination)} \} \\
&= \{ \text{false = ok} \} \\
&= \{ \text{A.1.32 (law::-propositional-calculus:-equivalence:-boolean)} \} \\
&= \exists ok
\end{align*}
\]

\[\text{A.7.4 law::-relational-calculus:-assignment:-substitution}
\]

\[\text{Law A.7.4 (law::-relational-calculus:-assignment:-substitution)} \text{ This needs to be generalised to allow its application to a heterogeneous alpha.}
\]

\[
(x,y := \{x',x',y,z,z\} \ e \ f)[g/x'] = (g = e) \land (\text{end } x ; y := \{y,y',z,z\} \ f \ ; \var x)
\]

\[\begin{align*}
&= \{ \text{A.5.1 (definition::-relational-calculus:-assignment)} \} \\
&= \{ \text{A.7.6 (law::-substitution)} \} \\
&= \{ \text{false = ok} \} \\
&= \{ \text{false = ok} \} \\
&= \{ \text{false = ok} \} \\
&= \{ \text{false = ok} \} \\
&= \exists ok
\end{align*}
\]
\[= \{ \text{A.7.6 (law::-substitution)} \}\]
\[(g = e) \land (y' = f) \land (z' = z)\]
\[= \{ \text{A.5.2 (law::-relational-calculus::assignment::cancel-by-end-var)} \}\]
\[(g = e) \land (y := (x,x',y,y',z,z') f ; \text{end } x ; \text{var } x)\]
\[= \{ \text{A.5.1 (law::-relational-calculus::assignment::alphabet-extension)} \}\]
\[(g = e) \land ((x,y := (y,y',z,z') f, x) ; \text{end } x ; \text{var } x)\]
\[= \{ \text{A.5.8 (law::-relational-calculus::end::assignment)} \}\]
\[(g = e) \land (\text{end } x ; y := (y,y',z,z') f ; \text{var } x)\]

\[\square\]

A.7.5 \hspace{1em} \textit{law::-relational-calculus::var::-substitution}

Law A.7.5 \hspace{1em} \textit{(law::-relational-calculus::var::-substitution)}

\[ (\text{var } x ; P)[e/y'] = \text{var } x ; P[e/y'] \]

\textbf{Proof}

\[(\text{var } x ; P)[e/y']\]
\[= \{ \text{A.5.33 (law::-relational-calculus::var::-sequence)} \}\]
\[(\exists x \bullet P)[e/y']\]
\[= \{ \text{A.7.6 (law::-substitution)} \}\]
\[\exists x \bullet P[e/y']\]
\[= \{ \text{A.5.33 (law::-relational-calculus::var::-sequence)} \}\]
\[\text{var } x ; P[e/y']\]

\[\square\]
A.7.6 law::-substitution

Law A.7.6 (law::-substitution)

\[
(P_A)[e/x] = (P[e/x])_A \\
P[v/v] = P \\
P[e/x] = P \\
(e = f)[g/x] = (e[g/x] = f/g/x) \\
P_{+m}[e/v] = P[e/v]_+m \\
P^{-m}[e/v] = P[e/v]_-m \\
(\neg b)[e/x] = \neg b[e/x] \\
(P \land Q)[e/x] = P[e/x] \land Q[e/x] \\
(P \lor Q)[e/x] = P[e/x] \lor Q[e/x] \\
(P \Rightarrow Q)[e/x] = P[e/x] \Rightarrow Q[e/x] \\
(P \Leftrightarrow Q)[e/v] = P[e/v] \Leftrightarrow Q[e/v] \\
(\exists x \cdot P)[e/y] = \exists x \cdot P[e/y] \\
(P ; Q)[e/x] = P[e/x] ; Q \\
(P ; Q)[e/x'] = P[e/x'] \\
(\exists x : s \cdot P)[f] = \exists x : s \cdot P_f \\
(\forall x : s \cdot \neg P)[f] = \forall x : s \cdot \neg P_f \\
\text{lift}(v := w)[e/w] = \text{lift}(v := e)
\]

A.7.7 law::-substitution-2

Law A.7.7 (law::-substitution-2)

\[
P_{+m}[s] = P[s]_{+m} \\
true[s] = true \\
(P \parallel Q)[s] = P[s] \parallel Q[s] \\
(P \land Q)[s] = P[s] \land Q[s] \\
P_{+m}[s] = P[s]_{+m} \\
P^{-m}[s] = P[s]_-m \\
(P ; Q)[v := e] = P[v := e] ; Q \\
(c.e \rightarrow A)[f/x] = (c.e[f/x] \rightarrow A[f/x])
\]

A.8 concurrency

A.8.1 definition::-concurrency::-disjoint-parallel

Definition A.8.1 (definition::-concurrency::-disjoint-parallel)

\[
P \parallel Q \Doteq P \land Q \quad \text{for } \alpha'(P) \cap \alpha'(Q) = \{ok\}
\]

A.8.2 law::-concurrency::-disjoint-parallel:-monotonic

Law A.8.1 (law::-concurrency::-disjoint-parallel:-monotonic)

\[
(P \sqsubset Q) \Rightarrow ((P \parallel R) \sqsubset (Q \parallel R))
\]
Proof

\[(P \parallel R) \subseteq (Q \parallel R)\]
\[= \{ \text{A.5.8 (definition::relational-calculus::refinement)} \} \]
\[\[ (Q \parallel R) \Rightarrow (P \parallel R) \] \]
\[= \{ \text{A.8.1 (definition::concurrency::disjoint-parallel), twice } \} \]
\[Q \land R \Rightarrow P \land R \]
\[= \{ \text{A.1.27 (law::propositional-calculus::implies::and-consequent)} \} \]
\[Q \land R \Rightarrow P \] \[\land (Q \land R \Rightarrow R) \]
\[= \{ \text{A.1.26 (law::propositional-calculus::implies::and-antecedent)} \} \]
\[Q \land R \Rightarrow P \] \[\land ((Q \Rightarrow R) \lor (R \Rightarrow R)) \]
\[= \{ \text{A.1.30 (law::propositional-calculus::implies::identity)} \} \]
\[Q \land R \Rightarrow P \] \[\land ((Q \Rightarrow R) \lor \text{true}) \]
\[= \{ \text{A.1.14 (law::propositional-calculus::or::zero)} \} \]
\[Q \land R \Rightarrow P \] \[\land \text{true} \]
\[= \{ \text{A.1.6 (law::propositional-calculus::and::unit)} \} \]
\[Q \land R \Rightarrow P \]
\[= \{ \text{A.1.26 (law::propositional-calculus::implies::and-antecedent)} \} \]
\[Q \Rightarrow P \] \[\lor (R \Rightarrow P) \]
\[\Rightarrow \{ \text{A.1.9 (law::propositional-calculus::or::elimination)} \} \]
\[Q \Rightarrow P \]
\[= \{ \text{A.5.8 (definition::relational-calculus::refinement)} \} \]
\[P \subseteq Q \]

\[\square\]

A.8.3 law::concurrency::disjoint-parallel::substitution::plain

Law A.8.2 (law::concurrency::disjoint-parallel::substitution::plain)

\[(P \parallel Q)[e/x] = P[e/x] \parallel Q[e/x]\]

Proof

\[(P \parallel Q)[e/x]\]
\[= \{ \text{A.8.1 (definition::concurrency::disjoint-parallel)} \} \]
\[(P \land Q)[e/x]\]
\[= \{ \text{A.7.6 (law::substitution)} \} \]
\[P[e/x] \land Q[e/x]\]
D23.3-4 - CML Definition 2 — Operational Semantics (Public)

A.8.1 (definition::-concurrency:-disjoint-parallel)

\[
P[e/x] \parallel Q[e/x]
\]

\[
\]

A.8.4 law::-concurrency:-disjoint-parallel:-substitution:-separation

Law A.8.3 (law::-concurrency:-disjoint-parallel:-substitution:-separation)

\[(P \parallel Q)[b, c/x', y'] = P[c/x'] \parallel Q[c/y']\]
for \(x'\) not in \(\alpha(Q)\), \(y'\) not in \(\alpha(P)\)

Proof This proof requires a property of substitution: that it is purely syntactic. So \(P[e/x]\)
is well-defined no matter whether \(x\) is in \(P\)'s alphabet or not.

\[
(P \parallel Q)[b, c/x', y']
\]
= \{ A.8.1 (definition::-concurrency:-disjoint-parallel) \}
\[
(P \land Q)[b, c/x', y']
\]
= \{ A.7.6 (law::-substitution) \}
\[
P[b, c/x', y'] \land Q[b, c/x', y']
\]
= \{ substitution, \(x' \notin \alpha(Q)\), \(y' \notin \alpha(P)\) \}
\[
P[c/x'] \land Q[c/y']
\]
= \{ A.8.1 (definition::-concurrency:-disjoint-parallel) \}
\[
P[c/x'] \parallel Q[c/y']
\]
\[
\]

A.9 design

A.9.1 definition::-design

Definition A.9.1 (definition::-design)

\[(P \vdash Q) \equiv ok \land P \Rightarrow ok' \land Q\]
for \(ok, ok'\) not free in \(P, Q\)

A.9.2 definition::-design:-ok'-substitution

Definition A.9.2 (definition::-design:-ok'-substitution)

\[P^b = P[b/ok']\]

A.9.3 definition::-design:-wait:-substitution

Definition A.9.3 (definition::-design:-wait:-substitution)

\[P_w = P[b/wait]\]
A.9.4  law::design::exists

Law A.9.1 (law::design::exists)

\[ \exists x \cdot (P \vdash Q) = (\forall x \cdot P \vdash \exists x \cdot Q) \]

A.9.5  law::design::ok'::substitution::true

Law A.9.2 (law::design::ok'::substitution::true)

\[ (P \vdash Q)^t = ok \land P^t \Rightarrow Q^t \]

A.9.6  law::design::ok'::substitution::false

Law A.9.3 (law::design::ok'::substitution::false)

\[ (P \vdash Q)^f = \neg (ok \land P^f) \]

Proof

\[ (P \vdash Q)^f \]

\[ = \{ \text{A.9.1 (definition::design)} \} \]

\[ (ok \land P \Rightarrow ok' \land Q)^f \]

\[ = \{ \text{A.7.6 (law::substitution)} \} \]

\[ (ok \land P)^f \Rightarrow (ok' \land Q)^f \]

\[ = \{ \text{A.7.6 (law::substitution)} \} \]

\[ ok \land P \Rightarrow (ok' \land Q)^f \]

\[ = \{ \text{A.7.6 (law::substitution)} \} \]

\[ ok \land P \Rightarrow \text{false} \land Q^f \]

\[ = \{ \text{A.1.7 (law::propositional-calculus::and::zero)} \} \]

\[ ok \land P \Rightarrow \text{false} \]

\[ = \{ \text{A.1.28 (law::propositional-calculus::implies::contradiction)} \} \]

\[ \neg (ok \land P) \]

\[ \square \]

A.9.7  law::design::post-ok::Leibniz

Law A.9.4 (law::design::post-ok::Leibniz)

\[ (P \vdash Q) = (P \vdash Q[true/ok]) \]
A.9.8  \textit{law::-design:-post-ok}  

Law A.9.5 (\textit{law::-design:-post-ok})  
\[(P \vdash Q) \equiv (P \vdash \text{ok} \land Q)\]  

Proof  
\[
P \vdash Q  
\quad = \{ \text{A.9.1 (definition::-design)} \}  
\quad \text{ok} \land P \Rightarrow \text{ok}' \land Q  
\quad = \{ \text{A.1.29 (law::-propositional-calculus:-implies:-export)} \}  
\quad \text{ok} \land P \Rightarrow \text{ok} \land \text{ok}' \land Q  
\quad = \{ \text{A.9.1 (definition::-design)} \}  
\quad P \vdash \text{ok} \land Q
\]

\[\square\]

A.9.9  \textit{law::-design:-post:-simplification}  

Law A.9.6 (\textit{law::-design:-post:-simplification})  
\[(P \Rightarrow (Q = R)) \Rightarrow ((P \vdash Q) = (P \vdash R))\]  

A.9.10  \textit{law::-design:-pre-ok:-Leibniz}  

Law A.9.7 (\textit{law::-design:-pre-ok:-Leibniz})  
\[(P \vdash Q) = (P[\text{true}/\text{ok}] \vdash Q)\]  

A.9.11  \textit{law::-design:-pre-ok}  

Law A.9.8 (\textit{law::-design:-pre-ok})  
\[(P \vdash Q) = (\text{ok} \land P \vdash Q)\]  

Proof  
\[
P \vdash Q  
\quad = \{ \text{A.9.1 (definition::-design)} \}  
\quad \text{ok} \land P \Rightarrow \text{ok}' \land Q  
\quad = \{ \text{A.1.4 (law::-propositional-calculus:-and:-idempotence)} \}  
\quad \text{ok} \land \text{ok} \land P \Rightarrow \text{ok}' \land Q  
\quad = \{ \text{A.9.1 (definition::-design)} \}  
\quad \text{ok} \land P \vdash Q
\]

\[\square\]
A.9.12  \textit{law::-design::-refinement::-strengthen-post}

Law A.9.9 (\textit{law::-design::-refinement::-strengthen-post})

\[(Q \Rightarrow R) \Rightarrow (P \vdash R) \subseteq (P \vdash Q)\]

A.9.13  \textit{law::-design::-refinement::-weaken-pre}

Law A.9.10 (\textit{law::-relational-calculus::-refinement::-weaken-pre})

\[(P \Rightarrow Q) \Rightarrow (P \vdash R) \subseteq (Q \vdash P)\]
A.10  \textit{reactive}

A.10.1  \textit{reactive:\-II-\textendash}R

A.10.1.1  \textit{definition::\-reactive::R1}

Definition A.10.1 (\textit{definition::\-reactive::R1})

\[ R1(P) \triangleq P \land tr \leq tr' \]

A.10.1.2  \textit{definition::\-reactive::II-\textendash}R\textendash R1

Definition A.10.2 (\textit{definition::\-reactive::II-\textendash}R\textendash R1)

\[ \Pi_R = R1(\Pi_R) \]

A.10.1.3  \textit{definition::\-reactive::II-\textendash}R

Definition A.10.3 (\textit{definition::\-reactive::II-\textendash}R)

\[ \Pi_R \triangleq (\neg ok \land tr \leq tr') \lor (ok' \land \Pi\{wait, tr, ref, v}\}) \]

A.10.1.4  \textit{definition::\-reactive::R3}

Definition A.10.4 (\textit{definition::\-reactive::R3})

\[ R3(P) \triangleq (\Pi_R \triangleleft wait \triangleright P) \]

A.10.1.5  \textit{definition::\-reactive::R3post}

Definition A.10.5 (\textit{definition::\-reactive::R3post})

\[ R3_{post}(P) = \Pi \triangleleft wait \triangleright P \]

A.10.1.6  \textit{definition::\-reactive::R3pre}

Definition A.10.6 (\textit{definition::\-reactive::R3pre})

\[ R3_{pre}(P) = true \triangleleft wait \triangleright P \]

A.10.1.7  \textit{definition::\-reactive::R}

Definition A.10.7 (\textit{definition::\-reactive::R})

\[ R(P) \triangleq R1 \circ R3(P) \text{assuming} P \text{is} R2 \]
A.10.1.8  \textit{law::-reactive::-R1::-design::-pre-cancellation}

Law A.10.1 (\textit{law::-reactive::-R1::-design::-pre-cancellation})

\[ R1(\neg R1(\neg P) \vdash Q) = R1(P \vdash Q) \]

A.10.1.9  \textit{law::-reactive::-R1::-extension}

Law A.10.2 (\textit{law::-reactive::-R1::-extension})

\[ R1(tr = tr' \cap \langle a \rangle) = (tr' = tr \cap \langle a \rangle) \]

A.10.1.10  \textit{law::-reactive::-J::-splitting}

Law A.10.3 (\textit{law::-reactive::-J::-splitting})

\[ P; J = P' \lor (P' \land ok') \]

\textbf{Proof}

\[ P; J \]
\[ = \{ \textbf{A.10.8} (\textit{definition::-reactive::-J}) \} \]
\[ P; (ok \Rightarrow ok') \land \Pi(A\{ok,ok'\}) \]
\[ = \{ \textbf{A.1.29} (\textit{law::-propositional-calculus:-implies:-export}) \} \]
\[ P; (ok \Rightarrow ok' \land \Pi(A\{ok,ok'\})) \]
\[ = \{ \textbf{A.1.23} (\textit{law::-propositional-calculus:-implies}) \} \]
\[ P; (\neg ok \lor (ok \land ok')) \land \Pi(A\{ok,ok'\}) \]
\[ = \{ \textbf{A.1.5} (\textit{law::-propositional-calculus:-and:-or-distributivity}) \} \]
\[ P; (\neg ok \land \Pi(A\{ok,ok'\})) \lor (ok \land ok' \land \Pi(A\{ok,ok'\})) \]
\[ = \{ \textbf{A.5.16} (\textit{law::-relational-calculus:-sequence:-disjunctivity}) \} \]
\[ (P; \neg ok \land \Pi(A\{ok,ok'\})) \]
\[ \lor (P; ok \land ok' \land \Pi(A\{ok,ok'\})) \]
\[ = \{ \textbf{A.5.14} (\textit{law::-relational-calculus:-sequence:-condition-swing}) \} \]
\[ (P[false/ok'] \Pi(A\{ok,ok'\})) \]
\[ \lor (P; ok \land ok' \land \Pi(A\{ok,ok'\})) \]
\[ = \{ \textbf{A.9.2} (\textit{definition::-design::-ok'::-substitution}) \} \]
\[ (P' \Pi(A\{ok,ok'\})) \]
\[ \lor (P; ok \land ok' \land \Pi(A\{ok,ok'\})) \]
\[ = \{ \textbf{A.5.17} (\textit{law::-relational-calculus:-sequence:-identity}) \} \]
\[ P' \lor (P; ok \land ok' \land \Pi(A\{ok,ok'\})) \]
\[ = \{ \textbf{A.5.14} (\textit{law::-relational-calculus:-sequence:-condition-swing}) \} \]
\[ P^f \lor (P[\text{true}/ok'] ; ok' \land \Pi_{(A\backslash\{ok,ok\})}) \]
\[ = \{ \text{A.5.24 (law::relational-calculus::sequence::post-separation)} \} \]
\[ P^f \lor (P^t ; ok' \land \Pi_{(A\backslash\{ok,ok\})}) \]
\[ = \{ \text{A.9.2 (definition::design::ok'-substitution)} \} \]
\[ P^f \lor ((P^t ; \Pi_{(A\backslash\{ok,ok\})}) \land ok') \]
\[ = \{ \text{A.5.17 (law::relational-calculus::sequence::identity)} \} \]
\[ P^f \lor (P^t \land ok') \]

\[ \Box \]

Definition A.10.8 (definition::reactive::J)
\[ J \triangleq (ok \Rightarrow ok') \land \Pi_{(A\backslash\{ok,ok\})} \quad \text{for } \alpha(J) = A \]

A.10.1.11 law::reactive::R1-R3::commutative

Law A.10.4 (law::reactive::R1-R3::commutative)
\[ R1 \circ R3(P) = R3 \circ R1(P) \]

A.10.1.12 law::reactive::R1-R3post::commutative

Law A.10.5 (law::reactive::R1-R3post::commutative)
\[ R1 \circ R3_{\text{post}}(P) = R3_{\text{post}} \circ R1(P) \]

A.10.1.13 law::reactive::R1::conditional

Law A.10.6 (law::reactive::R1::conditional)
\[ R1(P \triangleleft \text{wait} \triangleright Q) = R1(P) \triangleleft \text{wait} \triangleright R1(Q) \]

Proof
\[ R1(P \triangleleft \text{wait} \triangleright Q) \]
\[ = \{ \text{A.10.1 (definition::reactive::R1)} \} \]
\[ (P \triangleleft \text{wait} \triangleright Q) \land tr \leq tr' \]
\[ = \{ \text{A.1.37 (law::propositional-calculus::conditional::and-distributivity)} \} \]
\[ (P \land tr \leq tr') \triangleleft \text{wait} \triangleright (Q \land tr \leq tr') \]
\[ = \{ \text{A.10.1 (definition::reactive::R1)} \} \]
\[ R1(P) \triangleleft \text{wait} \triangleright R1(Q) \]

\[ \Box \]
A.10.1.14 \textit{law::reactive:-R1:-conjunctive-cancellation}

Law A.10.7 (\textit{law::reactive:-R1:-conjunctive-cancellation})
\[ R_1(P) \land R_1(Q) = R_1(P) \land Q \]

\textbf{Proof}
\begin{align*}
R_1(P) \land R_1(Q) \\
= \{ \text{A.10.1 (definition::reactive:-R1) } \} \\
P \land tr \leq tr' \land Q \land tr \leq tr' \\
= \{ \text{A.1.4 (law::propositional-calculus::and::idempotence) } \} \\
P \land tr \leq tr' \land Q \\
= \{ \text{A.10.1 (definition::reactive:-R1) } \} \\
R_1(P) \land Q
\end{align*}
\[\square\]

A.10.1.15 \textit{law::reactive:-R1:-constant}

Law A.10.8 (\textit{law::reactive:-R1:-constant})
\[ R_1(P_{+tr}) = P_{+tr} \]

A.10.1.16 \textit{law::reactive:-R1:-detach}

Law A.10.9 (\textit{law::reactive:-R1:-detach})
\[ R_1(P \land Q) = R_1(P) \land Q \]

A.10.1.17 \textit{law::reactive:-R1:-disjunctivity}

Law A.10.10 (\textit{law::reactive:-R1:-disjunctivity})
\[ R_1(P \lor Q) = R_1(P) \lor R_1(Q) \]

\textbf{Proof}
\begin{align*}
R_1(P \lor Q) \\
= \{ \text{A.10.1 (definition::reactive:-R1) } \} \\
(P \lor Q) \land tr \leq tr' \\
= \{ \text{A.1.5 (law::propositional-calculus::and::or-distributivity) } \} \\
(P \land tr \leq tr') \lor (Q \land tr \leq tr') \\
= \{ \text{A.10.1 (definition::reactive:-R1) } \} \\
R_1(P) \lor R_1(Q)
\end{align*}
\[\square\]
A.10.1.18  \( \text{law::-reactive:-R1:-exists} \)

Law A.10.11 (\( \text{law::-reactive:-R1:-exists} \))

\[
\exists x \cdot R_1(P) = R_1(\exists x \cdot P) \text{providing } x \text{ is not free in } R_1
\]

A.10.1.19  \( \text{law::-reactive:-R1:-false} \)

Law A.10.12 (\( \text{law::-reactive:-R1:-false} \))

\[
R_1(\text{false}) = \text{false}
\]

A.10.1.20  \( \text{law::-reactive:-R1:-idempotence} \)

Law A.10.13 (\( \text{law::-reactive:-R1:-idempotence} \))

\[
R_1(R_1(P)) = R_1(P)
\]

Proof

\[
R_1(R_1(P)) = \{ \text{A.10.1 (definition::-reactive:-R1), twice } \}
\]

\[
= \{ \text{A.10.1 (definition::-reactive:-R1) } \}
\]

\[
R_1(P)
\]

A.10.1.21  \( \text{law::-reactive:-R1:-II} \)

Law A.10.14 (\( \text{law::-reactive:-R1:-II} \))

\[
R_1(\Pi) = \Pi
\]

A.10.1.22  \( \text{law::-reactive:-R1:-ok'-substitution} \)

Law A.10.15 (\( \text{law::-reactive:-R1:-ok'-substitution} \))

\[
(R_1(P))^f = R_1(P^f)
\]

Proof
\((R1(P))^f\) 
\[
= \{ \text{A.10.1 (definition::-reactive:-R1)} \}
\]
\((P \land tr \leq tr')^f\) 
\[
= \{ \text{A.7.6 (law::-substitution)} \}
\]
\(P^f \land (tr \leq tr')^f\) 
\[
= \{ \text{A.7.6 (law::-substitution)} \}
\]
\(P^f \land tr \leq tr'\) 
\[
= \{ \text{A.10.1 (definition::-reactive:-R1)} \}
\]
\(R1(P^f)\)

\[\Box\]

A.10.1.23  \text{law::-reactive:-R1:-sequence:-closure}

Law A.10.16 (\text{law::-reactive:-R1:-sequence:-closure})

\[R1(P) ; R1(Q) = R1(R1(P) ; R1(Q))\]

A.10.1.24  \text{law::-reactive:-R1:-sequence:-left-zero}

Law A.10.17 (\text{law::-reactive:-R1:-sequence:-left-zero})

\((R1(\neg ok) ; P = R1(\neg ok)\)

A.10.1.25  \text{law::-reactive:-R1:-substitution:-wait'}

Law A.10.18 (\text{law::-reactive:-R1:-substitution:-wait'})

\((R1(P))[b/\text{wait'}] = R1(P[b/\text{wait'}])\)

A.10.1.26  \text{law::-reactive:-R1:-substitution}

Law A.10.19 (\text{law::-reactive:-R1:-substitution})

\((R1(P))[e/x] = R1(P[e/x])\)  \text{providing x is not free in R1}

A.10.1.27  \text{law::-reactive:-R1:-wait-substitution}

Law A.10.20 (\text{law::-reactive:-R1:-wait-substitution})

\((R1(P))_f = R1(P_f)\)
A.10.1.28 \textit{law::-reactive::-R3::-design-split}

Law A.10.21 (\textit{law::-reactive::-R3::-design-split})

\[ R1 \circ R3 (P \vdash Q) = R1(R3_{\text{pre}}(P) \vdash R3_{\text{pre}}(Q)) \]

A.10.1.29 \textit{law::-reactive::-R3::-not-wait::-substitution}

Law A.10.22 (\textit{law::-reactive::-R3::-not-wait::-substitution})

\[ (R3(P))_f = P_f \]

Proof

\[
(R3(P))_f \\
= \{ \text{A.10.4 (definition::-reactive::-R3)} \} \\
= \{ \text{A.7.6 (law::-substitution)} \} \\
= \{ \text{A.1.35 (law::-propositional-calculus:-conditional:-constant:-false)} \} \\
P_f
\]

\[\square\]

A.10.1.30 \textit{law::-reactive::-R3::-not-wait}

Law A.10.23 (\textit{law::-reactive::-R3::-not-wait})

\[ R3(P) = R3(P_f) \]

Proof

\[
R3(P) \\
= \{ \text{A.10.4 (definition::-reactive::-R3)} \} \\
= \{ \text{A.1.33 (law::-propositional-calculus:-conditional:-assumption-else)} \} \\
= \{ \text{A.3.2 (law::-equals:-Leibniz)} \} \\
= \{ \text{A.1.33 (law::-propositional-calculus:-conditional:-assumption-else)} \} \\
\]

\[\square\]
A.10.1.31 \textit{law::-reactive:-R3:-ok'}-substitution

Law A.10.24 (\textit{law::-reactive:-R3:-ok'}-substitution)

\[(R3(P))_f = (\neg ok \land tr \leq tr') \triangleright wait \triangleright P_f\]

\textbf{Proof}

\[
(R3(P))_f = \{ A.10.4 \text{ (definition::-reactive:-R3)} \} \\
(\Pi_R \triangleright wait \triangleright P)_f = \{ A.7.6 \text{ (law::-substitution)} \} \\
\Pi_R \triangleright wait \triangleright P_f = \{ A.10.3 \text{ (definition::-reactive:-II-R)} \} \\
(\Pi \triangleright ok \triangleright tr \leq tr')_f \triangleright wait \triangleright P_f = \{ A.7.6 \text{ (law::-substitution)} \} \\
(\Pi_f \triangleright ok \triangleright tr \leq tr') \triangleright wait \triangleright P_f = \{ A.7.3 \text{ (law::-relational-calculus:-II:-ok'}-substitution) \} \\
(\neg ok \land \Pi_f \triangleright ok \triangleright tr \leq tr') \triangleright wait \triangleright P_f = \{ A.1.34 \text{ (law::-propositional-calculus:-conditional:-assumption-then)} \} \\
(ok \land \neg ok \land \Pi_f \triangleright ok \triangleright tr \leq tr') \triangleright wait \triangleright P_f = \{ A.1.17 \text{ (law::-propositional-calculus:-negation:-contradiction)} \} \\
(false \land \Pi_f \triangleright ok \triangleright tr \leq tr') \triangleright wait \triangleright P_f = \{ A.1.7 \text{ (law::-propositional-calculus:-and:-zero)} \} \\
(false \triangleright ok \triangleright tr \leq tr') \triangleright wait \triangleright P_f = \{ A.1.36 \text{ (law::-propositional-calculus:-conditional:-constant:-then)} \} \\
(\neg ok \land tr \leq tr') \triangleright wait \triangleright P_f
\]

\[\square\]

A.10.1.32 \textit{law::-reactive:-R3:-sequence-closure}

Law A.10.25 (\textit{law::-reactive:-R3:-sequence-closure})

\[\text{for } P, Q \text{ } R3\text{-healthy } P ; Q = R3(P ; Q)\]

A.10.1.33 \textit{law::-reactive:-R3post:-wait:-false}

Law A.10.26 (\textit{law::-reactive:-R3post:-wait:-false})

\[(R3_{post}(P))_f = P\]
A.10.1.34 \textit{law::-reactive:-R3post:-wait:-true}

Law A.10.27 (\textit{law::-reactive:-R3post:-wait:-true})

\[
(R3_{\text{post}}(P))_t = \top
\]

A.10.1.35 \textit{law::-reactive:-R3pre:-wait:-false}

Law A.10.28 (\textit{law::-reactive:-R3pre:-wait:-false})

\[
(R3_{\text{pre}}(P))_f = P
\]

A.10.1.36 \textit{law::-reactive:-R3pre:-wait:-true}

Law A.10.29 (\textit{law::-reactive:-R3pre:-wait:-true})

\[
(R3_{\text{pre}}(P))_t = \text{true}
\]

A.10.2 \textit{wp-R1}

A.10.2.1 \textit{definition::-wp-R1}

Definition A.10.9 (\textit{definition::-wp-R1})

\[
P_{\text{wp}} \ R1 Q = \neg (P ; R1(Q))
\]

A.10.2.2 \textit{law::-reactive:-wp-R1:-simplification-1}

Law A.10.30 (\textit{law::-reactive:-wp-R1:-simplification-1})

\[
(P \land \text{wait'})_{\text{wp}} R1 R3_{\text{pre}}(Q) = \text{true}
\]

A.10.2.3 \textit{law::-reactive:-wp-R1:-conditional}

Law A.10.31 (\textit{law::-reactive:-wp-R1:-conditional})

\[
(P \triangleleft b \triangleright Q)_{\text{wp}} R1 R = ((P \land b)_{\text{wp}} R1 Q) \land ((P \land \neg b)_{\text{wp}} R1 Q)
\]
A.11 reactive-designs

A.11.1 law::reactive-design::assignment::declaration

Law A.11.1 (law::reactive-design::assignment::declaration)

\[ \text{var} x : x :=_{RD} w \ = \ \text{var} x :=_{RD} w \]

A.11.2 law::reactive-design::assignment::simple

Law A.11.2 (law::reactive-design::assignment::simple)

\[ x :=_{RD} w \ = \ \text{lift}(x := w) \]

A.11.3 law::reactive-design::CSP3

Law A.11.3 (law::reactive-design::CSP3)

\[ R1 \circ R3(\exists \text{ref} \cdot (P \vdash Q)) = \text{CSP3} \circ R1 \circ R3(P \vdash Q) \]

A.11.4 law::reactive-design::declaration

Law A.11.4 (law::reactive-design::declaration)

\[ \text{var}_{RD} x ; P \ = \ \text{var} x ; P \]

A.11.5 law::reactive-design::decl::elimination

Law A.11.5 (law::reactive-design::decl::elimination)

\[ \text{var}_{RD} x :=_{RD} w ; P \ = \ P(w/x) \]

Providing \( x \) does not appear in expression \( w \)

Proof

\[ \text{var}_{RD} x :=_{RD} w ; P \]
\[ = \ \{ \text{A.5.11 (definition::relational-calculus::var::initialised-Declaration)} \} \]
\[ \text{var}_{RD} x ; x :=_{RD} w ; P \]
\[ = \ \{ \text{A.11.4 (law::reactive-design::declaration)} \} \]
\[ \text{var} x ; x :=_{RD} w ; P \]
\[ = \ \{ \text{A.11.2 (law::reactive-design::assignment::simple)} \} \]
\[ \text{var} x ; \text{lift}(x := w) ; P \]
\[ = \ \{ \text{3.2.10 (law::lift::leading::substitution)} \} \]
\text{var} \ x; \ P[w/x] \\
= \{ \ \text{A.7.1 (law::substitution::alphabet-restrict)} \} \\
P(w/x)
\square

A.11.6 \text{law::reactive-design::external-choice::distributed::associativity}

Definition A.11.1 \text{(law::reactive-design::external-choice::distributed::associativity)}

\[ P \prod S \equiv R_1 \circ R_3(\neg P_f^i \land \neg (\prod S)^f_j \models P_f^i \land (\prod S)^f_j \land (tr' = tr) \land \text{wait'} \models P_f^i \lor (\prod S)^f_j) \]

A.11.7 \text{law::reactive-design::lift}

Law A.11.6 \text{(law::reactive-design::lift)}

\[ \text{lift}(s) ; \ R_1 \circ R_3(P \models Q) = R_1 \circ R_3(P[s] \models Q[s]) \]

A.11.8 \text{law::reactive-design::parallel-by-merge}

Law A.11.7 \text{(law::reactive-design::parallel-by-merge)}

\[ R_1 \circ R_3(\neg (P_f^i) \models P_f^i) ||_M R_1 \circ R_3(\neg (Q_f^j) \models Q_f^j) \]
\[ = R_1 \circ R_3(\neg (P_f^i ||_M Q_f^j) \land \neg (P_f^i ||_M Q_f^j) \models (P_f^i ||_M Q_f^j)) \]

A.11.9 \text{law::reactive-design::prefix}

Law A.11.8 \text{(law::reactive-design::prefix)}

\[ a \to P = \]
\[ \text{CSP3} \circ R_1 \circ R_3(tr \models \langle a \rangle) \leq tr' \Rightarrow \neg P_f^i[tr \models \langle a \rangle / tr] \]
\[ \vdash (a \notin \text{ref'} \land \text{wait'})_{tr,v} \lor P_f^i[tr \models \langle a \rangle / tr] \]

Proof

\[ l \to P \]
\[ = \{ \ \text{3.1.4 (definition::reactive-design::prefix)} \} \]
\[ l \to \text{Skip} ; P \]
\[ = \{ \ \text{3.1.3 (definition::reactive-design::simple-prefix)} \} \]
\[ R_1 \circ R_3(\text{true}) \vdash ((tr' = tr) \land a \notin \text{ref'} \models \text{wait'} \models tr' = tr \models \langle a \rangle)_{tr,v} ; P \]
\[ = \{ \ \text{A.11.13 (law::reactive-design)} \} \]
\[ \text{R1} \circ \text{R3}(\text{true}) = \left((\text{tr}' = \text{tr}) \land a \not\in \text{ref}' \land \text{wait}' \Rightarrow \text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right); \]
\[ \text{R1} \circ \text{R3}(\neg P^f_{+} \lor P^f_{+}) \]
\[ = \{ 3.1.2 \text{(definition::reactive-design::sequence)} \} \]
\[ \text{R1} \circ \text{R3}(\left((\text{tr}' = \text{tr}) \land a \not\in \text{ref}' \land \text{wait}' \Rightarrow \text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right)_+ \text{wp}_{\text{R1,R3} \text{pre}(\neg P^f_{+})} \]
\[ \vdash \text{R1}(\left((\text{tr}' = \text{tr}) \land a \not\in \text{ref}' \land \text{wait}' \Rightarrow \text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right)_+ ; \text{R3} \text{post}(P^f_{+}) \]
\[ = \{ \text{A.10.31} \text{(law::reactive::wp-R1::conditional)} \} \]
\[ \text{R1} \circ \text{R3}(\left((\text{tr}' = \text{tr}) \land a \not\in \text{ref}' \land \text{wait}' \Rightarrow \text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right)_+ \text{wp}_{\text{R1,R3} \text{pre}(\neg P^f_{+})} \]
\[ \land \left((\text{tr}' = \text{tr} \land \langle a \rangle \land \neg \text{wait}' \land \text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right)_+ \text{wp}_{\text{R1,R3} \text{pre}(\neg P^f_{+})} \]
\[ \vdash \text{R1}(\left((\text{tr}' = \text{tr}) \land a \not\in \text{ref}' \land \text{wait}' \Rightarrow \text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right)_+ ; \text{R3} \text{post}(P^f_{+}) \]
\[ = \{ \text{A.10.30} \text{(law::reactive::wp-R1::simplification)} \} \]
\[ \text{R1} \circ \text{R3}(\text{true} \land \left((\text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right)_+ \text{wp}_{\text{R1} \neg P^f_{+}} \]
\[ \vdash \text{R1}(\left((\text{tr}' = \text{tr}) \land a \not\in \text{ref}' \land \text{wait}' \Rightarrow \text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right)_+ ; \text{R3} \text{post}(P^f_{+}) \]
\[ = \{ \text{A.1.6} \text{(law::propositional-calculus::and::unit)} \} \]
\[ \text{R1} \circ \text{R3}(\left((\text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right)_+ \text{wp}_{\text{R1} \neg P^f_{+}} \]
\[ \vdash \text{R1}(\left((\text{tr}' = \text{tr}) \land a \not\in \text{ref}' \land \text{wait}' \Rightarrow \text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right)_+ ; \text{R3} \text{post}(P^f_{+}) \]
\[ = \{ \text{A.1.1} \text{(definition::propositional-calculus::conditional)} \} \]
\[ \text{R1} \circ \text{R3}(\left((\text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right)_+ \text{wp}_{\text{R1} \neg P^f_{+}} \]
\[ \vdash \text{R1}(\left((\text{tr}' = \text{tr}) \land a \not\in \text{ref}' \land \text{wait}' \Rightarrow \text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right)_+ ; \text{R3} \text{post}(P^f_{+}) \]
\[ = \{ \text{A.10.10} \text{(law::reactive::R1::disjunctivity)} \} \]
\[ \text{R1} \circ \text{R3}(\left((\text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right)_+ \text{wp}_{\text{R1} \neg P^f_{+}} \]
\[ \vdash \text{R1}(\left((\text{tr}' = \text{tr}) \land a \not\in \text{ref}' \land \text{wait}' \Rightarrow \text{tr}' = \text{tr} \land \langle a \rangle_{+v}\right)_+ ; \text{R3} \text{post}(P^f_{+}) \]
\[ = \{ \text{A.10.10} \text{(law::reactive::R1::disjunctivity)} \} \]
\[(\text{R1}(\text{tr} = \text{tr'}) \land a \notin \text{ref'} \land \text{wait}') \lor \text{R1}(\text{tr} = \text{tr} \land \text{wait}')_{+v} ; \text{R3post}(P_f^{i}))\]

\[= \{ \text{A.4.1 (law::alphabet::lifting::disjunctivity)} \}\]

\[\text{R1} \circ \text{R3}(\text{tr} = \text{tr} \land \text{a} \notin \text{ref'} \land \text{wait}')_{+v} ; \text{R1} \neg P_f^{i} \]

\[\vdash \text{R1}(\text{tr} = \text{tr'} \land a \notin \text{ref'} \land \text{wait}')_{+v} \lor \text{R1}(\text{tr} = \text{tr} \land \text{wait}')_{+v} ; \text{R3post}(P_f^{i})\]

\[= \{ \text{A.5.16 (law::relational-calculus::sequence::disjunctivity)} \}\]

\[\text{R1} \circ \text{R3}(\text{tr} = \text{tr} \land \text{a} \notin \text{ref'} \land \text{wait}')_{+v} \]

\[\vdash \text{R1}(\text{tr} = \text{tr'} \land a \notin \text{ref'} \land \text{wait}')_{+v} ; \text{R3post}(P_f^{i})\]

\[= \{ \text{A.5.22 (law::relational-calculus::sequence::one-point::left)} \}\]

\[\text{R1} \circ \text{R3}(\text{tr} = \text{tr} \land \text{a} \notin \text{ref'} \land \text{wait}')_{+v} \]

\[\vdash \text{R1}(\text{tr} = \text{tr'} \land a \notin \text{ref'} \land \text{wait}')_{+v} ; \text{R3post}(P_f^{i})\]

\[= \{ \text{A.10.5 (definition::reactive::R3post)} \}\]

\[\text{R1} \circ \text{R3}(\text{tr} = \text{tr} \land \text{a} \notin \text{ref'} \land \text{wait}')_{+v} \]

\[\vdash \text{R1}(\text{tr} = \text{tr'} \land a \notin \text{ref'} \land \text{wait}')_{+v} ; \text{R3post}(P_f^{i})\]

\[= \{ \text{A.5.6 (definition::relational-calculus::II)} \}\]

\[\text{R1} \circ \text{R3}(\text{tr} = \text{tr} \land \text{a} \notin \text{ref'} \land \text{wait}')_{+v} \]

\[\vdash \text{R1}(\text{tr} = \text{tr'} \land a \notin \text{ref'} \land \text{wait}')_{+v} ; \text{R3post}(P_f^{i})\]

\[= \{ \text{A.10.1 (definition::reactive::R1)} \}\]

\[\text{R1} \circ \text{R3}(\text{tr} = \text{tr} \land \text{a} \notin \text{ref'} \land \text{wait}')_{+v} \]

\[\vdash \text{R1}(\text{tr} = \text{tr'} \land a \notin \text{ref'} \land \text{wait}')_{+v} ; \text{R3post}(P_f^{i})\]
\[ \neg (a \notin \text{ref}' \land \text{wait}') \lor ((\text{tr}' = \text{tr} \land \langle a \rangle)_{+,v} ; P_f) \]

Continuing with the precondition:

\[
(\text{tr}' = \text{tr} \land \langle a \rangle)_{+,v} \triangleright P_f
\]

\[= \{ \text{A.10.9 (definition::-wp-R1)} \}\]

\[\neg (R1(\text{tr}' = \text{tr} \land \langle a \rangle)_{+,v} ; R1(\neg \neg P_f))\]

\[= \{ \text{A.10.2 (law::reactive::R1::extension)} \}\]

\[\neg ((\text{tr}' = \text{tr} \land \langle a \rangle)_{+,v} ; R1(\neg \neg P_f))\]

\[= \{ \text{A.1.20 (law::propositional-calculus::negation::double-negation)} \}\]

\[\neg ((\text{tr}' = \text{tr} \land \langle a \rangle)_{+,v} ; P_f \land \text{tr} \leq \text{tr}')\]

\[= \{ \text{A.5.9 (definition::-relational-calculus::sequence)} \}\]

\[\neg (\exists \text{tr}_0, \text{ref}_0, v_0 \bullet \text{tr}_0 = \text{tr} \land \langle a \rangle \land v_0 = v \land P_f[\text{tr}_0, \text{ref}_0, v_0/\text{tr}, \text{ref}, v] \land \text{tr}_0 \leq \text{tr}')\]

\[= \{ \text{A.2.6 (law::predicate-calculus::exists::one-point), twice } \}\]

\[\neg (\exists \text{ref}_0 \bullet P_f[\text{tr} \land \langle a \rangle/\text{tr}, \text{ref}] \land \text{tr} \land \langle a \rangle \leq \text{tr}')\]

\[= \{ \text{A.2.8 (law::predicate-calculus::exists::rename-bound-var)} \}\]

\[\forall \text{ref} \bullet \neg (P_f[\text{tr} \land \langle a \rangle/\text{tr} \land \text{tr} \land \langle a \rangle \leq \text{tr}')\]

\[= \{ \text{A.1.31 (law::propositional-calculus::implies::negation) } \}\]

\[\forall \text{ref} \bullet \text{tr} \land \langle a \rangle \leq \text{tr}' \Rightarrow \neg P_f[\text{tr} \land \langle a \rangle/\text{tr}]\]

\[(a \notin \text{ref}' \land \text{wait}')_{+,v} \lor (((\text{tr}' = \text{tr} \land \langle a \rangle)_{+,v} ; P_f)\]

And now continuing with the postcondition:

\[\exists \text{ref} \bullet (a \notin \text{ref}' \land \text{wait}')_{+,v} \lor (((\text{tr}' = \text{tr} \land \langle a \rangle)_{+,v} ; P_f)\]

\[= \{ \text{A.5.9 (definition::-relational-calculus::sequence)} \}\]

\[\exists \text{ref} \bullet (a \notin \text{ref}' \land \text{wait}')_{+,v} \lor (\exists \text{tr}_0, \text{ref}_0, v_0 \bullet \text{tr}_0 = \text{tr} \land \langle a \rangle \land v_0 = v \land P_f[\text{tr}_0, \text{ref}_0, v_0/\text{tr}, \text{ref}, v])\]

\[= \{ \text{A.2.6 (law::predicate-calculus::exists::one-point), twice } \}\]

\[\exists \text{ref}_0 \bullet (a \notin \text{ref}' \land \text{wait}')_{+,v} \lor (\exists \text{ref}_0 \bullet P_f[\text{tr} \land \langle a \rangle/\text{tr}]\]

97
\[ \exists \text{ref} \bullet (a \notin \text{ref}' \land \text{wait}')_{+tr,v} \lor P_f^t[\text{tr} \smallfrown \langle a \rangle / \text{tr}] \]

Putting it all together:

\[ R1 \circ R3( \\neg P_f^t[\text{tr} \smallfrown \langle a \rangle / \text{tr}] \) = { \text{previous results} } \]

\[ R1 \circ R3( \exists \text{ref} \bullet (a \notin \text{ref}' \land \text{wait}')_{+tr,v} \lor P_f^t[\text{tr} \smallfrown \langle a \rangle / \text{tr}] \) = { A.9.1 (law::-design:-exists) } \]

\[ R1 \circ R3( \exists \text{ref} \bullet (\text{tr} \smallfrown \langle a \rangle \leq \text{tr}' \Rightarrow \neg P_f^t[\text{tr} \smallfrown \langle a \rangle / \text{tr}] \) = { A.11.3 (law::-reactive-design:-CSP3) } \]

\[ CSP3 \circ R1 \circ R3( \text{tr} \smallfrown \langle a \rangle \leq \text{tr}' \Rightarrow \neg P_f^t[\text{tr} \smallfrown \langle a \rangle / \text{tr}] \) = \]

A.11.10  \text{law::-reactive-design:-R1:-cancellation:-pre}

Law A.11.9 (law::-reactive-design:-R1:-cancellation:-pre)

\[ R1(P \land \neg R1(Q) \vdash R) = R1(P \land \neg Q \vdash R) \]

A.11.11  \text{law::-reactive-design:-R1:-cancellation:-post}

Law A.11.10 (law::-reactive-design:-R1:-cancellation:-post)

\[ R1(P \vdash R1(Q)) = R1(P \vdash Q) \]
A.11.12  \textit{law::-reactive-design:-sequence:-R1-R3} \\

Law A.11.11  \textit{(law::-reactive-design:-sequence:-R1-R3)}

\[ R_1 \circ R_3(P \vdash Q) ; R_1 \circ R_3(R \vdash S) = \]
\[ R_1 \circ R_3(\neg (R_1(\neg P) ; R_1(\text{true})) \land \neg (R_1(Q) \land \neg \text{wait} \land R_1(\neg R))) \]
\[ \vdash R_1(Q) ; (\not\not \text{wait} \triangleright R_1(S)) \]

Proof

\[ R_1 \circ R_3(P \vdash Q) ; R_1 \circ R_3(R \vdash S) \]
\[ = \{ \text{A.11.12 (law::-reactive-design:-sequence:-R1)} \} \]
\[ R_1(\neg (R_1(\neg R_3\text{pre}(P)) ; R_1(\text{true})) \land \neg (R_1 \circ R_3\text{post}(Q) ; R_1(\neg R_3\text{pre}(R)))) \]
\[ \vdash R_1 \circ R_3\text{post}(Q) ; R_1 \circ R_3\text{post}(S) \]
\[ = \{ \text{A.9.1 (definition::-design)} \} \]
\[ R_1(\neg \text{ok} \lor (1) \lor (2) \lor (\text{ok}' \land (3))) \]
\[ = (**) \]
\[ (1) \]
\[ = \{ \text{sub-law } \} \]
\[ R_1(\neg R_3\text{pre}(P)) ; R_1(\text{true}) \]
\[ = \{ \text{A.1.41 (law::-propositional-calculus:-conditional:-idempotence)} \} \]
\[ R_1(\neg R_3\text{pre}(P)) ; R_1(\text{true}) \not\not \text{wait} \triangleright R_1(\neg R_3\text{pre}(P)) ; R_1(\text{true}) \]
\[ = \{ \text{A.1.39 (law::-propositional-calculus:-conditional:-export:-then)} \} \]
\[ \{ \text{A.1.40 (law::-propositional-calculus:-conditional:-export:-else)} \} \]
\[ (R_1(\neg R_3\text{pre}(P)) ; R_1(\text{true}))_{\not\not \text{wait} \triangleright (R_1(\neg R_3\text{pre}(P)) ; R_1(\text{true}))} \]
\[ = \{ \text{A.7.6 (law::-substitution), twice } \} \]
\[ (R_1(\neg R_3\text{pre}(P)))_{\not\not \text{wait} \triangleright (R_1(\neg R_3\text{pre}(P)))} ; R_1(\text{true}) \]
\[ = \{ \text{A.10.19 (law::-reactive:-R1:-substitution): wait not free in R1, twice } \} \]
\[ R_1((\neg R_3\text{pre}(P)))_{\not\not \text{wait} \triangleright (R_1(\neg R_3\text{pre}(P)))} ; R_1(\text{true}) \]
\[ = \{ \text{A.10.29 (law::-reactive:-R3pre:-wait:-true), A.10.28 (law::-reactive:-R3pre:-wait:-false)} \} \]
\[ R_1(\neg \text{true}) ; R_1(\text{true}) \not\not \text{wait} \triangleright R_1(\neg P) ; R_1(\text{true}) \]
\[ = \{ \text{A.1.16 (law::-propositional-calculus:-negation:-true)} \} \]
\[ R_1(\text{false}) ; R_1(\text{true}) \not\not \text{wait} \triangleright R_1(\neg P) ; R_1(\text{true}) \]
\[ = \{ \text{A.10.12 (law::-reactive:-R1:-false)} \} \]
\[ \text{false} ; R_1(\text{true}) \not\not \text{wait} \triangleright R_1(\neg P) ; R_1(\text{true}) \]
\[ = \{ \text{A.5.20 (law::-relational-calculus:-sequence:-left-zero)} \} \]
false \iff\ wait \triangleright R1(\neg \ P) ; R1(true)

(2)

= \{ \text{sub-law} \}

R1 \circ R3_{\text{pre}}(Q) ; R1(\neg R3_{\text{pre}}(R))

= \{ \text{A.1.41 (law::propositional-calculus::conditional::idempotence)} \}

R1 \circ R3_{\text{post}}(Q) ; R1(\neg R3_{\text{pre}}(R)) \iff\ wait \triangleright R1 \circ R3_{\text{post}}(Q) ; R1(\neg R3_{\text{pre}}(R))

= \{ \text{A.1.39 (law::propositional-calculus::conditional::export::then)} \}

A.1.40 (law::propositional-calculus::conditional::export::else)

(R1 \circ R3_{\text{post}}(Q) ; R1(\neg R3_{\text{pre}}(R))) \iff\ wait \triangleright (R1 \circ R3_{\text{post}}(Q) ; R1(\neg R3_{\text{pre}}(R)))

= \{ \text{A.7.6 (law::substitution), twice} \}

(R1 \circ R3_{\text{post}}(Q)) ; R1(\neg R3_{\text{pre}}(R)) \iff\ wait \triangleright (R1 \circ R3_{\text{post}}(Q) ; R1(\neg R3_{\text{pre}}(R)))

= \{ \text{A.10.19 (law::reactive::R1::substitution): wait not free in R1, twice} \}

R1((R3_{\text{post}}(Q))) ; R1(\neg R3_{\text{pre}}(R)) \iff\ wait \triangleright R1((R3_{\text{post}}(Q))) ; R1(\neg R3_{\text{pre}}(R))

= \{ \text{A.10.27 (law::reactive::R3_{\text{post}}::wait::true), A.10.26 (law::reactive::R3_{\text{post}}::wait::false)} \}

R1(\neg R3_{\text{pre}}(R)) \iff\ wait \triangleright R1(Q) ; R1(\neg R3_{\text{pre}}(R))

= \{ \text{A.10.14 (law::reactive::R1::if)} \}

\iff\ R1(\neg R3_{\text{pre}}(R)) \iff\ wait \triangleright R1(Q) ; R1(\neg R3_{\text{pre}}(R))

= \{ \text{A.1.39 (law::propositional-calculus::conditional::export::then)} \}

(R1(\neg R3_{\text{pre}}(R))) \iff\ wait \triangleright R1(Q) ; R1(\neg R3_{\text{pre}}(R))

= \{ \text{A.10.19 (law::reactive::R1::substitution): wait not free in R1} \}

R1(\neg R3_{\text{pre}}(R)) \iff\ wait \triangleright R1(\neg R3_{\text{pre}}(R))

= \{ \text{A.10.29 (law::reactive::R3_{\text{pre}}::wait::true)} \}

R1(\text{true}) \iff\ wait \triangleright R1(Q) ; R1(\neg R3_{\text{pre}}(R))

= \{ \text{A.1.16 (law::propositional-calculus::negation::true)} \}

R1(\text{false}) \iff\ wait \triangleright R1(Q) ; R1(\neg R3_{\text{pre}}(R))

= \{ \text{A.10.12 (law::reactive::R1::false)} \}

false \iff\ wait \triangleright R1(Q) ; R1(\neg R3_{\text{pre}}(R))

= \{ \text{spawn sub-law} \}

false \iff\ wait \triangleright (4)

(4)

= \{ \text{sub-law} \}

R1(Q) ; R1(\neg R3_{\text{pre}}(R))

= \{ \text{A.1.21 (law::propositional-calculus::negation::excluded-middle)} \}

R1(Q) \land (\text{wait}' \lor \neg \text{wait}') ; R1(\neg R3_{\text{pre}}(R))

= \{ \text{A.5.16 (law::relational-calculus::sequence::disjunctivity)} \}

(R1(Q) \land \text{wait}'; R1(\neg R3_{\text{pre}}(R))) \lor (R1(Q) \land \neg \text{wait}'; R1(\neg R3_{\text{pre}}(R)))
\[
\begin{align*}
\text{法律规定2:} & \quad (A.5.22) \quad \text{逻辑代数:序列:单点:右零} \\
& \quad (R1(Q))\text{[true/等待']} \land (R1(\neg R3\text{pre}(R))) \lor ((R1(Q))\text{[false/等待']} \land (R1(\neg R3\text{pre}(R)))) \\
& \quad (A.10.18) \quad \text{反应:R1:替换:等待'} \\
& \quad (R1(Q))\text{[true/等待']} \land (\neg (R3\text{pre}(R))) \lor ((R1(Q))\text{[false/等待']} \land (R1(\neg (R3\text{pre}(R)))) \\
& \quad (A.10.29) \quad \text{反应:R3pre:等待:真}, \quad (A.10.28) \quad \text{反应:R3pre:等待:假} \\
& \quad (R1(Q))\text{[true/等待']} \land (R1(\neg \text{true})) \lor ((R1(Q))\text{[false/等待']} \land (R1(\neg R_f)) \\
& \quad (A.1.16) \quad \text{逻辑代数:否定:真} \\
& \quad (R1(Q))\text{[false/等待']} \land (R1(\neg R_f)) \\
& \quad (A.1.16) \quad \text{逻辑代数:否定:真} \\
& \quad (R1(Q))\text{[true/等待']} \land (\text{false}) \lor ((R1(Q))\text{[false/等待']} \land (R1(\neg R_f)) \\
& \quad (A.5.26) \quad \text{逻辑代数:序列:右零} \\
& \quad \text{false} \lor ((R1(Q))\text{[false/等待']} \land (R1(\neg R_f)) \\
& \quad (A.1.13) \quad \text{逻辑代数:或:单元} \\
& \quad (R1(Q))\text{[false/等待']} \land (R1(\neg R_f)) \\
& \quad (A.5.22) \quad \text{逻辑代数:序列:单点:左} \\
& \quad R1(Q) \land (\neg \text{等待'}) \land (R1(\neg R)) \\
& \quad (1) \lor (2) \\
& \quad \text{sub-laws} \\
& \quad (\text{false} \land \text{等待'} \lor R1(\neg P) \lor R1(\text{true})) \lor (\text{false} \land \text{等待'} \lor R1(Q) \land (R1(\neg R_3\text{pre}(R))) \\
& \quad (\text{false} \land \text{等待'} \lor R1(\neg P) \lor R1(\text{true})) \lor (\text{false} \land \text{等待'} \lor R1(\neg R) \lor R1(\neg R_f)) \\
& \quad (A.1.38) \quad \text{逻辑代数:条件:交换:或} \\
& \quad (\text{false} \lor \text{false} \land \text{等待'} \lor (R1(\neg P) \lor R1(\text{true})) \lor (R1(Q) \land (\neg \text{等待'}) \lor (R1(\neg R))) \\
& \quad (A.1.38) \quad \text{逻辑代数:条件:交换:或} \\
& \quad (\text{false} \land \text{等待'} \lor (R1(\neg P) \lor R1(\text{true})) \lor (R1(Q) \land (\neg \text{等待'}) \lor (R1(\neg R))) \\
& \quad (A.1.20) \quad \text{逻辑代数:否定:双负} \\
& \quad (\text{true} \land \text{等待'} \lor (R1(\neg P) \lor R1(\text{true})) \lor (R1(Q) \land (\neg \text{等待'}) \lor (R1(\neg R))) \\
& \quad (A.1.42) \quad \text{逻辑代数:条件:否定} \\
& \quad (\text{true} \land \text{等待'} \lor (R1(\neg P) \lor R1(\text{true})) \lor (R1(Q) \land (\neg \text{等待'}) \lor (R1(\neg R))) \\
& \quad (A.1.10) \quad \text{定义:反应:R3pre} \\
& \quad (\neg R3\pre(\neg (R1(\neg P) \lor R1(\text{true})) \lor (R1(Q) \land (\neg \text{等待'}) \lor (R1(\neg R)))) \\
& \quad (A.10.6) \quad (3) \\
& \quad (\text{false} \land \text{等待'}) \land (R1(Q) \land (\neg \text{等待'}) \lor (R1(\neg R))) \\
& \quad \text{sub-law} \\
& \quad R1 \circ R3\text{post}(Q) \land R1 \circ R3\text{post}(S) \\
& \quad (A.1.41) \quad \text{逻辑代数:条件:双重对称性} \\
& \quad R1 \circ R3\text{post}(Q) \land R1 \circ R3\text{post}(S) \land \text{等待'} \lor R1 \circ R3\text{post}(Q) \land R1 \circ R3\text{post}(S) \\
& \quad (A.10.5) \quad \text{反应:R1-R3post:交换} \\
& \quad R3\text{post} \circ R1(Q) \land R3\text{post} \circ R1(S) \land \text{等待'} \lor R3\text{post} \circ R1(Q) \land R1 \circ R3\text{post}(S)
\end{align*}
\]
\[
\begin{align*}
&= \{ \text{A.1.39 (law::propositional-calculus::conditional::export::then), twice} \} \\
&\quad (R_3^\text{post} \circ R_1(Q) ; R_3^\text{post} \circ R_1(S)) \preceq \text{wait} \supset (R_3^\text{post} \circ R_1(Q) ; R_1 \circ R_3^\text{post}(S))_f \\
&= \{ \text{A.10.4 (law::reactive::R1-R3::commutative)} \} \\
&\quad (R_3^\text{post} \circ R_1(Q)_l) ; R_3^\text{post} \circ R_1(S) \preceq \text{wait} \supset (R_3^\text{post} \circ R_1(Q)_f) ; R_1 \circ R_3^\text{post}(S) \\
&= \{ \text{A.10.27 (law::reactive::R3post::wait::true), A.10.26 (law::reactive::R3post::wait::false)} \} \\
&\quad \exists ; R_3^\text{post} \circ R_1(S) \preceq \text{wait} \supset R_1(Q) ; R_1 \circ R_3^\text{post}(S) \\
&= \{ \text{A.5.19 (law::relational-calculus::sequence::left-unit)} \} \\
&\quad R_3^\text{post} \circ R_1(S) \preceq \text{wait} \supset R_1(Q) ; R_1 \circ R_3^\text{post}(S) \\
&= \{ \text{A.1.39 (law::propositional-calculus::conditional::export::then)} \} \\
&\quad (R_3^\text{post} \circ R_1(S)_l) \preceq \text{wait} \supset R_1(Q) ; R_1 \circ R_3^\text{post}(S) \\
&= \{ \text{A.10.27 (law::reactive::R3post::wait::true)} \} \\
&\quad \exists \preceq \text{wait} \supset R_1(Q) ; R_1 \circ R_3^\text{post}(S) \\
&= \{ \text{A.10.5 (definition::reactive::R3post)} \} \\
&\quad R_3^\text{post}(R_1(Q) ; R_1 \circ R_3^\text{post}(S)) \\
&\quad (***) \\
&\quad \{ \text{definition} \} \\
&\quad R_1(\neg \text{ok} \lor (1) \lor (\text{ok}' \land (3))) \\
&= \{ \text{sub-laws} \} \\
&\quad R_1 \\
&\quad \neg \text{ok} \\
&\quad \lor \neg R_3^\text{pre}(\neg ((R_1(\neg P) ; R_1(\text{true}))) \lor (R_1(Q) \land \neg \text{wait}'; R_1(\neg R))) \\
&\quad \lor (\text{ok}' \land R_3^\text{post}(R_1(Q) ; R_1 \circ R_3^\text{post}(S))) \\
&= \{ \text{A.9.1 (definition::design)} \} \\
&\quad R_1(R_3^\text{pre}(\neg ((R_1(\neg P) ; R_1(\text{true}))) \lor (R_1(Q) \land \neg \text{wait}'; R_1(\neg R))) \\
&\quad \lor R_3^\text{post}(R_1(Q) ; R_1 \circ R_3^\text{post}(S))) \\
&= \{ \text{A.10.21 (law::reactive::R3::design-split)} \} \\
&\quad \exists \neg ((R_1(\neg P) ; R_1(\text{true}))) \lor (R_1(Q) \land \neg \text{wait}'; R_1(\neg R))) \\
&\quad \neg (R_1(Q) ; R_1 \circ R_3^\text{post}(S)) \\
&= \{ \text{A.1.19 (law::propositional-calculus::negation::De-Morgan)} \} \\
&\quad R_1 \circ R_3 \neg (R_1(\neg P) ; R_1(\text{true})) \land \neg (R_1(Q) \land \neg \text{wait}'; R_1(\neg R)) \\
&\quad \neg (R_1(Q) ; R_1 \circ R_3^\text{post}(S)) \\
&= \{ \text{A.10.5 (definition::reactive::R3post)} \} \\
&\quad R_1 \circ R_3 \neg (R_1(\neg P) ; R_1(\text{true})) \land \neg (R_1(Q) \land \neg \text{wait}'; R_1(\neg R)) \\
&\quad \neg (R_1(Q) ; (\exists \preceq \text{wait} \supset R_1(S)))
\end{align*}
\]
A.11.13  \textit{law::-reactive-design::-sequence::-R1}

Law A.11.12 (\textit{law::-reactive-design::-sequence::-R1})

\[ R1(P \vdash Q) ; R1(R \vdash S) = R1(\neg (R1(\neg P) ; R1(\text{true})) \land \neg (R1(Q) ; R1(\neg R))) \]
\[ \vdash R1(Q) ; R1(S) \]

for ok, ok' not occurring in P, Q, R, S

Proof

\[ R1(P \vdash Q) ; R1(R \vdash S) = \{ \text{A.9.1 (definition::-design)} \} \]
\[ R1(\neg (ok \land P) \lor (ok' \land Q)) ; R1(\neg (ok \land R) \lor (ok' \land S)) = \{ \text{A.10.10 (law::-reactive:-R1:-disjunctivity)} \} \]
\[ R1(\neg (ok \land P)) \lor R1(\neg (ok \land R)) ; R1(ok' \land Q) \lor R1(ok' \land S) = \{ \text{A.5.16 (law::-relational-calculus::-sequence::-disjunctivity)} \} \]
\[ R1(\neg (ok \land P)) ; R1(\neg (ok \land R))(1) \]
\[ \lor R1(\neg (ok \land P)) ; R1(ok' \land Q)(2) \]
\[ \lor R1(ok' \land Q) ; R1(\neg (ok \land R))(3) \]
\[ \lor R1(ok' \land Q) ; R1(ok' \land S)(4) = \{ \text{spawn sub-laws} \} \]
\[ (*) \]
\[ (1) \]
\[ = \{ \text{sub-law} \} \]
\[ R1(\neg (ok \land P)) ; R1(\neg (ok \land R)) = \{ \text{A.1.19 (law::-propositional-calculus:-negation:-De-Morgan)} \} \]
\[ R1(\neg ok \lor \neg P) ; R1(\neg (ok \land R)) = \{ \text{A.10.10 (law::-reactive:-R1:-disjunctivity)} \} \]
\[ (R1(\neg ok) ; R1(\neg (ok \land R))) \lor (R1(\neg (ok \land R)) ; R1(\neg (ok \land R))) = \{ \text{A.10.17 (law::-reactive:-R1:-sequence:-left-zero)} \} \]
\[ R1(\neg ok) \lor (R1(\neg P) ; R1(\neg (ok \land R))) = \{ \text{A.1.19 (law::-propositional-calculus:-negation:-De-Morgan)} \} \]
\[ R1(\neg (ok \land P)) ; R1(\neg (ok \land R)) = \{ \text{A.10.10 (law::-reactive:-R1:-disjunctivity)} \} \]
\[ R1(\neg ok) \lor (R1(\neg P) ; R1(\neg ok \lor \neg R)) = \{ \text{A.10.17 (law::-reactive:-R1:-sequence:-left-zero)} \} \]
\[ R1(\neg ok) \lor (R1(\neg (ok \land R))) = \{ \text{A.5.23 (law::-relational-calculus::-sequence::-one-point::-right)} \} \]
\[ R1(\neg ok) \lor (R1(\neg P) ; R1(\text{true}) \lor R1(\neg R)) = \{ \text{A.1.12 (law::-propositional-calculus:-or:-subsumption)}: \neg R \Rightarrow \text{true} \} \]
\[ R1(\neg ok) \lor (R1(\neg P) ; R1(\text{true})) \]
(2)
\[ R1(\neg (ok \land P)) ; R1(ok' \land S) = \{ \text{ A.1.19 (law::-propositional-calculus:-negation:-De-Morgan) } \} \]
\[ R1(\neg ok \lor \neg P) ; R1(ok' \land S) = \{ \text{ A.10.10 (law::-reactive:-R1:-disjunctivity) } \} \]
\[ R1(\neg ok) \lor R1(\neg P) ; R1(ok' \land S) = \{ \text{ A.5.16 (law::-relational-calculus:-sequence:-disjunctivity) } \} \]
\[ (R1(\neg ok) ; R1(ok' \land S)) \lor (R1(\neg P) ; R1(ok' \land S)) = \{ \text{ A.10.17 (law::-reactive:-R1:-sequence:-left-zero) } \} \]
\[ R1(\neg ok) \lor (R1(\neg P) ; R1(ok' \land S)) \]
\[ \square \{ \text{ A.1.12 (law::-propositional-calculus:-or:-subsumption): ok' \land S \Rightarrow true } \}
\[ \text{ A.5.21 (law::-relational-calculus:-sequence:-monotonic-2) } \]
(1)
(3)
\[ R1(ok' \land Q) ; R1(\neg (ok \land R)) = \{ \text{ A.1.19 (law::-propositional-calculus:-negation:-De-Morgan) } \} \]
\[ R1(ok' \land Q) ; R1(\neg ok \lor \neg R) = \{ \text{ A.10.10 (law::-reactive:-R1:-disjunctivity) } \} \]
\[ R1(ok' \land Q) ; R1(\neg ok) \lor R1(\neg R) = \{ \text{ A.5.16 (law::-relational-calculus:-sequence:-disjunctivity) } \} \]
\[ (R1(ok' \land Q) ; R1(\neg ok)) \lor (R1(ok' \land Q) ; R1(\neg R)) = \{ \text{ A.10.9 (law::-reactive:-R1:-detach) } \} \]
\[ (R1(Q) \land ok' ; \neg ok \land R1(\text{true})) \lor (R1(ok' \land Q) ; R1(\neg R)) = \{ \text{ A.5.14 (law::-relational-calculus:-sequence:-condition-swing) } \} \]
\[ (R1(Q) ; ok \land \neg ok \land R1(\text{true})) \lor (R1(ok' \land Q) ; R1(\neg R)) = \{ \text{ A.1.17 (law::-propositional-calculus:-negation:-contradiction) } \} \]
\[ (R1(Q) ; \text{false}) \lor (R1(ok' \land Q) ; R1(\neg R)) = \{ \text{ A.5.26 (law::-relational-calculus:-sequence:-right-zero) } \} \]
\[ \text{false} \lor (R1(ok' \land Q) ; R1(\neg R)) = \{ \text{ A.1.13 (law::-propositional-calculus:-or:-unit) } \} \]
\[ R1(ok' \land Q) ; R1(\neg R) = \{ \text{ A.5.22 (law::-relational-calculus:-sequence:-one-point:-left) } \} \]
\[ R1(Q) ; R1(\neg R) \]
(4)
\[ = \{ \text{ sub-law } \} \]
\[ R1(ok' \land Q) ; R1(ok' \land S) \]
\( = \{ \text{A.5.22 (law::relational-calculus::sequence::one-point::left)} \} \)
\( R1(Q) ; R1(ok' \land S) \)

(*)

\( = \{ \text{collect sub-laws} \} \)
\( R1(\neg ok) \lor (R1(\neg P) ; R1(true)) \)
\( \lor (R1(Q) ; R1(\neg R)) \)
\( \lor (R1(Q) ; R1(ok' \land S)) \)

\( = \{ \text{A.10.16 (law::reactive::R1::sequence::closure), three times} \} \)
\( R1(\neg ok) \lor R1(R1(\neg P) ; R1(true)) \)
\( \lor R1(R1(Q) ; R1(\neg R)) \)
\( \lor R1(R1(Q) ; R1(ok' \land S)) \)

\( = \{ \text{A.10.10 (law::reactive::R1::disjunctivity)} \} \)
\( R1(\neg ok) \lor (R1(\neg P) ; R1(true)) \)
\( \lor (R1(Q) ; R1(\neg R)) \)
\( \lor ((R1(Q) ; R1(S)) \land ok') \)

\( = \{ \text{A.5.15 (law::relational-calculus::sequence::detach-post)} \} \)
\( R1(\neg ok) \lor (R1(\neg P) ; R1(true)) \)
\( \lor (R1(Q) ; R1(\neg R)) \)
\( \lor ((R1(Q) ; R1(S)) \land ok') \)

\( = \{ \text{A.1.23 (law::propositional-calculus::implies)} \} \)
\( R1(ok \land \neg (R1(\neg P) ; R1(true)) \land \neg (R1(Q) ; R1(\neg R)) \Rightarrow ok' \land (R1(Q) ; R1(S))) \)

\( = \{ \text{A.9.1 (definition::design)} \} \)
\( R1(\neg (R1(\neg P) ; R1(true)) \land \neg (R1(Q) ; R1(\neg R)) \vdash R1(Q) ; R1(S)) \)

\( \square \)

A.11.14 \( \text{law::reactive-design} \)

Law A.11.13 \( \text{(law::reactive-design)} \)
\[ P = R(\neg P \vdash P) \]
providing \( P \) is \( R1, R3, CSP1, \) and \( CSP2 \)

\textbf{Proof}

\[ P \]
= \{ \text{assumption: } P \text{ is } R_1, R_3, \text{CSP2} \}

R_1 \circ R_3 \circ \text{CSP1} \circ \text{CSP2}(P)

= \{ \text{A.12.2 (definition::-CSP::-CSP2)} \}

R_1 \circ R_3(P ; J)

= \{ \text{A.10.3 (law::-reactive::J::splitting)} \}

R_1 \circ R_3(P^f \lor (P^t \land ok'))

= \{ \text{A.10.23 (law::-reactive::R3::not-wait)} \}

R_1 \circ R_3((P^f \lor (P^t \land ok'))^f)

= \{ \text{A.7.6 (law::-substitution)} \}

R_1 \circ R_3(P^f \lor (P^t \land ok'))

= \{ \text{assumption: } P \text{ is } \text{CSP1} \}

R_1 \circ R_3((\text{CSP1}(P))^f \lor (P^t \land ok'))

= \{ \text{A.12.1 (definition::-CSP::-CSP1)} \}

R_1 \circ R_3((R_1(\neg \text{ok}) \lor P^f)^f \lor (P^t \land ok'))

= \{ \text{A.7.6 (law::-substitution)} \}

R_1 \circ R_3((R_1(\neg \text{ok}))^f \lor P^f \lor (P^t \land ok'))

= \{ \text{A.7.6 (law::-substitution)} \}

R_1 \circ R_3(R_1(\neg \text{ok}) \lor P^f \lor (P^t \land ok'))

= \{ \text{A.10.10 (law::-reactive::R1::disjunctivity)} \}

R_1 \circ R_3(\neg \text{ok} \lor P^f \lor (P^t \land ok'))

= \{ \text{A.9.1 (definition::-design)} \}

R_1 \circ R_3(\neg P^f \vdash P^f)

\square
A.12  CSP

A.12.1  definition:::-CSP:-CSP1

Definition A.12.1 (definition:::-CSP:-CSP1)

\[ \text{CSP} \text{-CSP2}(P) \triangleq R1(\neg \text{ok}) \lor P \]

A.12.2  definition:::-CSP:-CSP2

Definition A.12.2 (definition:::-CSP:-CSP2)

\[ \text{CSP2}(P) \triangleq P ; J \]

A.12.3  definition:::-CSP:-M-CSP-2

Definition A.12.3 (definition:::-CSP:-M-CSP-2)

\[ M_{\text{CSP2}} \triangleq \text{var}0.\text{ok},1.\text{ok} ; M_{\text{CSP-}\neg \text{ok}} \]

A.12.4  definition:::-CSP:-CSP3

Definition A.12.4 (definition:::-CSP:-CSP3)

\[ \neg \text{wait} \Rightarrow (P = \exists \text{ref} \bullet P) \]

A.12.5  definition:::-CSP:-M-CSP

Definition A.12.5 (definition:::-CSP:-M-CSP)

\[ M_{\text{CSP}} \triangleq N_{\text{CSP}} ; \text{Skip} \]

A.12.6  definition:::-CSP:-M-ok

Definition A.12.6 (definition:::-CSP:-M-ok)

\[ M_0.k \triangleq M^4[\text{true, true/0.ok, 1.ok}] \]
A.12.7  **definition:::-CSP:-N-CSP**

Definition A.12.7 (**definition:::-CSP:-N-CSP**)

\[ N_{\text{CSP}} \triangleq \]
\[ (\text{ok}^t = (0.\text{ok} \land 1.\text{ok})) \]
\[ \land (\text{wait}^t = (0.\text{wait} \lor 1.\text{wait})) \]
\[ \land (\text{ref}^t = 0.\text{ref} \cup 1.\text{ref}) \]
\[ \land \exists u \ldots \]
\[ \land (v^t = \ldots) \]

A.12.8  **law:::-CSP:-CSP2:-conjunctive**

Law A.12.1 (**law:::-CSP:-CSP2:-conjunctive**) Suppose that \( P = \text{CSP2}(P) \) and \( Q = \text{CSP2}(Q) \), then

\[ P \land Q = \text{CSP2}(P \land Q) \]

**Proof**

\[ \text{CSP2}(P \land Q) \]
\[ = \{ \text{assumption} \} \]
\[ \text{CSP2}(\text{CSP2}(P) \land \text{CSP2}(Q)) \]
\[ = \{ \text{A.10.3 (law:::-reactive:-J:-splitting)} \} \]
\[ \text{CSP2}((P^f \lor (P^t \land \text{ok}^t)) \land (Q^f \lor (Q^t \land \text{ok}^t))) \]
\[ = \{ \text{A.10.3 (law:::-reactive:-J:-splitting)} \} \]
\[ ((P^f \lor (P^t \land \text{ok}^t)) \land (Q^f \lor (Q^t \land \text{ok}^t)))^f \]
\[ \lor (((P^f \lor (P^t \land \text{ok}^t)) \land (Q^f \lor (Q^t \land \text{ok}^t)))^t \land \text{ok}^t) \]
\[ = \{ \text{A.7.6 (law:::-substitution)} \} \]
\[ ((P^f \lor (P^t \land \text{false})) \land (Q^f \lor (Q^t \land \text{false}))) \]
\[ \lor (((P^f \lor (P^t \land \text{true})) \land (Q^f \lor (Q^t \land \text{true}))) \land \text{ok}^t) \]
\[ = \{ \text{A.1.7 (law:::-propositional-calculus:-and:-zero)} \} \]
\[ \text{A.1.13 (law:::-propositional-calculus:-or:-unit)} \]
\[ (P^f \land Q^f) \lor (((P^f \lor P^t) \land (Q^f \lor Q^t)) \land \text{ok}^t) \]
\[ = \{ \text{A.1.5 (law:::-propositional-calculus:-and:-or-distributivity)} \} \]
\[ (P^f \land Q^f) \]
\[ \lor (P^f \land Q^f \land \text{ok}^t) \]
\[ \lor (P^f \land Q^t \land \text{ok}^t) \]
\[ \lor (P^t \land Q^f \land \text{ok}^t) \]
\[ \lor (P^t \land Q^t \land \text{ok}^t) \]
\[ = \{ \text{A.1.8 (law:::-propositional-calculus:-or:-absorption)} \} \]
\[ (P^f \land Q^f) \]
\[\forall (P \land Q \land ok') \\land (P \land Q') \land (P \land Q')\]

\[= \{ \text{A.1.5 (law::-propositional-calculus:-and:-or-distributivity)} \} \]

\[(P \lor (P \land ok')) \land (Q \lor (Q' \land ok'))\]

\[= \{ \text{A.10.3 (law::-reactive:-J:-splitting)} \} \]

\[\text{CSP2}(P) \land \text{CSP2}(Q)\]

\[= \{ \text{assumption} \} \]

\[P \land Q\]

\[\square\]

A.12.9  \textit{law::-CSP:-CSP3:-consequence}

Law A.12.2 (\textit{law::-CSP:-CSP3:-consequence}) Suppose that \(R_1 \circ R_3(P \vdash Q)\) is CSP3,

\[\neg \text{wait} \Rightarrow (R_1 \circ R_3(P \vdash Q) = \neg \text{wait} \land R_1 \circ R_3(\forall \text{ref} \vdash P \vdash \exists \text{ref} \bullet Q))\]

\textbf{Proof}  assume \(\neg \text{wait}\)

\[R_1 \circ R_3(P \vdash Q)\]

\[= \{ \text{A.12.4 (definition::-CSP:-CSP3)} \} \]

\[\exists \text{ref} \bullet R_1 \circ R_3(P \vdash Q)\]

\[= \{ \text{A.10.11 (law::-reactive:-R1:-exists), ref not free in R1} \} \]

\[R_1(\exists \text{ref} \bullet R_3(P \vdash Q))\]

\[= \{ \text{A.10.22 (law::-reactive:-R3:-not-wait:-substitution) (assumption: \neg \text{wait})} \} \]

\[R_1(\exists \text{ref} \bullet (P \vdash Q))\]

\[= \{ \text{A.9.1 (definition::-design)} \} \]

\[R_1(\exists \text{ref} \bullet \neg \text{ok} \lor \lnot P \lor (ok' \land Q))\]

\[= \{ \text{A.2.7 (law::-predicate-calculus:-exists:-or-distributivity)} \} \]

\[R_1((\exists \text{ref} \bullet \neg \text{ok}) \lor (\exists \text{ref} \bullet \lnot P) \lor (\exists \text{ref} \bullet ok' \land Q))\]

\[= \{ \text{A.2.1 (law::-predicate-calculus:-exists:-and:-non-free), ref not free in \neg \text{ok}} \} \]

\[R_1(\neg \text{ok} \lor (\exists \text{ref} \bullet \lnot P) \lor (\exists \text{ref} \bullet ok' \land Q))\]

\[= \{ \text{A.2.2 (law::-predicate-calculus:-exists:-De-Morgan)} \} \]

\[R_1(\neg \text{ok} \lor (\forall \text{ref} \bullet P) \lor (\exists \text{ref} \bullet ok' \land Q))\]

\[= \{ \text{A.2.1 (law::-predicate-calculus:-exists:-and:-non-free), ref not free in ok'} \} \]

\[R_1(\neg \text{ok} \lor (\forall \text{ref} \bullet P) \lor (ok' \land \exists \text{ref} \bullet Q))\]

\[= \{ \text{A.9.1 (definition::-design)} \} \]

\[R_1(\forall \text{ref} \bullet P \vdash \exists \text{ref} \bullet Q)\]

\[= \{ \text{A.10.22 (law::-reactive:-R3:-not-wait:-substitution) (assumption: \neg \text{wait})} \} \]
A.12.10 *law*:\:-CSP:\:-CSP2:\:-J-splitting

Law A.12.3 (*law*:\:-CSP:\:-CSP2:\:-J-splitting)

\[
\text{CSP2}(P) = P^f \lor (P^t \land \text{ok}')
\]

Proof

\[
\begin{align*}
\text{CSP2}(P) &= \{ \text{A.12.2 (definition::-CSP::-CSP2)} \} \\
&= P^f ; J \\
&= \{ \text{A.10.3 (law::reactive::J::splitting)} \} \\
&= P^f \lor (P^t \land \text{ok}')
\end{align*}
\]

A.12.11 *law*:\:-CSP:\:-CSP2:\:-wait-ok-monotonic

Law A.12.4 (*law*:\:-CSP:\:-CSP2:\:-wait-ok-monotonic)

\[
(P = \text{CSP2}(P)) \implies [P^f \implies P^f_t]
\]

Proof

\[
\begin{align*}
P &= \text{CSP2}(P) \\
&= \{ \text{A.12.3 (law::-CSP::-CSP2::-J-splitting)} \} \\
&= P^f \lor (P^t \land \text{ok}') \\
&= \{ \text{A.2.11 (law::predicate-calculus::forall::boolean)} \} \\
&= (P = P^f \lor (P^t \land \text{ok}'))^t \land (P = P^f \lor (P^t \land \text{ok}'))^t \\
&= \{ \text{A.1.3 (law::propositional-calculus::and::elimination)} \} \\
&= P^f \lor (P^t \land \text{true}) \\
&= \{ \text{A.1.6 (law::propositional-calculus::and::unit)} \} \\
&= P^f \lor P^t \\
&= \{ \text{A.2.12 (law::predicate-calculus::forall::universal)} \} \\
&\implies [P^t \implies P^f] \land [P^f \lor P^t \implies P^t]
\end{align*}
\]
\[ \Rightarrow \{ \text{A.1.3 (law::-propositional-calculus:-and:-elimination)} \} \\
[P^f \lor P^t \Rightarrow P^t] \\
= \{ \text{A.1.9 (law::-propositional-calculus:-or:-elimination)} \} \\
[P^f \Rightarrow P^t] \land [P^t \Rightarrow P^t] \\
= \{ \text{A.1.3 (law::-propositional-calculus:-and:-elimination)} \} \\
[P^f \Rightarrow P^t] \\
= \{ \text{A.2.11 (law::-predicate-calculus:-forall:-boolean)} \} \\
[(P^f \Rightarrow P^t)^f] \land [(P^f \Rightarrow P^t)^t] \\
\Rightarrow \{ \text{A.1.3 (law::-propositional-calculus:-and:-elimination)} \} \\
[(P^f \Rightarrow P^t)^f] \\
\Rightarrow \{ \text{A.7.6 (law::-substitution)} \} \\
[P^f \Rightarrow P^f] \]

\[\square\]

**A.12.12**  
**law::-CSP:-M-CSP:-ok’-substitution**

**Law A.12.5 (law::-CSP:-M-CSP:-ok’-substitution)**

\[ M_{CSP}^f = \neg (0.ok \land 1.ok) \land M_{CSP}^f \]

**Proof**

\[ M_{CSP}^f \]

\[ = \{ \text{A.12.5 (definition::-CSP::M-CSP)} \} \]

\[ (N_{CSP} \ ; \ Skip)^f \]

\[ = \{ \text{A.7.6 (law::-substitution)} \} \]

\[ N_{CSP} \ ; \ Skip^f \]

\[ = \{ \text{A.13.13 (law::-CML::Skip::ok’-substitution)} \} \]

\[ N_{CSP} \ ; \ \neg ok \land Skip^f \]

\[ = \{ \text{A.5.14 (law::-relational-calculus::sequence::condition-swing)} \} \]

\[ N_{CSP} \land \neg ok' \ ; \ Skip^f \]

\[ = \{ \text{A.12.6 (law::-CSP::N-CSP::ok’-substitution)} \} \]

\[ \neg (0.ok \land 1.ok) \land N_{CSP} \land \neg ok' \ ; \ Skip^f \]

\[ = \{ \text{A.5.14 (law::-relational-calculus::sequence::condition-swing)} \} \]

\[ \neg (0.ok \land 1.ok) \land N_{CSP} \ ; \ \neg ok \land Skip^f \]

\[ = \{ \text{A.13.13 (law::-CML::Skip::ok’-substitution)} \} \]

\[ \neg (0.ok \land 1.ok) \land N_{CSP} \ ; \ Skip^f \]

\[ = \{ \text{A.5.25 (law::-relational-calculus::sequence::pre-separation)} \} \]

\[ \neg (0.ok \land 1.ok) \land (N_{CSP} \ ; \ Skip^f) \]
= \{ \textbf{A.7.6 (law:::-substitution)} \}
\neg (0.ok \land 1.ok) \land (N\_CSP \; ; \; \text{Skip})^f
= \{ \textbf{A.12.5 (definition:::-CSP:::-M-CSP)} \}
\neg (0.ok \land 1.ok) \land M^f_{\_CSP}

\Box

\textbf{A.12.13 \textit{law:::-CSP:::-N-CSP:::-ok'}-substitution}

\textbf{Law A.12.6 \textit{(law:::-CSP:::-N-CSP:::-ok'}-substitution)}

\[ N^f_{\_CSP} \Rightarrow \neg (0.ok \land 1.ok) \]

\textbf{Proof}

\[ N^f_{\_CSP} \]
\[ = \{ \textbf{A.12.7 (definition:::-CSP:::-N-CSP)} \}
(\text{ok'} = (0.ok \land 1.ok))
\land (\text{wait'} = (0.wait \lor 1.wait))
\land (\text{ref'} = 0.ref \cup 1.ref)
\land (\exists u \bullet
(u \uparrow \alpha(P) = (0.tr - tr))
\land (u \uparrow \alpha(Q) = (1.tr - tr))
\land (u \uparrow \alpha(P \mid Q) = u)
\land (tr' = tr \cap u)
)
\land (v' = \text{varMerge}(0.v, 1.v))
\}^f
\]
\[ = \{ \textbf{A.7.6 (law:::-substitution)} \}
(ok' = (0.ok \land 1.ok))^f
\land ((\text{wait'} = (0.wait \lor 1.wait))
\land (\text{ref'} = 0.ref \cup 1.ref)
\land (\exists u \bullet
(u \uparrow \alpha(P) = (0.tr - tr))
\land (u \uparrow \alpha(Q) = (1.tr - tr))
\land (u \uparrow \alpha(P \mid Q) = u)
\land (tr' = tr \cap u)
)
\land (v' = \text{varMerge}(0.v, 1.v))^f\]
\[
\Rightarrow \{ \text{A.1.3 (law::propositional-calculus::and::elimination)} \}
\]

\[
(\text{ok}' = (0.\text{ok} \land 1.\text{ok})')
\]

\[
= \{ equals : -substitution \}
\]

\[
(\text{false} = (0.\text{ok} \land 1.\text{ok}))
\]

\[
= \{ \text{propositional calculus : contradiction : rewrite} \}
\]

\[
\neg (0.\text{ok} \land 1.\text{ok})
\]
A.13  **CML**

A.13.1  **let**

A.13.1.1  **definition::-CML::-let**

Definition A.13.1 (**definition::-CML::-let**)  

\[ \text{let } x \bullet P \overset{\text{def}}{=} P \]

A.13.2  **loc**

A.13.2.1  **definition::-CML::-loc**

Definition A.13.2 (**definition::-CML::-loc**)  

\[ \text{loc } s \bullet P \overset{\text{def}}{=} \text{lift}(s) ; P \]

A.13.2.2  **law::-CML::-loc:: monotonic**

Law A.13.1 (**law::-CML::-loc:: monotonic**)  

\[ P \sqsubseteq Q \Rightarrow \text{loc } s \bullet P \sqsubseteq \text{loc } s \bullet Q \]

Proof

\[ \begin{align*}
\text{loc } s \bullet P \\
= \{ \text{A.13.2 (**definition::-CML::-loc**)} \}
\text{lift}(s) ; P \\
\sqsubseteq \{ \text{assumption: } P \sqsubseteq Q, \text{A.5.21 (**law::-relational-calculus::sequence::monotonic-2**)} \}
\text{lift}(s) ; Q \\
= \{ \text{A.13.2 (**definition::-CML::-loc**)} \}
\text{loc } s \bullet Q
\end{align*} \]

□

A.13.3  **merge**

A.13.3.1  **definition::-CML::-merge::separating-sim::Ui**

Definition A.13.3 (**definition::-CML::-merge::separating-sim::Ui**)  

\[ \text{Ui}(m) \overset{\text{def}}{=} \text{var } i.m := m ; \text{end } m \]
A.13.3.2 \( \text{definition::-CML:-merge:-separating-sim:-V0(m,n)} \)

Definition A.13.4 (\( \text{definition::-CML:-merge:-separating-sim:-V0(m,n)} \))

\[
V_0(m, n) \triangleq \text{var} 0.m ; 0.n := n ; \text{end} m
\]

A.13.3.3 \( \text{law::-CML:-merge:-M-ok'-true} \)

Law A.13.2 (\( \text{law::-CML:-merge:-M-ok'-true} \))

\[
M^t = \{ \text{A.7.6 (law::-substitution), A.12.5 (definition::-CSP::M-CSP)} \} \\
0.ok' \land 1.ok \land M^t
\]

A.13.3.4 \( \text{law::-CML:-merge-separating-sim:-U0:-0.ok'-1} \)

Law A.13.3 (\( \text{law::-CML:-merge-separating-sim:-U0:-0.ok'-1} \))

\[
(P ; U_0(m))[b/0.ok'] = P^b ; V_0(m, n)
\]

A.13.3.5 \( \text{law::-CML:-merge-separating-sim:-U0:-0.ok'-2} \)

Law A.13.4 (\( \text{law::-CML:-merge-separating-sim:-U0:-0.ok'-2} \))

\[
P ; U_0(m)[b/0.ok'] = P^b ; V_0(m, n)
\]

Proof

\[
P ; U_0(m)[b/0.ok'] = \{ \text{A.13.4 (definition::-CML:-merge:-separating-sim:-V0(m,n))} \} \\
P ; (ok = b) \land V_0(m, n) = \{ \text{A.5.23 (law::-relational-calculus:-sequence:-one-point:-right)} \} \\
P^b ; V_0(m, n)[b/ok] = \{ \text{A.13.8 (law::-CML:-merge-separating-sim:-V0(m,n):-ok-substitution)} \} \\
P^b ; V_0(m, n)
\]

\[\square\]

A.13.3.6 \( \text{law::-CML:-merge-separating-sim:-U0:-ok'-1} \)

Law A.13.5 (\( \text{law::-CML:-merge-separating-sim:-U0:-ok'-1} \))

\[
(P ; U_0(m))[b/0.ok'] = P^b ; V_0(m, n)
\]

Proof
\[(P ; U0(m))[b/0.ok']\]
\[= \{ A.7.6 (law::substitution) \}\]
\[P ; U0(m)[b/0.ok']\]
\[= \{ A.13.4 (law::CML::merge::separating-sim::U0::0.ok'-2) \}\]
\[P^b ; V0(m, n)\]

\[\square\]

A.13.3.7 \textit{law::CML::merge::separating-sim::Ui::i.ok'-substitution}

\textbf{Law} A.13.6 \textit{(law::CML::merge::separating-sim::Ui::i.ok'-substitution)}

\[Ui(m)[b/i.ok'] = (b = ok) \land (\text{end ok} ; Ui(n) ; \text{var ok}) \quad \text{for} \quad m = (ok, n)\]

\textbf{Proof}

\[Ui(m)[b/i.ok']\]
\[= \{ A.13.3 (definition::CML::merge::separating-sim::Ui) \}\]
\[(\text{var} i.m := \{m, m', i.m, i.m'\} \quad m ; \text{end m})[b/i.ok']\]
\[= \{ A.5.2 (definition::relational-calculus::assignment::declaration-init) \}\]
\[(\text{var} i.m ; i.m := \{m, m', i.m, i.m'\} \quad m ; \text{end m})[b/i.ok']\]
\[= \{ A.7.5 (law::relational-calculus::var::substitution) \}\]
\[(\text{var} i.m ; ((i.m := \{m, m', i.m, i.m'\} \quad m ; \text{end m})[b/i.ok'])\]
\[= \{ A.7.2 (law::relational-calculus::end::substitution) \}\]
\[(\text{var} i.m ; i.m := \{m, m', i.m, i.m'\} \quad m)[b/i.ok'] ; \text{end m}\]
\[= \{ \text{assumption:} (m = (ok, n)) \}\]
\[\text{var i.m} ;\]
\[(i.ok, i.n := \{ok, n, ok', n', i.ok, i.n, i.ok', i.n'\} \quad ok, n)[b/i.ok'] ; \text{end m}\]
\[= \{ A.7.4 (law::relational-calculus::assignment::substitution) \}\]
\[\text{var i.m} ;\]
\[(b = ok) \land (\text{end i.ok} ; i.n := \{ok, n, ok', n', i.ok, i.n, i.n'\} \quad n ; \text{var i.ok}) ; \text{end m}\]
\[= \{ A.5.30 (law::relational-calculus::var::pre-separation) \}\]
\[(b = ok) \land (\text{var i.m} ; \text{end i.ok} ; i.n := \{ok, n, ok', n', i.n, i.n'\} \quad n ; \text{var i.ok} ;\]

116
\textbf{end}\ m) \quad = \{ \textbf{assumption:} (m = (ok, n)) \} \\
(b = ok) \\
\land (\textbf{var} i.ok, i.n ; \textbf{end} i.ok ; \\
i.n := (ok, ok', i', i.n, i.n') n ; \\
\textbf{var} i.ok ; \textbf{end} m) \quad = \{ \text{A.5.34 (law::relational-calculus::var::split)} \} \\
(b = ok) \\
\land (\textbf{var} i.n ; \textbf{var} i.ok ; \textbf{end} i.ok ; \\
i.n := (ok, ok', i', i.n, i.n') n ; \\
\textbf{var} i.ok ; \textbf{end} m) \quad = \{ \text{A.5.29 (law::relational-calculus::var::identity)} \} \\
(b = ok) \\
\land (\textbf{var} i.n ; \textbf{II}\{m,i.n\} ; \\
i.n := (ok, ok', i', i.n, i.n') n ; \\
\textbf{var} i.ok ; \textbf{end} m) \quad = \{ \text{A.5.17 (law::relational-calculus::sequence::identity)} \} \\
(b = ok) \\
\land (\textbf{var} i.n ; \\
i.n := (ok, ok', i', i.n, i.n') n ; \\
\textbf{var} i.ok ; \textbf{end} m) \quad = \{ \textbf{assumption:} (m = (ok, n)) \} \\
(b = ok) \\
\land (\textbf{var} i.n ; \\
i.n := (ok, ok', i', i.n, i.n') n ; \\
\textbf{var} i.ok ; \textbf{end} ok, n) \quad = \{ \text{A.5.4 (definition::relational-calculus::end::split)} \} \\
(b = ok) \\
\land (\textbf{var} i.n ; \\
i.n := (ok, ok', i', i.n, i.n') n ; \}

117
\begin{verbatim}
var i.ok ;
end ok ;
end n)
= { A.5.10 (law::relational-calculus::end::var)  }
(b = ok)
\land ( var i.n ;
i.n :=\{ok,ok',n',i.n,i.n'\} n ;
end ok ;
var i.ok ;
end n)
= { A.5.8 (law::relational-calculus::end::assignment)  }
(b = ok)
\land ( var i.n ;
end ok ;
i.n :=\{n',i.n,i.n'\} n ;
var i.ok ;
end n)
= { A.5.10 (law::relational-calculus::end::var)  }
(b = ok)
\land ( end ok ;
var i.n ;
i.n :=\{n',i.n,i.n'\} n ;
var i.ok ;
end n)
= { A.5.10 (law::relational-calculus::end::var)  }
(b = ok)
\land ( end ok ;
var i.n ;
i.n :=\{n',i.n,i.n'\} n ;
end n ;
var i.ok)
= { A.13.3 (definition::CML::merge::separating-sim::Ui)  }
(b = ok)
\land ( end ok ;
Ui(n) ;
var i.ok)
\end{verbatim}
A.13.3.8 \(\text{law::CML::merge::separating-sim::Ui::i.ok}'\)

Law A.13.7 \(\text{law::CML::merge::separating-sim::Ui::i.ok}'\)

\[(P_f \ ; \ Ui(m))[true/i.ok'] = (P_f' \ ; \ Vi(m, n)\]

A.13.3.9 \(\text{law::CML::merge::separating-sim::V0(m,n)::ok-substitution}\)

Law A.13.8 \(\text{law::CML::merge::separating-sim::V0(m,n)::ok-substitution}\)

\[V0(m, n)[b/ok] = V0(m, n)\]

Proof

\[V0(m, n)[b/ok]\]
\[= \{ A.13.4 \ (\text{definition::CML::merge::separating-sim::V0(m,n)}) \} \]
\[(\text{var} 0.m ; 0.n := n ; \text{end} m)[b/ok]\]
\[= \{ A.5.10 \ (\text{definition::relational-calculus::var}), \text{twice} \} \]
\[(\exists 0.m, m' \bullet 0.n := n)[b/ok]\]
\[= \{ A.7.6 \ (\text{law::substitution}) \} \]
\[\exists 0.m, m' \bullet (0.n := n)[b/ok]\]
\[= \{ A.7.4 \ (\text{law::relational-calculus::assignment::substitution}) \} \]
\[\exists 0.m, m' \bullet 0.n := n[b/ok]\]
\[= \{ A.7.6 \ (\text{law::substitution}) \} \]
\[\exists 0.m, m' \bullet 0.n := n\]
\[= \{ A.5.10 \ (\text{definition::relational-calculus::var}), A.5.10 \ (\text{definition::relational-calculus::var}) \} \]
\[(\text{var} 0.m ; 0.n := n ; \text{end} m)\]
\[= \{ A.13.4 \ (\text{definition::CML::merge::separating-sim::V0(m,n)}) \} \]
\[V0(m, n)\]

\[\square\]

A.13.4 \(\text{Skip}\)

A.13.4.1 \(\text{definition::CML::Skip}\)

Definition A.13.5 \(\text{definition::CML::Skip}\)

\[\text{Skip} \triangleq R3(\exists \text{ref} \bullet \Pi_{\{tr,ref,ok,wait,v\}})\]
A.13.4.2  **law::-CML:-Skip:-lifted-assignment**

Law A.13.9  (**law::-CML:-Skip:-lifted-assignment**)  

\[
\text{Skip} = \text{lift}(v := v) \\
\text{Skip} = R(\text{true} \vdash (v' = v)_{tr} \land \neg \text{wait}') \\
\text{lift}(s) = R1 \circ R3(\text{true} \vdash s_{tr, ref} \land \neg \text{wait}')
\]

**Proof**

\[
\text{Skip} = \{ \text{A.13.15 (law::-CML:-Skip:-reactive-design)} \} \\
R1 \circ R3(\text{true} \vdash (v' = v)_{tr} \land \neg \text{wait}') \\
= \{ \text{A.13.7 (definition::-CML:-assignment)}, (\alpha((v' = v)_{tr}) = \{v, v', tr', ref', ref'\} \} \} \\
R1 \circ R3(\text{true} \vdash (v := v)_{tr, ref} \land \neg \text{wait}') \\
= \{ \text{3.2.5 (definition::-lift)} \} \\
\text{lift}(v := v)
\]

\[\square\]

A.13.4.3  **law::-CML:-Skip:-not-wait:-diverging**

Law A.13.10  (**law::-CML:-Skip:-not-wait:-diverging**)  

\[
\text{Skip}_f = (tr' = tr) \land \neg ok \land \neg \text{wait}' \land (v' = v)
\]

**Proof**

\[
\text{Skip}_f \\
= \{ \text{A.13.12 (law::-CML:-Skip:-not-wait)} \} \\
(tr' = tr) \land (ok = \text{false}) \land \neg \text{wait}' \land (v' = v) \\
= \{ \text{A.1.32 (law::-propositional-calculus:-equivalence:-boolean)} \} \\
(tr' = tr) \land \neg ok \land \neg \text{wait}' \land (v' = v)
\]

\[\square\]

A.13.4.4  **law::-CML:-Skip:-not-wait:-not-diverging**

Law A.13.11  (**law::-CML:-Skip:-not-wait:-not-diverging**)  

\[
\text{Skip}_f = (tr' = tr) \land ok \land \neg \text{wait}' \land (v' = v)
\]

**Proof**

\[
\text{Skip}_f = \{ \text{A.13.12 (law::-CML:-Skip:-not-wait)} \}
\]

120
\[(tr' = tr) \land (ok = true) \land \neg wait' \land (v' = v)\]
\[= \{ \text{A.1.32 (law::propositional-calculus::equivalence:boolean)} \}\]
\[(tr' = tr) \land ok \land \neg wait' \land (v' = v)\]

\[\square\]

A.13.4.5  \textit{law::CML::Skip::not-wait}\]

\textbf{Law A.13.12 (law::CML::Skip::not-wait)}

\[\text{Skip}^b_f = \Pi_{\{tr,v\}} (ok = b) \land \neg wait'\]

\textbf{Proof}

\[\text{Skip}^b_f\]
\[= \{ \text{A.13.5 (definition::CML::Skip)} \}\]
\[(\exists \text{ ref} \bullet \]
\[\Pi_{\{tr,ref,ok,wait,v\}} \}
\[)\]
\[)\]
\[^b_f\]
\[= \{ \text{A.10.22 (law::reactive::R3::not-wait::substitution)} \}\]
\[(\exists \text{ ref} \bullet \]
\[\Pi_{\{tr,ref,ok,wait,v\}} \]
\[)\]
\[^b_f\]
\[= \{ \text{A.7.6 (law::substitution)} \}\]
\[(\exists \text{ ref} \bullet \]
\[\Pi_{\{tr,ref,ok,wait,v\}} \]
\[)\]
\[^b_f\]
\[= \{ \text{A.5.11 (law::relational-calculus::II::unwinding)} \}\]
\[(\exists \text{ ref} \bullet \]
\[\Pi_{\{tr,ref,ok,wait,v\}} \]
\[)\]
\[^b_f\]
\[= \{ \text{A.7.6 (law::substitution)} \}\]
\[(\exists \text{ ref} \bullet \]
\[\Pi_{\{tr,ref,ok,wait,v\}} \]
\[)\]
\[^b_f\]
\[= \{ \text{A.1.32 (law::propositional-calculus::equivalence:boolean)} \}\]
\[\exists \text{ref} \cdot \]
\[(\Pi_{\{tr, \text{ref}, \text{ok}, v\}} \land \neg \text{wait}')^b = \{ \ A.7.6 \ (\text{law}::\text{-substitution}) \ \} \]
\[\exists \text{ref} \cdot \]
\[(\Pi_{\{tr, \text{ref}, v\}}^b \land (\text{ok}' = \text{ok})^b \land \neg \text{wait}'^b = \{ \ A.7.6 \ (\text{law}::\text{-substitution}) \ \} \]
\[\exists \text{ref} \cdot \]
\[(\Pi_{\{tr, v\}} \land (\text{ok} = b) \land \neg \text{wait}' = \{ \ A.5.11 \ (\text{law}::\text{-relational-calculus}::\text{-II}::\text{-unwinding}) \ \} \]
\[\exists \text{ref} \cdot \]
\[(\Pi_{\{tr, v\}} \land (\text{ref}' = \text{ref}) \land (\text{ok} = b) \land \neg \text{wait}' = \{ \ A.2.6 \ (\text{law}::\text{-predicate-calculus}::\text{-exists}::\text{-one-point}) \ \} \]
\[(\Pi_{\{tr, v\}}^b \land (\text{ok} = b) \land \neg \text{wait}')[\text{ref}'/\text{ref}] = \{ \ A.7.6 \ (\text{law}::\text{-substitution}) \ \} \]
\[\Pi_{\{tr, v\}}^b[\text{ref}'/\text{ref}] \land (\text{ok} = b)[\text{ref}'/\text{ref}] \land \neg \text{wait}'[\text{ref}'/\text{ref}] = \{ \ A.7.6 \ (\text{law}::\text{-substitution}) \ \} \]
\[\Pi_{\{tr, v\}} \land (\text{ok} = b) \land \neg \text{wait}' = \{ \ A.5.6 \ (\text{definition}::\text{-relational-calculus}::\text{-II}) \ \} \]
\[(\text{tr}' = \text{tr}) \land (\text{ok} = b) \land \neg \text{wait}' \land (v' = v)\]

\[\square\]

**A.13.4.6** \(\text{law}::\text{-CML}::\text{-Skip}::\text{-ok'}-\text{-substitution}\)

**Law** A.13.13 (\(\text{law}::\text{-CML}::\text{-Skip}::\text{-ok'}-\text{-substitution}\))

\(\text{Skip}' \Rightarrow \text{ok}\)
Proof

1. \( \text{Skip} = R(\text{ok}) \)

\( \text{Skip} \)

\[ = \{ \text{A.13.5 (definition::CML::Skip)} \} \]

\( R(\text{true} \rightarrow (tr' = tr) \land \text{wait'}) \)

\[ = \{ \text{A.10.7 (definition::reactive::R)} \} \]

\( R1 \circ R3(\text{true} \rightarrow (tr' = tr) \land \text{wait'}) \)

\[ = \{ \text{A.10.15 (law::reactive::R1::ok'-substitution)} \} \]

\( R1(R3(\text{true} \rightarrow (tr' = tr) \land \text{wait'})) \)

\[ = \{ \text{A.10.24 (law::reactive::R3::ok'-substitution)} \} \]

\( R1(\text{false} \land \text{true} \rightarrow \text{true} \rightarrow (tr' = tr) \land \text{wait'}) \)

\[ = \{ \text{A.9.3 (law::design::ok'-substitution::false)} \} \]

\( R1(\text{false} \land \text{true} \rightarrow \text{true} \rightarrow (tr' = tr) \land \text{wait'}) \)

\[ = \{ \text{A.1.6 (law::propositional-calculus::and::unit)} \} \]

\( R1(\text{false} \land \text{true} \rightarrow \text{true} \rightarrow (tr' = tr) \land \text{wait'}) \)

\[ = \{ \text{A.10.1 (definition::reactive::R1)} \} \]

\( R1(R1(\text{false} \land \text{true} \rightarrow \text{true} \rightarrow (tr' = tr) \land \text{wait'})) \)

\[ = \{ \text{A.10.6 (law::reactive::R1::conditional)} \} \]

\( R1(R1(\text{false} \land \text{true} \rightarrow \text{true} \rightarrow (tr' = tr) \land \text{wait'})) \)

\[ = \{ \text{A.10.13 (law::reactive::R1::idempotence)} \} \]

\( R1(\text{false} \land \text{true} \rightarrow \text{true} \rightarrow (tr' = tr) \land \text{wait'}) \)

\[ = \{ \text{A.1.41 (law::propositional-calculus::conditional::idempotence)} \} \]

\( R1(\text{false} \land \text{true} \rightarrow \text{true} \rightarrow (tr' = tr) \land \text{wait'}) \)

2. \( \text{Skip} \Rightarrow \neg \text{ok} \)

\( \text{Skip} \)

\[ = \{ \text{A.1.4 (law::propositional-calculus::and::idempotence)} \} \]

\( \text{Skip} \land \text{Skip} \)

\[ = \{ \text{part 1} \} \]

\( \text{Skip} \land R1(\neg \text{ok}) \)

\[ = \{ \text{A.13.14 (law::CML::Skip::R1)} \} \]

\( (R1(\text{Skip}) \land R1(\neg \text{ok}) \)

\[ = \{ \text{A.10.15 (law::reactive::R1::ok'-substitution)} \} \]

\( R1(\text{Skip}) \land R1(\neg \text{ok}) \)

\[ = \{ \text{A.10.7 (law::reactive::R1::conjunctive-cancellation)} \} \]

\( R1(\text{Skip}) \land \neg \text{ok} \)

\[ \Rightarrow \{ \text{A.1.3 (law::propositional-calculus::and::elimination)} \} \]

ok
A.13.4.7  law::CML::Skip::R1

Law A.13.14 (law::CML::Skip::R1)

\[ \text{Skip} = R1(\text{Skip}) \]

Proof

\[ \text{Skip} \]
\[ = \{ \text{A.13.5 (definition::CML::Skip)} \} \]
\[ R3(\exists \text{ref} \cdot \Pi_{\{\text{tr,ref,ok,wait,v}\}}) \]
\[ = \{ \text{A.10.4 (definition::reactive::R3)} \} \]
\[ \Pi_R \triangleleft \text{wait} \triangleright \exists \text{ref} \cdot \Pi_{\{\text{tr,ref,ok,wait,v}\}} \]
\[ = \{ \text{A.10.2 (definition::reactive::II-R::R1)} \} \]
\[ R1(\Pi_R) \triangleleft \text{wait} \triangleright \exists \text{ref} \cdot \Pi_{\{\text{tr,ref,ok,wait,v}\}} \]
\[ = \{ \text{A.5.11 (law::relational-calculus::II::unwinding)} \} \]
\[ R1(\Pi_R) \triangleleft \text{wait} \triangleright (\exists \text{ref} \cdot \Pi_{\{\text{ref,ok,wait,v}\}} \land (tr' = tr)) \]
\[ = \{ \text{A.2.1 (law::predicate-calculus::exists::and::non-free)} \} \]
\[ R1(\Pi_R) \triangleleft \text{wait} \triangleright (\exists \text{ref} \cdot \Pi_{\{\text{ref,ok,wait,v}\}} \land (tr' = tr)) \]
\[ = \{ \text{A.1.4 (law::propositional-calculus::and::idempotence)} \} \]
\[ R1(\Pi_R) \triangleleft \text{wait} \triangleright \]
\[ (\exists \text{ref} \cdot \Pi_{\{\text{ref,ok,wait,v}\}} \land (tr' = tr)) \land (tr' = tr) \]
\[ = \{ \text{A.5.11 (law::relational-calculus::II::unwinding)} \} \]
\[ R1(\Pi_R) \triangleleft \text{wait} \triangleright \]
\[ (\exists \text{ref} \cdot \Pi_{\{\text{tr,ref,ok,wait,v}\}} \land (tr' = tr)) \]
\[ = \{ \text{A.10.1 (definition::reactive::R1)} \} \]
\[ R1(\Pi_R) \triangleleft \text{wait} \triangleright \]
\[ (\exists \text{ref} \cdot \Pi_{\{\text{tr,ref,ok,wait,v}\}}) \]
\[ = \{ \text{A.10.6 (law::reactive::R1::conditional)} \} \]
\[ R1(\Pi_R \triangleleft \text{wait} \triangleright (\exists \text{ref} \cdot \Pi_{\{\text{tr,ref,ok,wait,v}\}})) \]
\[ = \{ \text{A.10.4 (definition::reactive::R3)} \} \]
\[ R1(R3(\Pi_R \triangleleft \text{wait} \triangleright (\exists \text{ref} \cdot \Pi_{\{\text{tr,ref,ok,wait,v}\}}))) \]
\[ = \{ \text{A.13.5 (definition::CML::Skip)} \} \]
\[ R1(\text{Skip}) \]
A.13.4.8 \textit{law::CML::Skip::reactive-design}

Law A.13.15 (\textit{law::CML::Skip::reactive-design})

\[\text{Skip} = R1 \circ R3 (true \vdash (tr' = tr) \land \neg wait' \land (v' = v))\]

\textbf{Proof}

\[\text{Skip} = \{ \text{A.13.15 (law::CML::Skip::reactive-design)} \}\]
\[R1 \circ R3 (\neg \text{Skip} \vdash \text{Skip})\]
\[= \{ \text{A.13.10 (law::CML::Skip::not-wait::diverging)} \}\]
\[R1 \circ R3 (\neg ((tr' = tr) \land \neg ok \land \neg wait' \land (v' = v)) \vdash \text{Skip})\]
\[= \{ \text{A.9.8 (law::design::pre-ok)} \}\]
\[R1 \circ R3 (ok \vdash \text{Skip})\]
\[= \{ \text{A.9.8 (law::design::pre-ok)} \}\]
\[R1 \circ R3 (true \vdash \text{Skip})\]
\[= \{ \text{A.1.22 (law::propositional-calculus::negation::absorption)} \}\]
\[R1 \circ R3 (true \vdash (tr' = tr) \land \neg wait' \land (v' = v))\]
\[= \{ \text{A.9.5 (law::design::post-ok)} \}\]
\[R1 \circ R3 (true \vdash (tr' = tr) \land \neg wait' \land (v' = v))\]

\[\square\]

A.13.5 \textit{assignment}

A.13.5.1 \textit{law::CML::assignment::composition}

Law A.13.16 (\textit{law::CML::assignment::composition})

\[(x, y := e, f); (x, y := e, y) = (x, y := e, f)\]

A.13.5.2 \textit{law::CML::assignment::restriction}

Law A.13.17 (\textit{law::CML::assignment::restriction})

\[
((x, y := e, f) \mid x) = (x := e_1)_{+y}
\]
A.13.5.3  law::<CML>::assignment::substitution::one-point-rule

Law A.13.18  (law::<CML>::assignment::substitution::one-point-rule)

\[ (\exists v \bullet s[v/v'] \land P) = P[s] \]
\[ s ; P = P[s] \]
\[ s + tr_{@ref} ; P = P[s] \]

Proof

\[ (v := e + tr_{@ref} ; P) = \{ \text{A.4.1 (definition::<alphabet>::lifting)} \} \]
\[ (v, tr := e, tr)_{@ref} ; P = \{ \text{A.5.5 (law::<relational-calculus>::assignment::sequence)} \} \]
\[ P[e, tr / v, tr] = \{ \text{A.7.6 (law::<substitution}) \} \]
\[ P[e / v] = \{ \text{A.13.19 (law::<CML>::assignment::substitution)} \} \]
\[ P[v := e] \]

\[\Box\]

A.13.5.4  definition::<CML>::assignment::substitution

Definition A.13.6  (definition::<CML>::assignment::substitution)

\[ P[x_1, \ldots, x_n := e_1, \ldots, e_n] \equiv P[e_1, \ldots, e_n / x_1, \ldots, x_n] \]

A.13.5.5  law::<CML>::assignment::substitution

Law A.13.19  (law::<CML>::assignment::substitution)

\[ (s ; (w = e)) = (w = e)[s] \]

Proof

\[ s ; (w = e) = \{ \text{A.5.9 (definition::<relational-calculus>::sequence)} \} \]
\[ \exists v0 \bullet s[v0/v'] \land (w = e)[v0/v] = \{ \text{change of variable} \} \]
\[ \exists v \bullet s[v/v'] \land (w = e) = \{ \text{A.13.18 (law::<CML>::assignment::substitution::one-point-rule)} \} \]
\((w = e[s]) = \{ \text{identity, } w \text{ constant } \} (w = e)[s]\)

\[\Box\]

**A.13.5.6  definition::-CML::assignment**

**Definition A.13.7  (definition::-CML::assignment)**

\[x := x', y, y' \quad e = (x' = e) \land (y' = y)\]

**A.13.6  input**

**A.13.6.1  definition::-CML::input**

**Definition A.13.8  (definition::-CML::input)**

\[d?x : T \rightarrow A \triangleq \square \{ u : T \bullet \var_{RD} x :=_{RD} u \bullet d.u \rightarrow A \}\]

**A.13.6.2  law::-CML::input::absorption**

**Law A.13.20  (law::-CML::input::absorption)**

\[\text{lift}(s) ; d?x : T \rightarrow A = (\text{lift}(s) ; d.w \rightarrow A[w/x]) \square (\text{lift}(s) ; d?x : T \rightarrow A) \quad \text{for } w \text{ in } T\]

**Proof**

\[
\text{lift}(s) ; d?x : T \rightarrow A
\]
\[
= \{ A.13.8 (definition::-CML::input) \}
\]
\[
\text{lift}(s) ; \square \{ u : T \bullet \var_{RD} x :=_{RD} u \bullet d.u \rightarrow A \}
\]
\[
= \{ A.6.1 (law::-set-theory::union::left-subset) \}
\]
\[
\text{lift}(s) ;
\]
\[
\Box (\{ \var_{RD} x :=_{RD} w ; d.w \rightarrow A \}) \cup \{ u : T \bullet \var_{RD} x :=_{RD} u ; d.u \rightarrow A \}
\]
\[
= \{ A.13.25 (law::-CML::external-choice::distribution::union) \}
\]
\[
\text{lift}(s) ;
\]
\[
(\var_{RD} x :=_{RD} w ; d.w \rightarrow A) \square \{ u : T \bullet \var_{RD} x :=_{RD} u ; d.u \rightarrow A \}
\]
\[
= \{ A.11.5 (law::-reactive-design::decl::elimination) \}
\]
\[
\text{lift}(s) ;
\]
\[
(d.w \rightarrow A)(w/x) \square \{ u : T \bullet \var_{RD} x :=_{RD} u ; d.u \rightarrow A \}
\]
\[
= \{ A.13.8 (definition::-CML::input) \}
\]
\[
\text{lift}(s) ;
\]
\[ (d.w \rightarrow A(w/x)) \mathcal{D} (d?x : T \rightarrow A) = \{ 3.2.7 \ (\text{law::lift::external-choice}) \} \]

\[ (\text{lift}(s) ; d.w \rightarrow A(w/x)) \mathcal{D} (\text{lift}(s) ; d?x : T \rightarrow A) = \]

\[ A.13.7 \ \text{external-choice} \]

\[ A.13.7.1 \ \text{definition::CML::external-choice::distributed} \]

Definition A.13.9 (\text{definition::CML::external-choice::distributed})

\[ \mathcal{D} S \triangleq R1 \circ R3(\forall X : S \cdot \neg X \vdash (\forall X : S \cdot X) \prec (tr' = tr) \land \text{wait}' \triangleright (\exists X : S \cdot X)) \]

\[ A.13.7.2 \ \text{law::CML::external-choice::assignment} \]

Law A.13.21 (\text{law::CML::external-choice::assignment})

\[ \text{lift}(s) \mathcal{D} P \subseteq \text{lift}(s) \]

\[ \text{Proof} \]

\[ \text{lift}(s) \mathcal{D} P = \{ 3.1.6 \ (\text{definition::reactive-design::external-choice}) \} \]

\[ R1 \circ R3(\text{true} \land \neg \neg P_f^l) \]

\[ \vdash s_{+tr\oplus ref} \land \neg \text{wait}' \land P_f^l \]

\[ \prec (tr' = tr) \land \text{wait}' \triangleright \]

\[ (s_{+tr\oplus ref} \land \neg \text{wait}') \lor P_f^l) ]

\[ = \{ A.1.6 \ (\text{law::propositional-calculus::and::unit}) \} \]

\[ R1 \circ R3(\neg P_f^l) \]

\[ \vdash s_{+tr\oplus ref} \land \neg \text{wait}' \land P_f^l \]

\[ \prec (tr' = tr) \land \text{wait}' \triangleright \]

\[ (s_{+tr\oplus ref} \land \neg \text{wait}') \lor P_f^l) \]

\[ \subseteq \{ A.9.9 \ (\text{law::design::refinement::strengthen-post}) \} \]

\[ R1 \circ R3(\neg P_f^l \vdash s_{+tr\oplus ref} \land \neg \text{wait}') \]

\[ \subseteq \{ A.9.10 \ (\text{law::relational-calculus::refinement::weaken-pre}) \} \]

\[ R1 \circ R3(\text{true} \vdash s_{+tr\oplus ref} \land \neg \text{wait}') \]

\[ = \{ 3.2.5 \ (\text{definition::lift}) \} \]

\[ \text{lift}(s) \]
A.13.7.3  \textit{law::-reactive-design:-external-choice:-distributed:-not-wait}

Law A.13.22 (\textit{law::-reactive-design:-external-choice:-distributed:-not-wait})

\[(\Box S)_f = R1((\forall X : S \cdot \neg X_f \vdash \forall X : S \cdot X_f \triangleleft (tr' = tr) \land \text{wait'} \triangleright \exists X : S \cdot X_f))\]

\textbf{Proof}

\[(\Box S)_f\]
\[= \{ \text{A.13.9 (definition::-CML:-external-choice:-distributed)} \} \]
\[(R1 \circ R3(\forall X : S \cdot \neg X \vdash (\forall X : S \cdot X) \triangleleft (tr' = tr) \land \text{wait'} \triangleright (\exists X : S \cdot X)))_f\]
\[= \{ \text{A.10.20 (law::-reactive:-R1:-wait-substitution). twice } \} \]
\[R1((\forall X : S \cdot \neg X_f \vdash (\forall X : S \cdot X) \triangleleft (tr' = tr) \land \text{wait'} \triangleright (\exists X : S \cdot X)))_f\]
\[= \{ \text{A.7.6 (law::-substitution)} \} \]
\[R1((\forall X : S \cdot \neg X_f) \vdash ((\forall X : S \cdot X) \triangleleft (tr' = tr) \land \text{wait'} \triangleright (\exists X : S \cdot X)))_f\]
\[= \{ \text{A.7.6 (law::-substitution)} \} \]
\[R1((\forall X : S \cdot \neg X_f \vdash \forall X : S \cdot X_f \triangleleft (tr' = tr) \land \text{wait'} \triangleright \exists X : S \cdot X_f))\]

\[\Box\]

A.13.7.4  \textit{law::-reactive-design:-external-choice:-distributed:-post}

Law A.13.23 (\textit{law::-reactive-design:-external-choice:-distributed:-post})

\[(\Box S)_f = R1(ok \land (\forall X : S \cdot \neg X_f) \Rightarrow (\forall X : S \cdot X_f \triangleleft (tr' = tr) \land \text{wait'} \triangleright \exists X : S \cdot X_f))\]

\textbf{Proof}

\[(\Box S)_f\]
\[= \{ \text{A.13.22 (law::-reactive-design:-external-choice:-distributed:-not-wait)} \} \]
\[R1((\forall X : S \cdot \neg X_f \vdash (\forall X : S \cdot X_f \triangleleft (tr' = tr) \land \text{wait'} \triangleright \exists X : S \cdot X_f))_f\]
\[= \{ \text{A.10.20 (law::-reactive:-R1:-wait-substitution) } \} \]
\[R1((\forall X : S \cdot \neg X_f \vdash (\forall X : S \cdot X_f) \triangleleft (tr' = tr) \land \text{wait'} \triangleright (\exists X : S \cdot X_f))_f\]
\[= \{ \text{A.9.2 (law::-design:-ok'-substitution::true) } \} \]
\[R1((\forall X : S \cdot \neg X_f \vdash (\forall X : S \cdot X_f) \triangleleft (tr' = tr) \land \text{wait'} \triangleright (\exists X : S \cdot X_f))_f)\]
\[= \{ \text{A.7.6 (law::-substitution) } \} \]
\[R1((\forall X : S \cdot \neg X_f) \Rightarrow (\forall X : S \cdot X_f \triangleleft (tr' = tr) \land \text{wait'} \triangleright \exists X : S \cdot X_f))\]

\[\Box\]
A.13.7.5  \textit{law::reactive-design::external-choice::distributed::pre}

Law A.13.24 \textit{(law::reactive-design::external-choice::distributed::pre)}

\[ \neg ((\Box S)^f_i) = \neg R1(\neg (ok \land \forall X : S \cdot \neg X_f)) \]

Proof

\[ \neg ((\Box S)^f_i) \]
\[ = \{ \text{A.13.22 (law::reactive-design::external-choice::distributed::not-wait)} \} \]
\[ \neg R1(\neg (ok \land \forall X : S \cdot \neg X_f) \land \forall X : S \cdot X_f \land wait \land \exists X : S \cdot X_f)^f \]
\[ = \{ \text{A.10.20 (law::reactive::R1::wait-substitution)} \} \]
\[ \neg R1(\neg (ok \land \forall X : S \cdot \neg X_f) \land \forall X : S \cdot X_f \land wait \land \exists X : S \cdot X_f)^f) \]
\[ = \{ \text{A.9.3 (law::design::ok::substitution::false)} \} \]
\[ \neg R1(\neg (ok \land \forall X : S \cdot \neg X_f)) \]
\[ = \{ \text{A.7.6 (law::substitution)} \} \]
\[ \neg R1(\neg (ok \land \forall X : S \cdot \neg X_f)) \]

\[ \square \]

A.13.7.6  \textit{law::CML::external-choice::distribution::union}

Law A.13.25 \textit{(law::CML::external-choice::distribution::union)}

\[ \Box(\{P\} \cup S) = P \Box \Box S \]

Proof

\[ P \Box \Box S \]
\[ = \{ \text{A.11.1 (law::reactive-design::external-choice::distributed::associativity)} \} \]
\[ R1 \circ R3(\neg P^f_i \land \neg ((\Box S)^f_i) \land P^f_i \land (\Box S)^f_i \land wait \land \exists X : S \cdot \neg X_f)^f \]
\[ = \{ \text{A.13.24 (law::reactive-design::external-choice::distributed::pre)} \} \]
\[ R1 \circ R3(\neg P^f_i \land \neg R1(\neg (ok \land \forall X : S \cdot \neg X_f)) \land \forall X : S \cdot X_f \land wait \land \exists X : S \cdot X_f)^f \]
\[ = \{ \text{A.11.9 (law::reactive-design::R1::cancellation::pre)} \} \]
\[ R1 \circ R3(\neg P^f_i \land \neg \neg (ok \land \forall X : S \cdot \neg X_f) \land \forall X : S \cdot X_f \land wait \land \exists X : S \cdot \neg X_f)^f \]
\[ = \{ \text{A.1.20 (law::propositional-calculus::negation::double-negation)} \} \]
\[ R1 \circ R3(\neg P^f_i \land ok \land \forall X : S \cdot \neg X_f \land \forall X : S \cdot \neg X_f \land wait \land \exists X : S \cdot \neg X_f)^f \]
\[
\begin{align*}
R1 \circ R3 & \cup \{ \text{A.9.7 (law::design::pre-ok::Leibniz)} \} \\
& = \{ \text{A.13.23 (law::reactive-design::external-choice::distributed::post)} \} \\
& = \{ \text{A.9.6 (law::design::post::simplification)} \} \\
& = \{ \text{A.1.43 (law::propositional-calculus::conditional::simplification-1)} \} \\
& = \{ \text{A.9.4 (law::design::post-ok::Leibniz)} \} \\
\end{align*}
\]
\[ R1 \circ R3( \forall X : \{ P \} \cup S \cdot \neg X_f ) \]
\[ = \{ \text{A.2.3 (law::predicate-calculus::exists::detach)} \} \]

\[ R1 \circ R3( \forall X : \{ P \} \cup S \cdot \neg X_f ) \]
\[ = \{ \text{A.11.1 (law::reactive-design::external-choice::distributed::associativity)} \} \]
\[ \Box(\{ P \} \cup S) \]

A.13.7.7 law::CML::external-choice::idempotence

Law A.13.26 (law::CML::external-choice::idempotence)

\[ P \Box P = P \]

Proof

\[ P \Box P \]
\[ = \]
\[ R1 \circ R3( \neg P_f^1 \land \neg P_f^2 \vdash P_f^1 \land P_f^2 \triangleleft (t_r' = t_r) \land \text{wait} \triangleright P_f^1 \lor P_f^2 ) \]
\[ = \{ \text{A.1.4 (law::propositional-calculus::and::idempotence), twice} \} \]
\[ R1 \circ R3( \neg P_f^1 \vdash P_f^1 \triangleleft (t_r' = t_r) \land \text{wait} \triangleright P_f^1 \lor P_f^2 ) \]
\[ = \{ \text{A.1.10 (law::propositional-calculus::or::idempotence)} \} \]
\[ R1 \circ R3( \neg P_f^1 \vdash P_f^1 \triangleleft (t_r' = t_r) \land \text{wait} \triangleright P_f^1 ) \]
\[ = \{ \text{A.1.41 (law::propositional-calculus::conditional::idempotence)} \} \]
\[ R1 \circ R3( \neg P_f^1 \vdash P_f^2 ) \]
\[ = \{ \text{A.11.13 (law::reactive-design)} \} \]
\[ P \]

A.13.7.8 law::CML::external-choice::lift-distributive

Law A.13.27 (law::CML::external-choice::lift-distributive)

\[ \text{lift}(s) ; (P \Box Q) = (\text{lift}(s) ; P) \Box (\text{lift}(s) ; Q) \]
A.13.7.9 \textit{law::-CML:-external-choice:-monotonic}

Law A.13.28 \textit{(law::-CML:-external-choice:-monotonic)}

\[ P_1 \subseteq P_2 \Rightarrow P_1 \not\subset Q \subseteq P_2 \not\subset Q \]

\textbf{Proof} \quad \text{Assume:}

\[ P_1 \subseteq P_2 = [\neg P_1 t \Rightarrow \neg P_2 t] \wedge [\neg P_1 t \wedge P_2 t \Rightarrow P_1 t] \]

\[ P_1 \not\subset Q \]

\[ = \{ \text{3.1.6 (definition::-reactive-design:-external-choice)} \} \]

\[ R_1 \circ R_3 (\neg P_1 j \wedge \neg Q_2 j \Leftarrow P_1 j \wedge Q_1 j \Leftarrow (t' = t) \wedge wait' \Leftarrow P_1 j \vee Q_1 j) \]

\[ \subseteq \{ \text{A.9.9 (law::-design:-refinement:-strengthen-post)} \} \]

\[ R_1 \circ R_3 (\neg P_1 j \wedge \neg Q_2 j \Leftarrow P_1 j \wedge Q_1 j \Leftarrow (t' = t) \wedge wait' \Leftarrow P_2 j \vee Q_2 j) \]

\[ \subseteq \{ \text{A.9.10 (law::-relational-calculus:-refinement:-weaken-pre)} \} \]

\[ R_1 \circ R_3 (\neg P_2 j \wedge \neg Q_2 j \Leftarrow P_2 j \wedge Q_1 j \Leftarrow (t' = t) \wedge wait' \Leftarrow P_2 j \vee Q_2 j) \]

\[ = \{ \text{3.1.6 (definition::-reactive-design:-external-choice)} \} \]

\[ P_2 \not\subset Q \]

\[ \square \]

A.13.7.10 \textit{law::-CML:-external-choice:-Skip}

Law A.13.29 \textit{(law::-CML:-external-choice:-Skip)}

\[ \text{Skip} \not\subset P \subseteq \text{Skip} \]

\textbf{Proof}

\[ \text{Skip} \not\subset P \]

\[ = \{ \text{A.13.9 (law::-CML:-Skip:-lifted-assignment)} \} \]

\[ \text{lift}(v := v) \not\subset P \]

\[ \subseteq \{ \text{A.13.21 (law::-CML:-external-choice:-assignment)} \} \]

\[ \text{lift}(v := v) \]

\[ = \{ \text{A.13.9 (law::-CML:-Skip:-lifted-assignment)} \} \]

\[ \text{Skip} \]

\[ \square \]
A.13.8  **extra-choice**

A.13.8.1  **definition::CML:-extrachoice**

Definition A.13.10 (definition::CML:-extrachoice)

\[ P \boxplus Q = P \boxdot Q \]

A.13.8.2  **law::CML:-extrachoice:-monotonic**

Law A.13.30 (law::CML:-extrachoice:-monotonic)

\[ P_1 \subseteq P_2 \Rightarrow P_1 \boxplus Q \subseteq P_2 \boxplus Q \]

**Proof**

\[ P_1 \boxplus Q \]

= \{  A.13.10 (definition::CML:-extrachoice) \}

\[ P_1 \boxdot Q \]

\[ \subseteq \{  A.13.28 (law::CML:-external-choice:-monotonic), \text{assumption:}\ P_1 \subseteq P_2 \} \]

\[ P_2 \boxdot Q \]

= \{  A.13.10 (definition::CML:-extrachoice) \}

\[ P_2 \boxplus Q \]

\[ \square \]

A.13.9  **parallel-composition**

A.13.9.1  **definition::CML:-parallel-by-merge**

Definition A.13.11 (definition::CML:-parallel-by-merge)

\[(P \parallel_M Q) \equiv ((P ; U0(m)_A \parallel Q ; U1(m)_B+M)_{m} ; M)\]

for (A = outα(P) \ {m'}) and (B = outα(Q) \ {m'})

A.13.9.2  **law::CML:-parallel:-cotermination**

Law A.13.31 (law::CML:-parallel:-cotermination)

\[(P \land ok') \parallel (Q \land ok') = (P \parallel Q) \land ok' \]

A.13.9.3  **law::CML:-parallel:-disjunctivity**

Law A.13.32 (law::CML:-parallel:-disjunctivity)

\[ P \parallel (Q \cap R) = (P \parallel Q) \cap (P \parallel R) \]

134
A.13.9.4  law::-CML:-parallel-post

Law A.13.33  (law::-CML:-parallel-post)

\((P \parallel M\text{\tiny CSP} Q)_f^j = (P_f^j \parallel M_{\text{\tiny ok}} Q_f^j)\)

Proof

\((P \parallel M Q)_f^j = \{ A.13.39 \ \text{(law::-CML:-parallel:-wait-substitution)} \} \)

\((P_f^j ; U0(m) \parallel Q_f^j ; U1(m))_{+m}^j ; M^t \)

\(= \{ A.7.6 \ \text{(law::-substitution), twice} \} \)

\((P_f^j ; U0(m) \parallel Q_f^j ; U1(m))_{+m}^j ; 0_{\text{\tiny ok}} \land 1_{\text{\tiny ok}} \land M^t \)

\(= \{ A.5.23 \ \text{(law::-relational-calculus:-sequence:-one-point:-right)} \} \)

\((P_f^j ; U0(m) \parallel Q_f^j ; U1(m))_{+m}^j [\text{true, true}/0, 1, 0, 1] ; M^t[\text{true, true}/0, 1, 1, 0] \)

\(= \{ A.12.6 \ \text{(definition::-CSP:-M-ok)} \} \)

\((P_f^j ; U0(m) \parallel Q_f^j ; U1(m))_{+m}^j [\text{true, true}/0, 1, 1, 0] ; M_\text{ok} \)

\(= \{ A.7.6 \ \text{(law::-substitution)} \} \)

\((P_f^j ; U0(m) \parallel Q_f^j ; U1(m))_{+m}^j [\text{true, true}/0, 1, 1, 0]_{+m}^j ; M_\text{ok} \)

\(= \{ A.8.3 \ \text{(law::-concurrency:-disjoint-parallel:-substitution:-separation)} \} \)

\((P_f^j ; U0(m)) [\text{true}/0, 1, 0, 1] \parallel (Q_f^j ; U1(m)) [\text{true}/1, 0, 1, 0]_{+m}^j ; M_\text{ok} \)

\(= \{ A.13.7 \ \text{(law::-CML:-merge:-separating-sim:-Ui:-i.ok')}, \ \text{twice} \} \)

\((P_f^j ; V0(m, n) \parallel Q_f^j ; V1(m, n))_{+m}^j ; M_\text{ok} \)

\(= \{ A.13.11 \ \text{(definition::-CML:-parallel-by-merge)} \} \)

\((P_f^j \parallel M_{\text{\tiny ok}} Q)_f^j \)

\(\square\)

A.13.9.5  law::-CML:-parallel-pre

Law A.13.34  (law::-CML:-parallel-pre)  Suppose that either \(P\) is \(\text{CSP2}\) or \(Q\) is. Then

\((P \parallel M\text{\tiny CSP} Q)_f^j = (P_f^j \parallel M_{\text{\tiny CSP}\text{\tiny \neg} \text{\tiny ok}} Q_f^j) \lor (P_f^j \parallel M_{\text{\tiny CSP}\text{\tiny \neg} \text{\tiny ok}} Q_f^j)\)

where

\(M_{\text{\tiny CSP}\text{\tiny \neg} \text{\tiny ok}} \equiv M_{\text{\tiny CSP}}[\text{false, false}/0, 1, 0] \)

Proof

\((P \parallel M\text{\tiny CSP} Q)_f^j \)

\(= \{ A.13.39 \ \text{(law::-CML:-parallel:-wait-substitution)} \} \)

135
\[
(P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; M^f_{CSP}
\]
\[
= \{ \ A.7.6 \ (law::-substitution) \ \}
\]
\[
(P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; M^f_{CSP}
\]
\[
= \{ \ A.12.5 \ (law::CSP::M-CSP::ok'-substitution) \ \}
\]
\[
(P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; \neg (0.ok \land 1.ok) \land M^f_{CSP}
\]
\[
= \{ \ A.1.19 \ (law::propositional-calculus::negation::De-Morgan) \ \}
\]
\[
(P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; (\neg 0.ok \lor \neg 1.ok) \land M^f_{CSP}
\]
\[
= \{ \ A.5.16 \ (law::relational-calculus::sequence::disjunctivity) \ \}
\]
\[
(P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; \neg 0.ok \land M^f_{CSP}
\]
\[
\lor (P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; \neg 1.ok \land M^f_{CSP}
\]
\[
= \{ \ A.1.21 \ (law::propositional-calculus::negation::excluded-middle), twice \ \}
\]
\[
(P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; \neg 0.ok \land (1.ok \lor \neg 1.ok) \land M^f_{CSP}
\]
\[
\lor (P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; (0.ok \lor \neg 0.ok) \land \neg 1.ok \land M^f_{CSP}
\]
\[
= \{ \ A.1.5 \ (law::propositional-calculus::and::or-distributivity) \ \}
\]
\[
A.5.16 \ (law::relational-calculus::sequence::disjunctivity) \ \}
\]
\[
(P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; \neg 0.ok \land 1.ok \land M^f_{CSP}
\]
\[
\lor (P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; \neg 0.ok \land 1.ok \land M^f_{CSP}
\]
\[
\lor (P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; \neg 0.ok \land 1.ok \land M^f_{CSP}
\]
\[
\lor (P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; \neg 0.ok \land 1.ok \land M^f_{CSP}
\]
\[
= \{ \ A.1.12 \ (law::propositional-calculus::or::subsumption) \ \}
\]
\[
(P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; \neg 0.ok \land 1.ok \land M^f_{CSP}
\]
\[
\lor (P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; \neg 0.ok \land 1.ok \land M^f_{CSP}
\]
\[
\lor (P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; 0.ok \land \neg 1.ok \land M^f_{CSP}
\]
\[
= \{ \ A.5.23 \ (law::relational-calculus::sequence::one-point::right), three times \ \}
\]
\[
(P_f ; U0(m) \parallel Q_f ; U1(m))_{+m}[false, true/0.ok', 1.ok'] ; M^f_{CSP, ok}
\]
\[
\lor (P_f ; U0(m) \parallel Q_f ; U1(m))_{+m}[false, false/0.ok', 1.ok'] ; M^f_{CSP, ok}
\]
\[
\lor (P_f ; U0(m) \parallel Q_f ; U1(m))_{+m}[true, false/0.ok', 1.ok'] ; M^f_{CSP, ok}
\]
\[
= \{ \ A.8.3 \ (law::concurrency::disjoint-parallel::substitution::separation), three times \ \}
\]
\[
(((P_f ; U0(m))[false/0.ok'])
\]
\[
\parallel (Q_f ; U1(m))[true/1.ok']
\]
\[
)_{+m} ;
\]
\[
M^f_{CSP, ok}
\]
\[
)\}
\]
\[
\lor (((P_f ; U0(m))[false/0.ok'])
\]
\[
\parallel (Q_f ; U1(m))[false/1.ok']
\]
\[
)_{+m} ;
\]
\[
M^f_{CSP, ok}
\]
\[ \forall ( ((P_f \ U0(m))[true/0.ok]) \parallel (Q_f \ U1(m))[false/1.ok]) +_m; \]
\[ M_{CSP_{\sim \ ok}} \]
\[ = \{ \text{A.13.6 (law::-CML:-merge:-separating-sim:-Ui:-i.ok'-'substitution), twice} \} \]
\[ (((P_f \ \neg \ ok \land (\text{end \ ok} \ U0(n) \ \var \ 0.ok)) \parallel (Q_f \ \neg \ ok \land (\text{end \ ok} \ U1(n) \ \var \ 1.ok)) +_m; \]
\[ M_{CSP_{\sim \ ok}} \]
\[ \forall ( ((P_f \ \neg \ ok \land (\text{end \ ok} \ U0(n) \ \var \ 0.ok)) \parallel (Q_f \ \neg \ ok \land (\text{end \ ok} \ U1(n) \ \var \ 1.ok)) +_m; \]
\[ M_{CSP_{\sim \ ok}} \]
\[ \forall ( ((P_f \ \text{end \ ok} \ U0(n) \ \var \ 0.ok)) \parallel (Q_f \ \text{end \ ok} \ U1(n) \ \var \ 1.ok)) +_m; \]
\[ M_{CSP_{\sim \ ok}} \]
\[ = \{ \text{A.5.23 (law::-relational-calculus:-sequence:-one-point:-right), six times} \} \]
\[ (((P_f^I \ \text{end \ ok} \ U0(n) \ \var \ 0.ok)) \parallel (Q_f^I \ \text{end \ ok} \ U1(n) \ \var \ 1.ok)) +_m; \]
\[ M_{CSP_{\sim \ ok}} \]
\[ \forall ( ((P_f^I \ \text{end \ ok} \ U0(n) \ \var \ 0.ok)) \parallel (Q_f^I \ \text{end \ ok} \ U1(n) \ \var \ 1.ok)) +_m; \]
\[ M_{CSP_{\sim \ ok}} \]
\[ \forall ( ((P_f^I \ \text{end \ ok} \ U0(n) \ \var \ 0.ok)) \parallel (Q_f^I \ \text{end \ ok} \ U1(n) \ \var \ 1.ok)) +_m; \]
\[ M_{CSP_{\sim \ ok}} \]
\[
\begin{array}{c}
\{ \text{ A.5.32 (law::-relational-calculus:-var:-parallel), six times } \} \\
((P_f^l ; \text{end ok ; } U_0(n)) \\
\parallel (Q_f^l ; \text{end ok ; } U_1(n)) \\
)_{+m} ; \\
\text{var} 0.\text{ok}, 1.\text{ok} ; M_{\text{CSP}_{\text{\sim}}} \text{ ok} \\
\} \\
\vee ((P_f^l ; \text{end ok ; } U_0(n)) \\
\parallel (Q_f^l ; \text{end ok ; } U_1(n)) \\
)_{+m} ; \\
\text{var} 0.\text{ok}, 1.\text{ok} ; M_{\text{CSP}_{\text{\sim}}} \text{ ok} \\
) \\
\vee ((P_f^l ; \text{end ok ; } U_0(n)) \\
\parallel (Q_f^l ; \text{end ok ; } U_1(n)) \\
)_{+m} ; \\
\text{var} 0.\text{ok}, 1.\text{ok} ; M_{\text{CSP}_{\text{\sim}}} \text{ ok} \\
) \\
\text{assumption: } P \text{ is CSP2} \\
\{ \text{ A.12.4 (law::-CSP:-CSP2:-wait-ok-monotonic) } \\
\text{ A.8.1 (law::-concurrency:-disjoint-parallel:-monotonic) } \\
\text{ A.1.12 (law::-propositional-calculus:-or:-subsumption) } \} \\
((P_f^l ; \text{end ok ; } U_0(n)) \\
\parallel (Q_f^l ; \text{end ok ; } U_1(n)) \\
)_{+m} ; \\
\text{var} 0.\text{ok}, 1.\text{ok} ; M_{\text{CSP}_{\text{\sim}}} \text{ ok} \\
) \\
\vee ((P_f^l ; \text{end ok ; } U_0(n)) \\
\parallel (Q_f^l ; \text{end ok ; } U_1(n)) \\
)_{+m} ; \\
\text{var} 0.\text{ok}, 1.\text{ok} ; M_{\text{CSP}_{\text{\sim}}} \text{ ok} \\
) \\
= \{ \text{ A.12.3 (definition::-CSP:-M-CSP-2) } \} \\
(P_f^l ; \text{end ok} \parallel_{M_{\text{CSP}_{\text{2}}}} Q_f^l ; \text{end ok}) \\
\vee (P_f^l ; \text{end ok} \parallel_{M_{\text{CSP}_{\text{2}}}} Q_f^l ; \text{end ok}) \\
\end{array}
\]
A.13.9.6  \textit{law::CML::parallel::CSP2}

Law A.13.35 (\textit{law::CML::parallel::CSP2}) Suppose that \(Q = \text{CSP2}(Q)\), then

\[
\text{CSP2}(P) \parallel Q \subseteq \text{CSP2}(P \parallel Q)
\]

\textit{Proof}

\[
\begin{align*}
\text{CSP2}(P) \parallel Q & = \{ \text{A.10.3 (law::reactive::J::splitting)} \} \\
(P^f \lor (P^t \land ok')) \parallel Q & = \{ \text{A.13.32 (law::CML::parallel::disjunctivity)} \} \\
(P \parallel Q) \lor ((P^t \land ok') \parallel Q) & = \{ \text{assumption} \} \\
(P \parallel \text{CSP2}(Q)) \lor ((P^t \land ok') \parallel \text{CSP2}(Q)) & = \{ \text{A.10.3 (law::reactive::J::splitting)} \} \\
(P \parallel (Q^t \lor (Q^t \land ok'))) \lor ((P^t \land ok') \parallel (Q^t \lor (Q^t \land ok'))) & = \{ \text{A.13.32 (law::CML::parallel::disjunctivity)} \} \\
(P \parallel Q^f) \lor (P^f \parallel (Q^t \land ok')) \lor ((P^t \land ok') \parallel Q^f) \lor ((P^t \land ok') \parallel (Q^t \land ok')) & \subseteq \{ \text{A.5.12 (law::relational-calculus::refinement:: nondeterministic-choice)} \} \\
(P \parallel Q^f) \lor ((P^t \land ok') \parallel (Q^t \land ok')) & = \{ \text{A.7.6 (law::substitution)} \} \\
(P \parallel Q^f) \lor ((P^t \land ok') \parallel (Q^t \land ok')) & = \{ \text{A.10.3 (law::reactive::J::splitting)} \}
\end{align*}
\]

\[
\text{CSP2}(P \parallel Q)
\]
A.13.9.7 \textit{law::CML::parallel::distribution-external-choice}

Law A.13.36 (\textit{law::CML::parallel::distribution-external-choice})

\[(P \parallel Q) \parallel R = (P \parallel R) \parallel (Q \parallel R)\]

A.13.9.8 \textit{law::CML::parallel::step-law}

Law A.13.37 (\textit{law::CML::parallel::step-law}) Suppose \(P = (a \rightarrow R) \boxtimes S\) and \(a \notin \alpha(Q)\), then

\[P \parallel Q = (a \rightarrow (R \parallel Q)) \boxtimes (P \parallel Q)\]
\[P \parallel Q = ((a \rightarrow R) \boxtimes S) \parallel Q\]

A.13.9.9 \textit{law::CML::parallel::substitution}

Law A.13.38 (\textit{law::CML::parallel::substitution})

\[(P^b \parallel M_f^b \ Q_f)[s] = ((P[s]^b_f ; V0) \parallel (Q[s]_f ; V1))_{+m} ; M_f^b\]

\textit{Proof}

\[\begin{align*}
(P^b_f \parallel M^b_f \ Q_f)[s] \\
= \{ \text{A.13.11 (definition::CML::parallel-by-merge)} \} \\
((P^b_f ; V0 \parallel Q_f ; V1)_{+m} ; M^b_f)[s] \\
= \{ \text{A.7.6 (law::substitution)} \} \\
((P^b_f ; V0 \parallel Q_f ; V1)_{+m})[s] ; M^b_f \\
= \{ \text{A.7.6 (law::substitution), state variables not shared} \} \\
((P^b_f ; V0 \parallel Q_f ; V1)[s])_{+m} ; M^b_f \\
= \{ \text{A.7.6 (law::substitution)} \} \\
((P^b_f ; V0)[s] \parallel (Q_f ; V1)[s])_{+m} ; M^b_f \\
= \{ \text{A.7.6 (law::substitution)} \} \\
((P[s]^b_f ; V0) \parallel (Q[s]_f ; V1))_{+m} ; M^b_f \\
\end{align*}\]

\[\square\]

A.13.9.10 \textit{law::CML::parallel::wait-substitution}

Law A.13.39 (\textit{law::CML::parallel::wait-substitution})

\[(P \parallel Q)_f = (P_f ; U0(m) \parallel Q_f ; U1(m))_{+m} ; M\]

\textit{Proof}

\[\]
(P ∥ Q)_f = \{ \text{A.13.11 (definition::CML::parallel-by-merge)} \}
\((P ; U0(m) ∥ Q ; U1(m))_f + M) = \{ \text{A.7.6 (law::substitution)} \}
\((P ; U0(m) || Q ; U1(m))_f + M) = \{ \text{A.7.6 (law::substitution)} \}
(P ; U0(m) || Q ; U1(m))_f + M = \{ \text{A.8.2 (law::concurrency::disjoint-parallel::substitution::plain)} \}
(P ; U0(m)) || (Q ; U1(m)) + M = \{ \text{A.7.6 (law::substitution)} \}
(P_f ; U0(m) || Q_f ; U1(m)) + M
\eqno{\Box}
Contributors:

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

Editors:

Jim Woodcock, York

Reviewers:

Joey Coleman, Aarhus
Marcel Oliveira, UFPE
Richard Payne, Newcastle
Document History

<table>
<thead>
<tr>
<th>Ver</th>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>06-03-2013</td>
<td>Samuel Canham</td>
<td>Initial document version</td>
</tr>
<tr>
<td>0.2</td>
<td>06-03-2013</td>
<td>Jim Woodcock</td>
<td>Working version</td>
</tr>
</tbody>
</table>
# Contents

1 Revised Semantics .......................... 5
   1.1 Timed Testing Traces .................. 6
      1.1.1 Healthiness Conditions .......... 8
      1.1.2 STOP .................................. 10
      1.1.3 Assignment .......................... 10
      1.1.4 SKIP .................................. 11
      1.1.5 Sequential Composition .......... 11
      1.1.6 Prefix ................................ 11
      1.1.7 Internal Choice .................... 12
      1.1.8 External Choice .................... 12
      1.1.9 Parallel Composition ............ 12
         1.1.9.0.1 Proof: .......................... 14
      1.1.10 Hiding ................................ 14
      1.1.11 Timeout ................................ 15
      1.1.12 Recursion ........................... 15
   1.2 Lowe & Ouaknine’s Axioms ............. 16
      1.2.1 Well Foundedness ................... 16
      1.2.2 Prefix Closure ...................... 16
      1.2.3 Refusals ............................... 17
      1.2.4 Timelock Freedom ................... 17
      1.2.5 Zeno Freedom ............................ 17
      1.2.6 Time-guardedness ..................... 17
Chapter 1

Revised Semantics

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

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

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

Consider the following CSP process: $a \rightarrow b \rightarrow \text{STOP}$. 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, $a \rightarrow b \rightarrow \text{STOP}$, 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 meander offer (that is, a subset of our original offer), say only $\{b, c\}$, then it would still refuse. Here are all the refusals:

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

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

As well as being able to see a process’s failures, an observer can also detect the passage
of time, up to a certain granularity. A refusal experiment is made to mark the passage of each time granularity.

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

- **Assignment**: \(( v := e )\).
- **Specification statement**: \( w : [ \text{pre}, \text{post} ] \).
- **Deadlocked action**: \( \text{STOP} \).
- **Successful termination**: \( \text{SKIP} \).
- **Sequential composition**: \( P ; Q \).
- **Prefixed action**: \( a \rightarrow P \).
- **Internal choice**: \( P \sqcap Q \).
- **External choice**: \( P \circ Q \).
- **Timeout**: \( P \nrightarrow Q \).
- **Parallel composition**: \( P \parallel Q \).
- **Hiding**: \( P \setminus \mathcal{A} \).
- **Recursion**: \( \mu X \bullet P(X) \).

## 1.1 Timed Testing Traces

Let \( \Sigma \) be the universe of events. Our semantic domain consists of traces with embedded refusal sets. For example, the trace

\[
(a, b, \{b, c\}, \emptyset, c)
\]

represents the observation:

- The trace \((a, b)\) occurred in the first time interval.
- At the end of this trace, the process refused the set of events \(\{b, c\}\).
- No events were observed during the second time interval.
- The third time interval is incomplete, but the trace \((c)\) was observed so far.

Traces bearing this structure are drawn from the following set:

**Definition 1.1.1**

\[
\text{timedTrace} \triangleq (\Sigma + \mathcal{P}(\Sigma))^* \]

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

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

**Definition 1.1.2**

$$\textit{squash}(\emptyset) = \emptyset$$
$$\textit{squash}(f) = \langle f(\min(\text{dom } f)) \rangle \upharpoonright \textit{squash}(\{\min(\text{dom } f)\} \triangleleft f)$$
$$t \upharpoonright S = \textit{squash}(t \triangleright S)$$

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

**Definition 1.1.3**

$$\textit{events}(t) = t \upharpoonright \Sigma$$
$$\textit{refsduring}(t) = \text{ran}(t \triangleright \mathbb{P}(\Sigma))$$
$$\textit{refusals}(t) = \bigcup \textit{refsduring}(t)$$
$$\textit{rejects}(t) = \bigcap \textit{refsduring}(t)$$

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

**Lemma 1.1.1 (Trace extraction)**

$$\text{trace}(\emptyset) = \emptyset$$
$$\text{trace}(\langle a \rangle \upharpoonright t) = \langle a \rangle \upharpoonright \text{trace}(t) \quad \text{if } a \in \Sigma^{\text{lock}}$$
$$\text{trace}(\langle X \rangle \upharpoonright t) = \text{trace}(t) \quad \text{if } X \in \mathbb{P}(\Sigma)$$

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

**Definition 1.1.4 (Testing trace precedence)** Let $a \in \Sigma \land X \subseteq Y$, then

$$\langle \rangle \preceq a$$
$$\langle a \rangle \upharpoonright t \preceq \langle a \rangle \upharpoonright u \quad \text{if } t \preceq u$$
$$\langle X, \tock \rangle \upharpoonright t \preceq \langle Y, \tock \rangle \upharpoonright u \quad \text{if } t \preceq u$$

This is a stronger relation than the usual prefix relation on traces, $\prec \preceq \preceq$:

**Lemma 1.1.2 (Precedence traces)**

$$t \preceq u \Rightarrow \text{trace}(t) \preceq \text{trace}(u)$$

*Proof* by induction on $t$.

A similar result holds for the refusals over testing traces:
Lemma 1.1.3 (Precedence refusals)

\[ t \preceq u \land a \in \text{refusals}(t) \Rightarrow a \in \text{refusals}(u) \]

We also define a function, \( tocks(t) \), which takes a trace and replaces all the refusal sets with a new event, \( tock \), useful for placing conditions on time.

**Definition 1.1.5**

\[
\begin{align*}
tocks(\langle \rangle) &= \langle \rangle \\
tocks(a \mapsto u) &= \langle a \rangle \mapsto tocks(u) \\
tocks(X \mapsto u) &= \langle tock \rangle \mapsto tocks(u)
\end{align*}
\]

We introduce four observation variables:

- **ok, ok’**: These are the observation variables from designs [?, 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 [?, Chapter 3]. The observation \( wait \) describes the situation in which a process occupies a waiting state of its sequential predecessor, whilst \( wait’ \) describes the situation in which the process has reached a waiting state. The combination of \( ok \) and \( wait \) and their dashed counterparts allow sequential combination to be defined as relational composition.
- **rt, rt’**: These are the observations of the trace of the previous process \( (rt) \) and the current process \( (rt’). \)
- **v, v’**: These are the variables which record our observations of the initial and final state of the current process.

We also introduce a derived variable - \( tt’ \) - 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 \( [\] \) or by sequential composition - it can always be replaced by \( rt’ - rt \) in any expression.

### 1.1.1 Healthiness Conditions

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

**Definition 1.1.6**

\[ P^c_b = P[b, c/\text{wait}, ok'] \]

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

There are six healthiness conditions.

The first requirement is that \( tt’ \) is well-defined. This requires that the observation of \( rt \) prefixes the observation of \( rt’ \).
Definition 1.1.7 (RT1)

\[ RT1(P) = P \land (rt \leq rt') \]

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

Definition 1.1.8 (RT2)

\[ RT2(P) = P[\emptyset, tt'/rt, rt'] \]

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

Definition 1.1.9 (RT3)

\[ RT3(P) = RT01(\text{true} \vdash (\alpha \Pi \prec wait \succ P)) \]

where \( \alpha \Pi = \{rt, rt', v, v', wait, wait'\} \)

Our fourth healthiness condition corresponds to \( CSP1 \) in Hoare & He’s theory of CSP (see [? , p.208]). If \( P \)'s predecessor is in an unstable state, then \( P \) will not be started and we have \( \neg ok \). What contribution will \( P \) now make to the divergent behaviour of its predecessor? It is allowed to behave almost arbitrarily: it cannot destroy the structure of the testing traces, nor can it interfere with the relationship that binds them together.

Definition 1.1.10 (RT4)

\[ RT4(P) = RT01(\neg ok) \lor P \]

Our fifth healthiness condition states that \( P \) must be monotonic in the value of the \( ok' \) variable, just like a design: \( P \) cannot demand instability and nontermination.

Definition 1.1.11 (RT5)

\[ RT5(P) = (\neg P^f) \Rightarrow (P^t \land ok') \]

where \( P^x = P[x/ok'] \)

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

The last healthiness condition ensures that all processes satisfy Lowe & Ouaknine's Zeno freedom condition. This is a bounded-speed condition which states that there is a bound \( n \) on the number of events that can be performed in the first \( k \) time units. Note that \( \#s \) is the length of the sequence \( s \).
Definition 1.1.12 (\(RT6\))

\[
RT6(P) = P \land \forall k. \exists n. \#(ttocks(tt') \uparrow tock) \leq k \Rightarrow \#(tt') \leq n
\]

Lemma 1.1.4 (\(RT\) functions are commuting monotonic idempotents)

1. \(RT1 - RT6\) are all monotonic idempotents.
2. \(RT1 - RT6\) all commute.

Definition 1.1.13 (\(RT\))

\[
RT \equiv RT1 \circ RT2 \circ RT3 \circ RT4 \circ RT5 \circ RT6
\]

We always assume that any constituent processes in a process definition are themselves \(RT\)-healthy. Any process which is an otherwise \(RT\)-healthy design is automatically \(RT4\) and \(RT5\)-healthy. A process which refers only to \(tt'\) and not directly to \(rt\) or \(rt'\) (except for as required by substitutions that are expansions of substitutions for \(tt'\)) will always be \(RT2\)-healthy.

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 [???]).

### 1.1.2 STOP

Our first language construct is the deadlocked action: \(STOP\). This action is an \(RT\)-healthy design with precondition true that never engages in any events and is perpetually waiting. In the postcondition, we must have that \(events(tt') = \langle \rangle\): no events are ever observed. \(STOP\) deadlock 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. It is \(RT\)-healthy.

Definition 1.1.14 (Deadlock)

\[
STOP_{rt} = RT1(true \vdash events(tt') = \langle \rangle \land wait')
\]

### 1.1.3 Assignment

For the assignment \(v := e\), we make the simplifying assumption that the expression \(e\) is well defined (we address this assumption in Chapter ???). 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 (\(v' = e\)). This design is then made healthy with \(RT0 \circ RT1 \circ RT3\), which we abbreviate to \(RT013\). Actually, it is by construction \(RT2\)-healthy (it does not constrain \(rt\) inappropriately, and we therefore do not need to enforce it) and \(RT4\) and \(RT5\)-healthy (it is a reactive design).

Definition 1.1.15 (Assignment)

\[
(x :=_{rt} e) = RT1(true \vdash tt' = \langle \rangle \land \neg wait' \land (v' = e))
\]
1.1.4 SKIP

We define SKIP to be the vacuous assignment.

Definition 1.1.16 (Termination)

\[ \text{SKIP}_{RT} = (v :=_{RT} v) \]

1.1.5 Sequential Composition

Sequential composition is simply relational composition, given our healthiness conditions. It is closed under the Healthiness Conditions.

Definition 1.1.17 (Sequential composition)

\[ P ;_{RT} Q = P ; Q \]

1.1.6 Prefix

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

The first case is when no events have been observed in the trace: \( \text{events}(tt') = \langle \rangle \). Over this period, the event \( a \) must not be refused: \( a \notin \text{refusals}(tt') \). Let \( \text{clean}(t, a) = (\text{events}(t) = \langle \rangle \land a \notin \text{refusals}(t)) \). The values of variables must not be changed and the process must not be waiting: \( v' = v \land \text{wait}' \). The process cannot diverge in this case.

In the second case, an event has been observed and it must have been the \( a \)-event: the first event must be \( a \). Let us name the activity of the process before the event occurs \( u \) - we will then have that \( \text{events}(u) = \langle \rangle \). The event \( a \) must not be refused during \( u \): \( a \notin \text{refusals}(u) \). After the event occurs, the action will continue as \( P \). The process can diverge if \( P \) can diverge if it is started with \( u < a > \) appended to \( rt \).

Definition 1.1.18 (Prefix)

\[ a \rightarrow_{RT} P = \begin{cases} \exists u \bullet \text{clean}(u, a) \land u \leftarrow \langle a \rangle \leq tt' \land P'^{f}_{f}[rt \leftarrow u \leftarrow \langle a \rangle / rt'] \\ \langle \text{events}(tt') = \langle a \rangle \rangle \end{cases} \]

Lemma 1.1.5 (Precondition/Postcondition form)

\[ a \rightarrow_{RT} P = \begin{cases} \forall u \bullet \text{clean}(u, a) \land u \leftarrow \langle a \rangle \leq tt' \Rightarrow \neg P'^{f}_{f}[rt \leftarrow u \leftarrow \langle a \rangle / rt'] \\ \text{RT} \vdash (\text{clean}(tt', a) \land (v' = v) \land \text{wait}') \\ \vee (\exists u \bullet \text{clean}(u, a) \land u \leftarrow \langle a \rangle \leq tt' \land P'^{f}_{f}[rt \leftarrow u \leftarrow \langle a \rangle / rt']) \end{cases} \]
1.1.7 Internal Choice

Internal choice is simply disjunction, as usual.

Definition 1.1.19 (Internal choice)

\[ P \sqcap_{rt} Q = P \lor Q \]

Lemma 1.1.6 (Precondition/Postcondition form)

\[ P \sqcap_{rt} Q = \text{RT} (\neg P_f^i \land \neg Q_f^i \lor P_f^i \lor Q_f^i) \]

1.1.8 External Choice

External choice is, of course, more involved than internal choice. The process \( P \boxtimes_{rt} Q \) diverges whenever either of its operands diverges (it is strict). In the external choice between \( P \) and \( Q \), the two actions are run in parallel until something observable occurs: one of the actions performs a visible event or one of the actions terminates. At that point the other action is discarded and the choice is made. Clearly, the two actions must agree on how long to wait, and this is formalised as \( (P \land Q)[rt \sqcap \text{idleprefix}(tt')/rt'] \). Subsequent behaviour is described by \( (P \lor Q) \).

Definition 1.1.20 (External choice)

\[ P \boxtimes_{rt} Q = \text{RT} (\neg (P \lor Q)_f^i \lor (P_f^i \land Q_f^i))[rt \sqcap \text{idleprefix}(tt')/rt'] \land (P_f^i \lor Q_f^i) \]

The difference between internal and external choice can be seen by comparing the following two processes:

\[ a \to \text{STOP} \sqcap b \to \text{STOP} \]

and

\[ a \to \text{STOP} \boxtimes b \to \text{STOP} \]

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

1.1.9 Parallel Composition

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

We call two timed reactive designs disjoint if they share no programming variables; they are allowed, of course, to share observational variables. This rules out shared variable parallelism.
The precondition of the parallel composition of \( P \) and \( Q \) is the conjunction of the preconditions of \( P \) and \( Q \). The postcondition merges the intermediate or final states of the two processes. Since the program variables are partitioned, the equation \( (\nu' = \nu) \) 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. The testing traces parallel operator has already taken care of \( rt' \).

**Definition 1.1.21 (Parallel composition)** for disjoint \( P \) and \( Q \)

\[
P \parallel_{A\textup{tr}} Q = RT \left( \begin{array}{l} 
\neg (P \lor Q) \\
\exists \text{wait}_1, \text{wait}_2, \text{tt}_1, \text{tt}_2 \bullet \\
\text{tt}' \in \text{tt}_1 \parallel A \text{tt}_2 \\
\land (\text{wait}' = \text{wait}_1 \lor \text{wait}_2) \\
\land P'[\text{wait}_1, rt \bowtie \text{tt}_1/\text{wait}', rt'] \\
\land Q'[\text{wait}_2, rt \bowtie \text{tt}_2/\text{wait}', rt'] 
\end{array} \right)
\]

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

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

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

**Definition 1.1.22**

\[
X \cap_A Y \triangleq \mathbb{P}((X \cap A) \cup (Y \cap A) \cup (X \cap Y))
\]

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

**Definition 1.1.23 (Trace interleaving)**

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

\[
\begin{align*}
\mathcal{t} \parallel_A u & = u \parallel_A t \\
\langle \rangle \parallel_A \langle \rangle & = \langle \rangle \\
\langle \rangle \parallel_A \langle b \rangle \bowtie u & = \langle \rangle \\
\langle \rangle \parallel_A \langle d \rangle \bowtie u & = \langle \langle d \rangle \bowtie \nu \mid \nu \in \langle \rangle \parallel_A u \rangle \\
\langle \rangle \parallel_A \langle T, \text{tock} \rangle \bowtie u & = \langle \rangle \\
\langle a \rangle \bowtie t \parallel_A \langle a \rangle \bowtie u & = \langle \langle a \rangle \bowtie \nu \mid \nu \in t \parallel_A u \rangle \\
\langle a \rangle \bowtie t \parallel_A \langle b \rangle \bowtie u & = \langle \rangle \\
\langle a \rangle \bowtie t \parallel_A \langle d \rangle \bowtie u & = \langle \langle d \rangle \bowtie \nu \mid \nu \in \langle a \rangle \bowtie t \parallel_A u \rangle \\
\langle a \rangle \bowtie t \parallel_A \langle T, \text{tock} \rangle \bowtie u & = \langle \rangle \\
\langle c \rangle \bowtie t \parallel_A \langle d \rangle \bowtie u & = \langle \langle c \rangle \bowtie \nu \mid \nu \in t \parallel_A \langle d \rangle \bowtie u \rangle \cup \\
& \cup \langle \langle d \rangle \bowtie \nu \mid \nu \in \langle c \rangle \bowtie t \parallel_A u \rangle \\
\langle c \rangle \bowtie t \parallel_A \langle T, \text{tock} \rangle \bowtie u & = \langle \langle c \rangle \bowtie \nu \mid \nu \in t \parallel_A \langle T, \text{tock} \rangle \bowtie u \rangle \\
\langle S, \text{tock} \rangle \bowtie t \parallel_A \langle T, \text{tock} \rangle \bowtie u & = \langle \langle U, \text{tock} \rangle \bowtie \nu \mid U = S \cap_A T \land \nu \in t \parallel_A u \rangle 
\end{align*}
\]
Note that traces must always agree on tock events: tock is implicitly assumed to be in \( A \). Further, the traces formed by merging a pair of timed testing traces are maximal: none is a prefix of any other.

**Lemma 1.1.7 (Minimality of trace composition)**

\[
\exists t \parallel_A u \Rightarrow \exists s, w \bullet ((s < t \lor w < u) \land r \parallel_A w)
\]

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

### 1.1.10 Hiding

The hiding operator provides a way to abstract processes by internalising some events, thus making them unobservable by the environment. There are two sources of divergence arising from hiding. First, a process \( P \setminus A \) may diverge because \( P \) itself diverges. Second, it may be that hiding an unbounded sequence of events causes the hiding process to diverge. This is captured by the precondition \( \neg (P_f \setminus \text{tt} \setminus A) \). An assumption of maximal progress requires that no time may elapse whilst hidden events are on offer: hidden events happen as soon as they become available. Once more, the definition is given using semantic functions:

**Definition 1.1.24 (Hiding)**

\[
P \setminus_{rt} A = \exists u \bullet A \subseteq \text{rejects}(u) \land u/A = tt' \land P[rt \setminus u/rt']
\]

**Lemma 1.1.8 (Precondition/Postcondition form)**

\[
P \setminus_{rt} A = RT \left( \forall u \bullet A \subseteq \text{rejects}(u) \land u/A = tt' \Rightarrow \neg P_f[rt \setminus u/rt'] \right)
\]

The assumption of maximal progress is modelled by considering only the \( A \)-urgent traces of \( P \): the traces where every event in \( A \) is refused before a refusal experiment. These traces represent states in which no further internal progress is possible using events from the set \( A \): all possible occurrences of those events must already have happened internally. In that case, the process will reject all of \( A \) in every refusal experiment. This is guaranteed by the expression \( A \subseteq \text{rejects}(u) \).

The semantic hiding operator is then defined inductively:

**Definition 1.1.25 (Trace hiding)**

\[
\begin{align*}
\langle \rangle \setminus A &= \langle \rangle \\
(S, \text{tock}) \setminus A &= (S \setminus A, \text{tock}) \setminus (\text{tt} \setminus A) \\
(a \setminus \text{tt}) \setminus A &= tt \setminus A \\
(b \setminus \text{tt}) \setminus A &= b \setminus (tt \setminus A)
\end{align*}
\]
1.1.11 Timeout

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

The timeout process comes in two parts. In one case, either less than $n$ time units have passed, or a visible event occurred in the first $n$ time units. The process will then behave like $P$. In the other case, $n$ time units have passed without a visible event. In this case, the trace can be divided up into two $RT$-healthy portions, the first of which satisfies $tsocks(u) = tock^n$ and the postcondition of $P$ (without changing the value of $v$ or deadlocking, and the second of which satisfies the precondition of $Q$.

Definition 1.1.26 (Timeout)

\[
P \Rightarrow^n_{\text{RT}} Q = \begin{pmatrix} \exists r_0, v_0. \begin{array}{l} rt \leq r_0 \\ \land tock^n = tocks(r_0 - rt) \\ \land P[r_0, v_0, false / rt', v', wait'] \\ \land Q_f[r_0, v_0, true / rt, v, ok] \\ \triangleleft tock^n \leq tocks(tt') \triangleright \\ P_f \end{array} \end{pmatrix}
\]

Lemma 1.1.9 (Precondition/Postcondition form)

\[
P \Rightarrow^n_{\text{RT}} Q = \begin{pmatrix} \forall r_0, v_0. \begin{array}{l} (rt \leq r_0 \land tock^n = tock(r_0 - rt)) \\ \Rightarrow \triangleright (P_f[r_0, v_0, false / rt', v', wait'] \land Q_f[r_0, v_0, true / rt, v, ok]) \\ \triangleleft tock^n \leq tocks(tt') \triangleright \\ P_f \end{array} \end{pmatrix}
\]

1.1.12 Recursion

We say that a function $F$ is $RT$-healthy if it maps $RT$-healthy programs to $RT$-healthy programs. Equivalently, there exist unique functions $F_1$ and $F_2$ such that $F(\text{RT}(A \vdash B)) = \text{RT}(F_1(A, B) \vdash F_2(A, B))$. 

15
For $RT$-healthy $F$, then the least fixed point of $F$ is just the least fixed point of $F$ made $RT$-healthy.

Definition 1.1.27

$$(\mu X \cdot F(X)) = RT(\mu X \cdot F(X))$$

where $\mu F = \cap \{ P \mid F(P) \subseteq P \}$

1.2 Lowe & Ouaknine’s Axioms

Our semantic domain is inspired by that of Lowe & Ouaknine. They start with five axioms, some of which we can consider as theorems of our definitions. One, Zeno freedom, is equivalent to RT6.

1.2.1 Well Foundedness

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

Definition 1.2.1 (T1: Well foundedness)

$T1(P) = P[\emptyset/\tt0]$  

Theorem 1.2.1 (Well foundedness) Every CML operator preserves $T1$-healthiness.

Proof See Appendix. □

1.2.2 Prefix Closure

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

Definition 1.2.2 (T2: Prefix closure)

$T2 \quad [ P \land t \preceq tt' \Rightarrow P[t/\tt0]]$

Theorem 1.2.2 (Prefix closure) Every CML operator preserves $T2$-healthiness.

Proof See Appendix. □
1.2.3 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 1.2.3 (T3: Refusals)**

\[
T_3(P) = P \land (P[rt \triangleright tt' \sim \{A\}/rt']) \land \neg P[rt \triangleright tt' \sim \{a\}/rt'] \Rightarrow P[rt \triangleright tt' \sim (A \cup \{A\})/rt']
\]

**Theorem 1.2.3 (Refusals)** Every CML operator preserves \( T_3 \)-healthiness.

1.2.4 Timelock Freedom

Most processes calways allow time to pass. Assignment and SKIP terminate immediately so there is no opportunity for time events to occur.

**Definition 1.2.4 (T4: Timelock freedom)**

\[
P \Rightarrow P[tt' \sim \emptyset/\triangleright tt']
\]

**Theorem 1.2.4 (Timelock freedom)** Every CML constructive operator preserves \( T_4 \)-healthiness. However, SKIP and assignment are not \( T_4 \)-healthy.

1.2.5 Zeno Freedom

Lowe & Ouaknine’s bounded-speed condition was used to make RT6. It gives a bound \( n \) on the number of events that can be performed in the first \( k \) time units.

**Definition 1.2.5 (T5: Zeno freedom)**

\[
T_5(P) = \forall k. \exists n. \forall tt'. P \Rightarrow (#(tt' \downarrow tock) \leq k \Rightarrow \#(trace(tt')) \leq n)
\]

We say that a recursive process is time-guarded if it cannot recurse without time passing. The Zeno-freedom axiom is satisfied inherently by CML processes made up from CML operators that contain only time-guarded recursions - that is to say that RT6 fixes these processes.

**Theorem 1.2.5 (Prefix closure)** Suppose that \( P \) is a time-guarded process, then for every \( k \) there is an \( n \), such that \( P \) is \( T_5 \)-healthy.

**Proof** See Appendix. □

1.2.6 Time-guardedness

A syntactic check is available to ensure that a CML process is time-guarded. The following is adapted from [?]

---

17
Definition 1.2.6 (time-guarded) If $X$, $Y$ are CML process variables, and $A, B \subseteq \Sigma$, then a CML term is time-guarded for $X$, provided it has been constructed from terms satisfying the following rules:

- $STOP$
- $Y \neq X$
- $a \rightarrow P$, provided $P$ is time-guarded for $X$
- $P \setminus A$
- $\mu Y . F(X)$
- $P \overset{n}{\rightarrow} Q$, provided $n \geq 1$ or $P$ is time-guarded for $X$
- $P \triangleleft Q$, $P \triangledown$, $P \parallel AQ$, provided $P$ and $Q$ are time-guarded for $X$
- $\mu X . P(X)$, provided $P$ is time-guarded for $X$. 