The Abstract Semantics of Tasks and Activity in the Discovery Method

Carlos Alberto Fernández y Fernández

Submitted in Partial Fulfilment of the Requirements for the Degree of

Doctor of Philosophy

The Department of Computer Science

The University of Sheffield, United Kingdom

February 2010

Abstract

Developing software is a complex activity supported by software engineering, a discipline that is still immature in some respects. While the complexity of software has been increasing over the years, the *art* of modelling software is still at an early stage. Software is designed and developed without any guarantees about whether it is going to work as desired. Design is based almost entirely on the experience of experts and, for most production software, formally verifying the design is not a possibility due to commercial pressure to finish the project.

One option for verifying software is to use formal methods to specify the software and use the specification to detect design errors at an early stage. The problem here is that, in many cases, developers are too busy producing and modifying software to create additional artefacts that describe the software formally; or they may have no experience in writing specifications using traditional formal methods. Instead, developers may only be familiar with graphical design notations, such as UML, which are used intuitively to capture aspects of the design, irrespective of whether these models are consistent, complete, or have an unambiguous semantics.

It is our firm belief that the gap between semi-formal visual modelling notations and precise formal specifications must be bridged. What is needed is a smaller and simpler object-oriented notation than UML, that could be easier to learn and use in an exact and repeatable way, to act as the basis for a completely formal treatment. This thesis proposes an abstract syntax and denotational semantics for the hierarchical decomposition of tasks and workflows, specifically in the Task Model of the Discovery Method. The Discovery Method is an approach to systems analysis and design, which adopts a restricted UML profile, focusing on minimalism and consistency.

Task Flow and Task Structure diagrams are mapped onto the terms of an abstract task algebra, a quotient algebra defined using an abstract syntax and axioms. The task algebra is then mapped onto a denotational semantics, consisting of sets of traces of events representing atomic tasks. The axioms of the task algebra support reasoning about the equivalence of Task Structures and Task Flows. This is proven in the denotational semantics, which maps everything onto sets of traces. In particular, the behaviour of *empty*, *abort* and *return* events is modelled correctly, in the presence of iteration and concurrency. A proof of concept is developed for model checking, implemented in the Haskell programming language, within which equivalence and temporal logic properties (in LTL and CTL) are checked.

As a result, any Task Model developed in the Discovery Method may be converted to equivalent expressions in the task algebra, with a corresponding unambiguous denotational semantics. Software developers need only use the precise, minimal Task Structure and Task Flow diagram notations to develop a hierarchical Task Model with a completely formal interpretation. After conversion to algebraic form, the designs are amenable to automatic model checking of equivalence and temporal logic properties. Such a facility supports the early validation of a design, establishing whether it is consistent and complete.

Contents

CHAPTER 1: INTRODUCTION....1

1.1 BACKGROUND AND MOTIVATION....1
1.2 GOAL OF THIS RESEARCH....3
1.3 OBJECTIVES....4
1.4 HYPOTHESIS....4
1.5 THESIS STRUCTURE....5
1.6 SUMMARY....6

CHAPTER 2: FORMAL METHODS AND MODELLING TOOLS....7

2.1 INTRODUCTION....7
2.2 TOOL SUPPORT FOR Z, ALLOY AND OCL....8
2.3 Z....9
2.3.1 An example: the birthday book....10
2.3.2 Schema calculus....12
2.3.3 Z tools....13
2.4 ALLOY....14
2.4.1 Signatures....15
2.4.2 Declaration area....16
2.4.3 Formulas....17
2.4.4 Executing an analysis....19
2.4.5 Metamodel....21
2.4.6 Summary....21
2.5 OCL....22
2.5.1 Syntax overview....22
2.5.2 BirthdayBook example....23
2.5.3 Summary....26
2.6 PROCESS ALGEBRA....26
2.6.1 ACP....27
2.6.2 CCS....28
2.6.3 CSP....28
2.7 SUMMARY....28

CHAPTER 3: OBJECT ORIENTED METHODOLOGIES AND THE DISCOVERY METHOD 30

3.1 INTRODUCTION....30
3.2 A BRIEF OBJECT-ORIENTED HISTORY....31
3.3 UML PROBLEMS....32
3.4 DISCOVERY METHOD....36
3.4.1 Business Modelling....38
3.4.2 Object Modelling....39
3.4.3 System Modelling....42
3.4.4 Software Modelling....44
3.5 SUMMARY....44

CHAPTER 4: THE INFORMAL SEMANTICS FOR THE TASK MODELS....45

4.1 THE INFORMAL SEMANTICS FOR THE TASK DIAGRAMS....45
4.1.1 Task Structure Diagram....45
4.1.2 Task Flow Diagram....47
4.2 THE ALLOY APPROACH....49
4.2.1 Methodology....49
4.2.2 Abstract syntax....49

4.2.3 Checking visual models with Alloy *51*
4.2.4 Evaluating Alloy *55*
4.2.5 Conclusions on the Alloy approach *56*
4.3 FROM THE TASK FLOW DIAGRAM TO THE TASK ALGEBRA 58
4.3.1 Sequence of tasks *58*
4.3.2 Selection *59*
4.3.3 Repetition *60*
4.3.4 Parallel composition *61*
4.3.5 The eating routine example *61*
4.4 SUMMARY 63

CHAPTER 5: AN ABSTRACT SYNTAX REPRESENTATION FOR THE TASK FLOW MODEL 64

5.1 INTRODUCTION 64
5.2 THE ABSTRACT SYNTAX 64
5.3 TASK MODEL CONSTRUCTIONS 66
5.3.1 Simple task *66*
5.3.2 Empty activity *67*
5.3.3 Finished activity *67*
5.3.4 Sequential composition *67*
5.3.5 Selection *68*
5.3.6 Parallel composition *69*
5.3.7 Repetition *72*
5.3.8 Encapsulation *73*
5.4 SUMMARY 74

CHAPTER 6: THE SEMANTICS OF TASKS 75

6.1 INTRODUCTION TO TRACE SEMANTICS 75
6.2 TRACE SEMANTICS FOR TASKS 76
6.3 THE TRACE DOMAIN 77
6.3.1 The Trace Alphabet *77*
6.3.2 Construction of Traces *78*
6.4 SEMANTIC FUNCTIONS OVER THE TRACE DOMAIN 79
6.4.1 Concatenation of Traces *79*
6.4.2 Concatenated Product of Trace Sets *83*
6.4.3 Interleaving of Traces *85*
6.4.4 Distributed Interleaving of Trace Sets *91*
6.4.5 Unpacking of Trace Sets *93*
6.5 INTERPRETING TASK ALGEBRA IN THE TRACE DOMAIN 94
6.5.1 Tracing Basic Elements *95*
6.5.2 Tracing a Sequence of Activity *95*
6.5.3 Tracing a Selection of Activity *99*
6.5.4 Tracing a Parallel Composition of Activity *101*
6.5.5 Tracing a Repetition of Activity *104*
6.5.6 Tracing the Unpacking of Activity *113*
6.6 SUMMARY 118

CHAPTER 7: SOUNDNESS FOR THE SEMANTICS OF TASKS 120

7.1 INTRODUCTION 120
7.2 SOUNDNESS 120
7.2.1 Sequential composition *121*
7.2.2 Selection *123*
7.2.3 Parallel composition *124*
7.2.4 Repetition *126*
7.2.5 Encapsulation *128*
7.3 CONGRUENCE 131
7.3.1 Showing congruence for basic operators in the associative sequence axiom *131*
7.4 SUMMARY 133

CHAPTER 8: THE TASK ALGEBRA IMPLEMENTATION 134

8.1 INTRODUCTION 134
8.2 TASK ALGEBRA IMPLEMENTATION 134
8.3 AN ELECTRONIC JOURNAL 138
8.3.1 Task Flow analysis 138
8.4 OPERATIONS ON TRACES 148
8.4.1 Set operations on traces 148
8.4.2 Model-checking with LTL 149
8.4.3 Model-checking with CTL 150
8.4.4 An implementation of model-checking with LTL 151
8.4.5 An implementation of model-checking with CTL 153
8.5 TESTS OF THE IMPLEMENTATION 159
8.6 SUMMARY 160

CHAPTER 9: CONCLUSIONS 161

9.1 RESULTS 161
9.2 EVALUATION 162
9.3 FUTURE WORK 163

REFERENCES 164

APPENDIX A: PROVING BASIC PROPERTIES 175

A.1 ASSOCIATIVITY OF ⊗ 175
A.2 DISTRIBUTION OF ⊗ OVER UNION 181
A.3 IDENTITY FOR ⊗ 182
A.4 ASSOCIATIVITY OF // 182
A.5 COMMUTATIVITY OF // 187
A.6 DISTRIBUTION OF // OVER UNION 190
A.7 IDENTITY FOR // 191
A.8 DISTRIBUTION OF UNPACK OVER UNION 192

APPENDIX B: CONGRUENCE FOR THE SEMANTICS OF TASKS 193

B.1 INTRODUCTION 193
B.2 SHOWING CONGRUENCE FOR SEQUENTIAL COMPOSITION 193
B.2.1 Showing congruence for basic operators in the associative sequence axiom 193
B.2.2 Showing congruence for basic operators in the right distributivity of sequence over selection axiom 195
B.2.3 Showing congruence for basic operators in the empty sequence axiom 199
B.2.4 Showing congruence for basic operators in the fail on sequence 202
B.2.5 Showing congruence for basic operators in the succeed on sequence axiom 204
B.3 SHOWING CONGRUENCE FOR SELECTION 207
B.3.1 Showing congruence for basic operators in the associative selection axiom 207
B.3.2 Showing congruence for basic operators in the commutative selection axiom 216
B.3.3 Showing congruence for basic operators in the idempotent selection axiom 218
B.4 SHOWING CONGRUENCE FOR PARALLEL COMPOSITION 219
B.4.1 Showing congruence for basic operators in the associative parallel composition axiom 220
B.4.2 Showing congruence for basic operators in the commutative parallel composition axiom 222
B.4.3 Showing congruence for basic operators in the right distributivity of concurrency over selection axiom 224
B.4.4 Showing congruence for basic operators in the instant synchronisation axiom 227
B.4.5 Showing congruence for basic operators in the fail in parallel composition axiom 229
B.4.6 Showing congruence for basic operators in the succeed in parallel composition axiom 232
B.5 SHOWING CONGRUENCE FOR REPETITION 235
B.5.1 Showing congruence for basic operators in the unrolling one cycle of until-loop repetition axiom 235
B.5.2 Showing congruence for basic operators in the unrolling one cycle of while-loop repetition axiom 239
B.6 SHOWING CONGRUENCE FOR ENCAPSULATION 243

B.6.1 Showing congruence for basic operators in the vacuous subtask axiom 243
B.6.2 Showing congruence for basic operators in the coincident exit axiom 246
B.6.3 Showing congruence for basic operators in the vacuous selection axiom 251
B.7 SUMMARY 261

APPENDIX C: SOURCE CODE 262

C.1 TASK ALGEBRA 262
C.2 LTL 269
C.3 CTL 272

List of Figures

Figure 2.1 Instance generated by the execution of the *BusyDay* predicate 20

Figure 2.2. Counterexample generated by the execution of the *DelIsUndo* assertion. 20

Figure 2.3. Metamodel for BirthdayBook..... 21

Figure 2.4. A possible representation of BirthdayBook 23

Figure 2.5. BirthdayBook modelled in ArgoUML 24

Figure 2.6. BirthdayBook modelled in USE..... 24

Figure 3.1 Thread, activation and stack-frame semantics focus bar [112] 33

Figure 3.2 Procedural and non-procedural message flow..... 34

Figure 3.3 Mealy, Moore and UML state machines 34

Figure 3.4 Equivalent UML models for concurrent substate machines [117]..... 35

Figure 3.5 Association and navigation in UML class diagram..... 36

Figure 3.6 Aggregation and generalisation for Data and Task Structure Diagrams.... 37

Figure 3.7 Initial and final states are real states..... 38

Figure 3.8 Notation for the State Diagram in the Discovery Method..... 40

Figure 3.9 Notation for Data Diagram in the Discovery Method 41

Figure 3.10 Notation for Collaboration Diagram in the Discovery Method..... 42

Figure 3.11 Notation for the Collaboration Diagram in the Discovery Method..... 43

Figure 4.1 Basic elements of Task Structure Diagrams..... 46

Figure 4.2 Structural relationships in the Task Structure Diagram 46

Figure 4.3 Elements of the Task Flow Diagram 47

Figure 4.4 Example showing parallel tasks (Modified from [129]) 48

Figure 4.5 General structure of the abstract syntax 50

Figure 4.6 Task Structure Diagram elements 51

Figure 4.7 Circulation Task Structure Diagram 52

Figure 4.8 Encoding the Circulation Task Structure Diagram 52

Figure 4.9 Loan Transaction Task Structure Diagram 52

Figure 4.10 Encoding the Loan Transaction Task Structure diagram 53

Figure 4.11 Encoding the Task Structure model 53

Figure 4.12 Empty predicate and exact scope specified for the run command 54

Figure 4.13 Solution generated by Alloy 55

Figure 4.14 Two diagrams creating an inconsistent Data Model 55

Figure 4.15 Abstract syntax metamodel 57

Figure 4.16 Sequence of tasks in the Task Flow Model 58

Figure 4.17 Selection in the Task Flow Diagram 59

Figure 4.18 Binary selection in the Task Flow Diagram 60

Figure 4.19 Repetition in the Task Flow Model 61

Figure 4.20 Parallel composition in the Task Flow Diagram 61

Figure 4.21 Task Flow Diagram showing the process of doing dinner 62

Figure 4.22 Until-loop repetition in the Task Flow Diagram 63

Figure 5.1. State transition diagram for expressions a;(b+c) and (a;b)+(a;c) 68

Figure 8.1. Structure of the Task Algebra implementation 135

Figure 8.2. Reader Task Flow Diagram 139

Figure 8.3. Author Task Flow Diagram 141

Figure 8.4. Login Task Flow Diagram 141

Figure 8.5. Reviewer Task Flow Diagram 143

Figure 8.6. Editor Task Flow Diagram 146

Figure 8.7. Tree representation of traces from diagram in Figure 8.4 155

Figure 8.8 A partial tree representation of traces from diagram in Figure 8.5 158

Figure 8.9 A partial tree representation of traces from an extract of the diagram in Figure 8.5 159

List of Tables

Table 2.1 Classification of formal methods 8

Table 2.2 Comparison of Z tools 13

Table 2.3. Some binary relations expressed in Alloy 17

Table 8.1 Comparison between original Task Algebra syntax and the Haskell implementation 135

Chapter 1:
Introduction

This chapter introduces the motivation behind the thesis, which is that software engineering notations need formal semantics. Section 1.1 mentions some relevant work related to the goal of formalising parts of the UML notations. Section 1.2 describes the overall goal of this research, which is to provide a complete formal basis for a small task-based notation. Subsequently, the individual objectives of the research, which include developing a formal semantics and model-checking tools, are explained in section 1.3. Finally, an explanation of the structure of the thesis is provided in section 1.4.

1.1 Background and motivation

Software Engineering is still a young discipline and, for some people, cannot yet be considered a proper engineering discipline, because of the toleration of informal and unregulated software development practices. This situation is probably also true in other professional areas that have to deal with people. After all, people bring uncertainty. Nevertheless, the problem is that this uncertainty is also brought to areas that should be more formal and precise. As one would expect in any new discipline, the processes, methods and tools for software development have been slowly but incrementally improving since the late 1970s. Unfortunately, much of what has been presented by way of "software design methods" has been anecdotal, based on the intuitions of what practitioners hoped might work at the time; and only recently has the field of empirical software engineering started to establish a proper evidential basis for comparing different approaches. While this is normal for an area evolving together with the techniques and technology, it should be expected that the more established parts of the discipline should develop a more formal justification.

One area to which this most clearly applies is the area of software engineering tools, which are used to create software designs and from which skeleton code may sometimes be generated. These tools should ideally be based on a formal model, which can guarantee mathematically that the tools can be trusted. Among the tool-supported techniques used by software engineers, visual modelling has become more important for medium and large software projects.

Visual modelling is the modelling of a computer program or larger software system using graphical notations to develop a model, expressed as one or more diagrams.

The model is intended to capture the essentials parts of a system [1] and is used to represent the business processes from a user-centred, or stakeholder's perspective. It contributes to the understanding of the business domain and helps later in the design of the information system.

The Unified Modeling Language (UML) is at present the standard visual modelling notation. At the time of writing, it provides thirteen different diagrams that can be used to represent a software system from different aspects and perspectives [2, 3]. The Unified Modeling Language (UML) [2] is an eclectic set of notations for modelling object-oriented designs. Under the supervision of the Object Management Group (OMG) since 1997, the notation set has grown larger and complex [3], to accommodate the concerns of different stakeholders in business and industry. This has led to some criticisms regarding the open-ended semantics and the lack of direction given in modelling [6-8].

Problems with UML diagrams creating ambiguous representations are mentioned in Chapter 3, although these can be summarized as legal UML diagrams having an unclear meaning, even if they are considered valid according to [4]. Various attempts to formalise parts of UML include the work of the Precise UML group (pUML) [5], which aims to clarify the semantics of UML and create tools to support the rigorous analysis of UML models. Jointly with IBM, pUML submitted a Meta-Modelling Framework (MMF) [6] to the OMG as an alternative to the original UML metamodel. Out of this work came the desire to create an Unambiguous UML, an idea partly inspired by the Catalysis method [7]. The Unambiguous UML (2U) Consortium [8], which grew out of pUML, submitted a full proposal for UML2.0 based on a set of architectural principles.

Some of the work on formalising UML has proposed the use of formal languages such as Z, which was used by Bruel and France [5] when they presented a transformation from UML class diagrams to a Z specification. Kim and Carrington [9] presented a formal mapping transforming UML class diagrams to a specification in Object-Z.

There are also different proposals to formalise UML using the Alloy formal language [10, 11]. Naumenko proposes in [12] an alternative metamodel for UML inspired by RM-ODP [13]. Bordbar and Anastasakis [14] propose a tool called UML2Alloy, where a model is transformed from the UML metamodel to the Alloy metamodel. In [15] Zito and Dingel model the UML 2 package merge operation with Alloy. Also, there is a language called Aaree [16] that supports some object oriented and imperative constructs and has a textual representation that can be translated into Alloy to be analysed.

There is also related work on the development of model checkers and tools for UML. However, the use of model checking to verify object-oriented models is still immature and the integration with UML tools has until now been slow [17].

Some examples include the Hugo tool, which compiles UML state machines into a format processed by the PROMELA model checker [17]. The USE tool (UML-based Specification Environment) [18] allows UML diagrams to be annotated with constraints written in OCL (the Object Constraint Language [19, 20]), after which the validity of models may be checked, using predicates also written in OCL. The tool verifies model instances against explicit predicates and implicitly against the

invariants defined in the model. Shen et al. [21] have proposed a toolset for static and dynamic model checking of UML that, using Abstract State Machines, validates the model with respect to the semantics of UML. ASM specifications of class diagrams and object diagrams are checked. UMC (UML on the fly Model Checker) is a tool designed by Gnesi et al. [22] that applies model checking to UML state machines. Störrle [23] describes a denotational semantics for the Activity Diagrams of UML 2, covering basic control flow and data flow using colored Petri-nets. XMF Mosaic is a tool developed by Xactium [24], which supplies the full capabilities of executable metamodelling for constructing and executing new language definitions. In addition, the tool has a collection of plugins to support the construction of UML-like class models, creating instances of these models and checking them against constraints. Other plugins support the description of mappings between models, and an execution environment for the operation of user-defined languages and tools as stand alone images.

In any case, and in spite of the contributions made by the research mentioned above, it is likely that UML will continue having many of the original problems that we describe later in more detail, due in part because it needs to provide backward-compatibility with its older characteristics, and due to the problem of its size. The position defended here is that what is needed is a smaller and simpler object-oriented notation, than can be easier to learn and use in an exact and repeatable way. This notation should be supported by a formal language in order to represent its semantics in a precise way, such that models could be verified against each other and the specification of a system could be demonstrably consistent and complete. One possible candidate for such a notation is the restricted UML profile adopted by the "Discovery Method" [25, 26], which strives for minimalism and consistency.

1.2 Goal of this research

A requisite for developing model checkers is the ability to encode model diagrams in a suitable abstract syntax, and from this to develop an abstract semantics [27]. Initially, after analysing the graphical notation deployed in the Discovery Method [25, 26] an approach to representing formally the whole notation was tried using the Alloy language [15, 16], but already at the level of the abstract syntax this proved to be difficult to model, even when the abstract syntax of the diagrams could be verified (see Chapter 4). Later, a different approach was defined limiting the scope of the formalisation to the *Task Model*[1], and the goal was established as follows:

To provide an abstract syntax and denotational semantics for tasks and activities in the Task Model of the Discovery Method.

Limiting the scope to the definition of tasks and activities has some advantages, such as of having a simple representation of the Task Flow Diagram. In addition, because Simons [28] defined "Task flow inversion", a direct transformation between the *Task Flow Diagram* and the *State Diagram* in the Discovery Method, it should be possible to represent State Diagrams from the Discovery Method using the same semantics as

[1] For this thesis, the term Task Model is used to include Task Structure and Task Flow models. The Discovery Method also includes Narratives as part of the Task Model, but this was not considered in this work so far, even when there is a correspondence between Narratives and these other diagrams.

that defined for the *Task Flow Diagram*. Finally, the *Task Flow Diagram* is closely related to the *Task Structure Diagram*, imposing already within this more limited project some useful restrictions on the correct construction of related diagrams, which could be tested by the formalisations proposed in the rest of this work.

1.3 Objectives

For the accomplishment of the objective of the project, four objectives were defined to specify the scope of the research. The objectives of this research were as follows:

- The definition of the abstract syntax should depict accurately the notion of tasks and activities in the various task-modelling diagrams.
- The denotational semantics for the abstract syntax and its associated task algebra should be defined in term of traces.
- Soundness and congruence of the proposed abstract syntax and the semantics should be proved.
- In order to test the feasibility of the formal representation, an implementation of the algebra should be built.

So far, the syntax of Task Flow diagrams has not been presented formally. Having an abstract syntax is desirable, both because it offers a succinct textual representation of the diagrams, and also because it serves as the basis for a formal definition of diagram well-formedness. A task algebra will be constructed from the abstract syntax, by introducing axioms over syntactic expressions that fall into the same equivalence class (a quotient algebra). After this, a denotational semantics will be presented, in which the meaning of all possible sequential and concurrent execution paths will be given as sets of traces. The semantics will be formalised by a proof of soundness and congruence. Soundness is a property which holds, when syntactic expressions that are judged equivalent by the axioms are also trace-equivalent in the semantics. Finally, an executable model of the task algebra is developed in the functional programming language Haskell, in order to test the feasibility of the proposal.

Additionally an implementation of model-checking using LTL and CTL expressions was also developed in order to take advantage of the task algebra implementation and to show possible practical uses of the tools. A case study is presented in Chapter 8 depicting the use of the task algebra, and checking for LTL and CTL properties using software written for this purpose**.**

1.4 Hypothesis

This research was originally motivated by the fact that software engineering notations are often vague, in the sense that they are incomplete, or ambiguous and so are open to different interpretations by software engineers. The hypothesis that is being investigated by the programme of research described above is the following:

It is possible to give an unambiguous, formal interpretation to the diagrammatic representation of tasks and activities in the Task Model of the Discovery Method, such that:

- *individual diagrams have a single, unambiguous procedural meaning and so may be translated into workflow-based procedural programs;*
- *a design may be broken down according to different high-level design choices, yet yield systems of diagrams that have equivalent meaning;*
- *questions may be formulated and tested about the validity of logical properties within a system of diagrams, using temporal logic.*

The benefit of providing software engineering notations with a fully formal abstract syntax and semantics, whose properties are known and provable, is that software engineers may then rely on the notations directly, with full confidence, without having to understand the underlying formal semantics. While the scope of this research only covers the Task Model in the Discovery Method, this notation is used in the early analysis phase to capture and represent the customer's requirements formally, which is the necessary foundation for developing a sound software system. We therefore anticipate that the work undertaken here will be of practical use for software engineers using the Discovery Method to develop software systems.

1.5 Thesis structure

This thesis consists of nine chapters, beginning with the present chapter 1, which introduces the background and presents the main objective and goals of the research. Chapter 2 will give an overview of formal methods and will include some small examples, particularly in Z, OCL and Alloy, using these notations for formal software modelling. This chapter will also provide an overview of process algebra. Subsequently, chapter 3 will offer an overview of object-oriented methodologies and UML, including a brief history of the object-oriented paradigm. Additionally, It presents an introduction to the Discovery Method, explaining every phase of the method generally. Afterwards, chapter 4 explains in detail the *Task Model* in the Discovery Method, explaining the *Task Structure* and the *Task Flow Diagram*. In addition, an experiment is conducted to check a large fragment of the Discovery metamodel in Alloy. Finally, this chapter introduces the task algebra and the relationship between the algebra and the *Task Model*. Chapter 5 depicts formally the abstract syntax representation for the *Task Flow Model*. It defines the syntax and a set of axioms constraining the definition. Subsequently, chapter 6 describes the semantics for the abstract syntax as a set of traces. Chapter 7 proves the soundness of the axioms for the abstract syntax presented in chapter 4. Some congruence properties are demonstrated also in this chapter. Chapter 8 shows an implementation of the proposed algebra in the Haskell language. This chapter also includes a case of study where the task algebra is used and an example implementation of operations applied over the trace semantics using set operations and temporal logic operations such as LTL and CTL expressions. Chapter 9 presents the conclusions of this work.

Moreover, three appendices are included in this work. Appendix A shows the proof of basic properties used to specify the semantics. Appendix B provides demonstrations of all the required congruence properties. Finally, Appendix C

includes the Haskell source code for the tool implementations, which include: the task algebra simulation, LTL model checking and CTL model checking tools.

1.6 Summary

Among current trends in software engineering, there are many attemps to formalise parts of the UML notations. This research found that practicality was lacking as an important aim in most of the proposals. This situation makes it difficult for the modeller to learn how to understand the formal representations behind the models, in order to gain any practical benefit from it. While a complete understanding of the formal semantics behind software models is desirable, here we believe the formal semantics should be relegated to the background; and visual modelling tools, being based on the formal semantics, should be used to create precise models of software.

This chapter introduced the thesis, describing the background and motivation for this project. Subsequently, the objective and goals of the research were explained. Finally, an explanation of the structure of the thesis was provided. The next chapter offers an overview of formal methods that were considered relevant to this research.

Chapter 2:
Formal Methods and Modelling Tools

The previous chapter presented an introduction to this thesis. In this chapter, an overview of formal methods is described with an emphasis on model based languages (such as Z, OCL and Alloy), and process algebra languages. The next chapter will discuss object-oriented methods and, in particular, the Discovery Method.

2.1 Introduction

Formal calculi for software construction have seen an increase in use over the last 25 years [29], but this form of representation has been used mainly in academia. Although there are some accounts of their use in the industry (basically in critical systems), the majority of the "real world" has for years been using visual modelling as a kind of "semi-formal" representation of software.

A method is considered formal if it has well-defined mathematical basis. Formal methods provide a syntactic domain (i.e., the notation or set of symbols of the method), a semantic domain (like its universe of objects), and a set of precise rules defining how an object can satisfy a specification [30]. In addition, a specification is a set of sentences built using the notation of the syntactic domain and it represents a subset of the semantic domain.

Spivey says that formal methods are based on mathematical notations and that *"they describe what the system must do without saying how it is to be done"* [31], which applies to the non-constructive approach only. Mathematical notations commonly have three characteristics:

- Conciseness. They represent complex facts of a system in a brief space.
- Precision. The model can specify exactly everything that is intended.
- Unambiguity. The interpretation of the specification has to be the same if a standard and well-understood language is used.

Essentially, a formal method can be applied to support the development of software and hardware. Bogdanov et al. [29] make a classification of these, summarized in Table 2.1.

Table 2.1 Classification of formal methods

Category	Description	Examples
Model-based languages	Set theory and function spaces are used to build models of system operations and of the system data that is modified by the operations. Constraints are expressed in first-order logic.	Z, VDM, B, Alloy, JML[32], lambda calculus
Finite state-based languages	Systems are modelled as finite state automata, with the operations styled as transitions from one state to the next. The states are either high-level abstractions over data, or control states.	Z, FSMs, SDL, Statecharts, X-machines
Process algebra languages	Systems are modelled as collections of independently-executing processes, which synchronise to exchange data. Each process is individually modelled as a finite state automaton.	CSP, CCS, ACP, LOTOS
Algebraic languages	Systems are modelled as collections of algebras, where an algebra consists of sorts (sets), operation signatures and axioms, describing the behaviour of an abstract data type.	OBJ, Larch[10]

Although there are many formal methods, Z, Alloy and OCL have recently received quite a lot of attention [5-10] and are most relevant to the work reported in this thesis, so these will be the focus later in this chapter. Many of these have a long and distinguished pedigree. The Z language has been in use for a long time in support of a formal method for software specification. It has been defined in an ISO standard since 2002 [33]. Alloy was a language emerging at the time of writing, which claimed to be based originally on Z, but which was easier to use. Alloy is also claimed to be syntactically closer to OCL than to Z [34]. Finally, OCL was chose because of its relationship with the UML, where the attempt was to provide UML diagrams with complementary text definitions to make precise UML models. These three languages seemed initially to be a good choice to represent the abstract syntax and semantics established in the objectives of this research. Alloy was investigated first, because offered a simple language with a better tool support than Z and OCL.

The next section presents an overview of the kind of tool support available for some of the formal methods introduced above.

2.2 Tool Support for Z, Alloy and OCL

Many formal methods have been supported with one or more tools. These, based on the characteristics of each formal method, can be broadly categorized either as theorem provers, or model checkers.

A theorem prover is a tool which takes a set of axioms and it tries to prove if the whole set (extended with some theorem) is valid. Some known theorem provers are HOL, Isabelle, and PVS. A problem mentioned by Simons in [35] is that if there are too many axioms in a theory, the theorem prover will try to explore redundant solutions, leading to an explosion in the search-space. Theorem provers must frequently be guided towards solutions, through interaction with the user, who identifies the most appropriate intermediate lemmas, or selects the appropriate proof tactics.

On the other hand, a model checker simulates a model of the system by exploration, analysing several state machines and these are compared automatically [36]. In this area can be seen SMV, SPIN, and FDR among others. Temporal logic has been applied to model checking from the early 1980s by Clarke and Emerson [37]. Model checking has advantages over theorem provers, the most important being that the procedure is completely automatic [38, 39], although it has the problem of state explosion too. A model checker uses a model described by the user to discover whether hypotheses are valid in the model. If the hypotheses are invalid, the model checker can build counterexamples and display the execution traces that lead to these.

There is a slightly different, but related, category of tools such as the Alloy analyser which is described as a "model finder". Alloy works by finding models that form counterexamples to assertions made by the user: *"Its engine takes a formula and attempts to find a model of it"* [36]. Paradox by Claessen et al. [40] is another program that implements techniques for finding finite models based on first order logic, whilst model checking is commonly based on temporal logic.

2.3 Z

Z is a formal language based on set theory, function spaces and first order predicate logic [41]. Two aspects of Z are different from classical set theory [42]: first, the sets defined in Z are partitioned into different categories, i.e., they are disjoint (*"the set theory is a typed set theory"*); the second aspect is the concept of the *schema*, where we may define descriptions of objects that can be referenced in the whole model. The schema representation can be used to describe the state of a system [43].

The Z language helps to describe a system in both its static and dynamic aspects [31]; in the first case Z can represent the states and the invariant relationships that are fixed for all the states of the system; in the dynamic aspect, it can depict the operations, the relationship between inputs and outputs, and the consequential changes of state.

Z has been one of the most popular formal languages to model systems, and for this reason we can find several tools that use it in different levels [44, 45], although most Z tools have historically only been able to check the syntax. There is an international project "Community Z Tools" (CZT) proposed by Andre Martin in 2001 [46] that aims to build *"a set of tools for editing, typechecking and animating formal specifications written in the Z specification language"* [47]. Other languages like OCL or Alloy are inspired by the Z language [7, 10].

2.3.1 An example: the birthday book

It is not the intention to give a complete treatment of Z, but just to include part of the classical example by Spivey [31, 41] in order to identify some of the characteristics of Z and other formal languages.

The *BirthdayBook* is a simple system that records birthdays of people and notifies when some birthday comes about. For this example, two basic sets are needed:

[NAME, DATE]

The above introduces two sets, which are uninterpreted, in the sense that nothing else is known about their elements. The sets are therefore completely abstract. Now, it is possible to define the state schema of *BirthdayBook* as follows:

BirthdayBook

$$known: \mathbb{P}\ NAME$$
$$birthday: NAME \pfun DATE$$
$$known = \mathrm{dom}\ birthday$$

A schema in the Z language consists of two parts, the first area is for declaration of variables and the second is for predicates. In the *BirthdayBook* schema, there are two variables (a set *known* and a function *birthday*) and one predicate. Variables of set-types are used to represent collections, and variables of function-types are used to represent maps, relationships between elements of different sets. The predicate constrains *known* to be the domain of the function *birthday*. This is a state schema, representing the data manipulated by the system.

The same schema style is used to represent both static aspects (data declarations) and dynamic aspects (operation specifications). The operation to add a birthday to the *BirthdayBook* is specified in the following schema:

AddBirthday

$$\Delta BirthdayBook$$
$$name?: NAME$$
$$date?: DATE$$
$$name? \notin known$$
$$birthday' = birthday \cup \{(name? \mapsto date?)\}$$

The first line (Δ*BirthdayBook*) is a Z short-hand for importing all the declarations of the *BirthdayBook* schema into the *AddBirthday* schema. The delta-convention means that two copies of all variables are imported, and by convention the unadorned variables (*known*, *birthday*) denote prior states and the primed variables (*known'*, *birthday'*) denote posterior states. The operation schema therefore makes a change to the state of the *BirthdayBook*.

The second and third declarations define *name* and *date* as inputs to the operation. Inputs are syntactically identified with a question mark at the end of their names.

In the predicate area, *AddBirthday* defines a precondition that we cannot add a name previously registered in the *BirthdayBook*. The precondition must be satisfied for the success of the operation. After that, as a postcondition, the next line declares what happens if the precondition is satisfied: the *birthday* function now includes the new maplet between *name?* and *date?*.

Although logically consistent, it is strictly unnecessary to add the following line to the predicate area of the *AddBirthday* schema, to express that a new name is known:

$known' = known \cup name?$

because this can be derived from the invariant defined in the *BirthdayBook* schema:

$known = \text{dom } birthday$

Consequently, it can be derived [31, 41], that after executing *AddBirthday*:

$known' = \text{dom } birthday'$

In a similar fashion, the *FindBirthday* schema defines an operation to look up the date of a person's birthday in the *BirthdayBook*. The first line (Ξ *BirthdayBook*) is a Z short-hand for importing declarations from *BirthdayBook*, similar to the delta-convention, but with the additional constraint that primed and unprimed variables are pairwise equivalent (an implicit predicate). So, the *FindBirthday* operation schema makes no change to the state variables.

FindBirthday

$\Xi BirthdayBook$
$name?: NAME$
$date!: DATE$

$name? \in known$
$date! = birthday\ (name?)$

The second and third lines define an input *name*? and an output *date*!, where outputs are identified syntactically by the exclamation mark at the end of their name. The predicate area has two declarations. The first is a precondition that the sought name must be in the *known* set. The second is a postcondition asserting that the output *date!* corresponds to the result of the function *birthday* with the argument *name?*. While pre- and postcondition predicates are not explicitly distinguished, it is clear that postconditions are those predicates that refer to output variables, while preconditions refer only to input variables.

Z has a particular semantic interpretation. In classical Z, operations are only well-defined if the preconditions are satisfied; otherwise they are undefined, in the sense that they could yield any arbitrary values. In some variants of Z, such as Object-Z, a

different blocking semantics is adopted, whereby operations are assumed to be blocked, if their preconditions are not satisfied.

2.3.2 Schema calculus

The use of schemas helps us to build more complex specifications using modular construction. For instance, Z supports schema inclusion, where a name of a schema can be used in the declarations of another schema. There is also a set of operators for combining different schemas logically, supporting the modular construction of system specifications [42]:

Disjunction: The schema operator $\vee$ declares two schemas as alternatives. For example:

$A \mathrel{\widehat=} B \vee C$

where A stands for the declarations of B and C joined and their predicates disjoined.

Conjunction: We can combine two schemas with the schema operator $\wedge$. For example:

$A \mathrel{\widehat=} B \wedge C$

where A stands for the declarations of B and C joined and their predicates conjoined.

Negation: $\neg$ applied to a schema keeps the declaration and negates the predicate of the schema. For example:

$\neg A$

where the negation of the schema A implies negation its predicate.

Composition: $\fatsemi$ specifies an operation as a composition of schemas. For example:

$A \fatsemi B$

depicts that A occurs, then B.

Quantification: $\bullet$ Used for schemas when we need to quantify over the elements of a schema. For example:

$Q\ d \bullet S$

where if *Q* is a quantifier, *d* a declaration, and *S* a schema, then the quantified schema is obtained from taking the components that are part of *d* and of *S*, and quantifying with *Q* in the predicate part.

Decoration: ’ is used to describe the effect of an operation. For example:

A, A’

where *A* represents the schema before the operation and *A'* the state of the *A* schema afterwards (whose variables are also considered primed).

The schema calculus is used to construct *robust schemas*, each being the disjunction of a regular schema and a complementary *error schema*, whose preconditions are the negation of the regular schema's preconditions. This ensures that system behaviour is totally defined over all inputs and states; and takes error recovery into account. It is also possible to check that the requirements are consistent, which is normally achieved by showing that the constraints of the schema are satisfiable showing at least an initial state; this is called an initialisation theorem. In addition, preconditions can be used when there is interest in showing that the operations are never applied out of their domain [43].

Preconditions can be used to describe a set of states that can be reached if the operation schemas are properly defined. The preconditions can be simplified using equivalences. Some of this work can be carried out with the support of theorem prover tools.

2.3.3 Z tools

As shown in Table 2.2, it is possible to find Z tools that help in aspects such as writing Z specifications (e.g., FuZZ that supports basically printing and type-checking for Z specifications based on LaTex). Additionally, there exist tools such as CADiZ and Z/EVES which provide visual editing of Z specifications and different levels of analysis. Another kind of tools tries to represent the links between Z and object-oriented notation [45]. This third category is more attractive and is relevant to the research. It can be seen tools like RoZ [48], although RoZ only generates Z specifications and uses Z/EVES in order to realize the consistency checks. Additionally, Sun et al. [44] are working in techniques for XML representation of Z and Object-Z on the web, and its transformations into UML diagrams. They developed XML browsing facilities for Z and Object-Z and proposed some techniques to project object-oriented Z models onto UML diagrams. Finally, Zeta is a tool partly completed which uses Isabelle to perform theorem proving, but the project was abandoned some years ago. CZT proposed by Andre Martin [46, 47]mentioned in section 2.3. The software includes, for example, a Z markup language defined in XML, a library in Java for Z annotated syntax trees, graphical Z editors and a Z animation tool (ZLive) for evaluating expressions, predicates and schemas [47].

Table 2.2 Comparison of Z tools

	GUI	Type-checking	Theorem proving	Integration with UML or other OO notation
FuZZ		X		
CADiZ	X	X	X	
Z/EVES	X	X	X	
RoZ			Using Z/Eves	Using Rose
Zeta	X	X	Using HOL-Z/Isabelle	
CZT	X	X		

2.4 Alloy

Alloy was developed by Daniel Jackson and the Software Design Group at the MIT and a first version of the tool was released in 1997. Alloy is a language inspired initially by Z [49, 50], although it has been changing from the original prototype. In May of 2004 the version 3 of Alloy was released in beta phase, improving some characteristics of the language and the analyser [51, 52]. At the end of 2006, Alloy Analyser 4 was released with a small number of syntactic improvements and using the new SAT-based model finder Kodkod [53].

In Jackson's words [10] *"Alloy is an attempt to combine the best features of Z and the Object Constraint Language of UML in a lightweight notation. It takes UML's emphasis on binary relations, and the expression of constraints with sets of objects formed by 'navigations', but with Z's much simpler semantics."*

The essential idea about Alloy can be summarized as follows: a micro-model with Alloy is built using signatures (i.e., a set of atoms) and formula paragraphs (i.e., predicates, functions, or assertions). Once the model has been compiled, every assertion can be checked with the expectation of finding a counterexample. In other words, the Alloy analyser looks for some instance of the micro-model that could be generated in violation of the assertions. It is for this reason that Jackson says in [10] that it is a refutation approach. If a counterexample is found, this means that the model was not created properly (the model is invalid). If a counterexample is not found, that does not mean that the model is necessarily correct. Alloy cannot prove that a model is correct, since although it performs exhaustive searching, creating the complete state-space of scenarios, it is limited by the number of exemplar instances in the checked system [54]. One model without a counterexample means that Alloy cannot find a counterexample in the scope specified, but there may still exist a counterexample in a larger scope [10]. The effectiveness of this method is based on the *small scope hypothesis* [55] that states that a high proportion of bugs tend to be found in a small scope.

Another analysis option that Alloy can do is checking the consistency of a formula [10]. Using a function or predicate, the Alloy analyser can try to generate an instance of the model in conformity with the constraints [56]. Obviously, this only proves that the model can generate valid instances, but nothing more can be affirmed from this.

Jackson [57] says that – in order to have simple semantics and a concise syntax - Alloy does not distinguish between an atom *a*, a tuple *(a)*, a set *b* that contains only the atom, or a set *b* containing the tuple. Alloy deals entirely with relations; although it consists additionally of atoms, which cannot be directly manipulated by the user.

Alloy is based on first-order logic and it can deal with quantifiers, polymorphism, signatures, and subtyping. The main characteristics of Alloy are as follows [58]:

- Infinite model: A model described in Alloy is considered infinite because, in contrast with traditional model checking, we do not specify the number of components that this model can have.

- Finite scope check: Although the model is infinite, the analysis has to be finite. We have to specify the scope of the model that we want to analyse. The analysis is incomplete for the whole model, but complete for the scope: the analyser always can find (if it exists) a counterexample for the specified scope.

- Automatic analysis: The Alloy analyser can generate examples and counterexamples of our model automatically.

In the next sections, a general overview of the language is offered.

2.4.1 Signatures

Sets, called signatures, can be declared in way similar to how a program is written in an object-oriented language. In Alloy, a set represents a unary relation. The simplest declaration of a signature can be for example:

```
sig MySig {}

sig Name {}

sig Date {}
```

This defines basic sets named *MySig*, *Name*, and *Date*. Inside the curly braces relations can be defined having the signature *MySig* as their domain:

```
sig MySig {

     fieldName1: fieldType1,

     fieldName2: fieldType2

}
```

where a field type can be another signature or a more complex expression and we can refer to signatures still not defined. In this case, we are declaring two binary relations called *fieldName1* and *fieldName2*; where, for example, *fieldName1* is a relation that maps each *MySig* to some *fieldType1*.

Alloy supports two slightly different kinds of signature specialisation:

- A subtype signature;

- A subset signature.

Subtypes in Alloy are basically disjoint subsets of a given type, however two types can still overlap if one is a subtype of the other (directly or indirectly). In contrast, subsets in Alloy are overlapping subsets of a given type. Whereas a subtype may only be created by disjointly specialising a given type or subtype, a subset may be declared of any type, subtype or subset. Both subtypes and subsets may declare additional fields (typically, relations) and transitively inherit all the fields of their parent types. It is not possible to disjointly partition a subset signature.

2.4.1.1 Subtype signatures

A subtype signature can be extended using the keyword *extends*. This instruction creates a subtype of the type of the signature extended, for example:

```
sig Y extends X { }
```

Y is considered a subtype of *X* and a type hierarchy can be created. If the signature *X* is not extending another signature, *X* is a top-type signature, and its type is a top-level type. The top-level types are the roots of the type hierarchy and these are mutually disjoint sets. In the same way, two or more subtype signatures are disjoint unless one extends the other. The main purpose of distinguishing disjoint subtypes from general subsets is in order to increase the efficiency of the model finder.

2.4.1.2 Subset signatures

A subset signature is declared using the keyword *in*, and it is a subset of the signature of its parent or parents. Actually, although in [52] Jackson mentions that a subset is *"a subset of the union of its parents"*, in practice the syntax definition only permits one parent signature.

Subset signatures can be declared as follows:

```
sig X, Y in Z { }
```

This means that both, *X* and *Y* are subsets of *Z* with the following constraints: *Z* may be a subset or a subtype signature, *X* and *Y* are not necessarily disjoint, and the union of *X* and *Y* are not necessarily equal to Z. Finally, it is important to say that subset signatures (e.g., X or Y in this case) cannot be disjointly partitioned using the "extends" keyword.

2.4.2 Declaration area

As mentioned briefly above, inside the curly braces of a signature we can declare fields. A field can denote:

- A unary relation or set: It can be fixed with a multiplicity keyword: *lone*, *one*, *some*, or *set*. If *one* is used, it is equivalent to omit the keyword and this means that the variable will be a scalar or singleton set; *lone* means either a singleton set or the empty set; the *some* keyword denotes a non empty set; and *set* keyword represents zero or more elements in the set. Subsequently, for example:

  ```
  sig BirthdayBook {

      known: set Name

  }
  ```

  Here *known* is a field representing a set of the signature *Name*. For the *BirthdayBook* signature, this means that *known* is a relation with zero or more

range elements from the *Name* type. Other multiplicity keywords *lone*, *one* or *some* may be used in similar contexts, where desired.

- A binary relation: In this case the expression is formed by the arrow operator `->`, and the multiplicity can be constrained in each side of the relation. Table 2.3 presents examples of binary relations.

Table 2.3. Some binary relations expressed in Alloy

Declaration	Semantics
`R: S -> T`	Relation (transition relation)
`R: S -> one T`	total function from S to T
`R: S one -> one T`	Bijection
`R: S -> lone T`	Partial function

Continuing with the example, a field relation named *date* can be added to represent the relation (a partial function) between the names of the *BirthdayBook* and their date of birthday:

```
sig BirthdayBook {

      known: set Name,

      date: known -> lone Date }
```

2.4.3 Formulas

Formulas are specially constructed expressions to be checked in the model. They are built with relational or logical expressions. The essential formulas that can be used in Alloy are:

- Quantified expression: The meaning of this expression is obtained from the use of a quantifier operator.
- Comparison formula: This is an expression constructed using comparison or negation operators.
- Compound formula: It is formed by the combination of smaller formulas using logical operators.
- Declaration formula: This kind of formula is used to put a multiplicity constraint on an expression.

There are more types of formulas that can be used in Alloy such as: negated formula, let formula, and sequence formula.

2.4.3.1 Formula paragraphs

By using formulas, constraints can be added to the model, organised around formula paragraphs. There are four types of formula paragraphs.

- *fact*: To indicate that a property is maintained, the formula is declared as a *fact* [32]. In other words, a fact must be true for any instance in the model. It consists of an optional name and a set of formulas forming a constraint.

  Example:

  ```
  fact OptionalName {
       all a: A | a in (B+C)
  }
  ```

- *pred*: A predicate (*pred*) defines a property without imposing it as a permanent constraint. This predicate can be applied where it is needed.

  Example:

  ```
  pred AddBirthday( bb, bb': BirthdayBook, n: Name,
  d: Date) {
       bb'.date = bb.date ++ (n->d)
      // where ++ is the override operator.
  }
  ```

  This predicate adds a relation between a name and a date.

- *assert*: An assertion is a theorem about a specification or property that is expected to hold in the model. The Alloy analyser can check the assertion looking for a counterexample. For instance, if the predicate *DelBirthday* is declared that specifies how is deleted a relation for a given name, the next assertion could be done:

  ```
  assert DelIsUndo{
       all bb1, bb2, bb3: BirthdayBook,
          n: Name,d: Date |
         AddBirthday(bb1, bb2, n, d) &&
         DelBirthday(bb2, bb3, n) =>
            bb1.date = bb3.date
  }
  ```

Where it is basically claiming that adding and deleting one record of the birthday book is equivalent to a null operation. The idea is to be able to verify that deleting a previously added birthday from the *BirthdayBook* leaves the model in the original state, before the birthday was added. To do this, declaring 3 instances of *BirthdayBook*: *bb1* represents the initial state, *bb2* represents an intermediate state, and *bb3* represents the final state. To check the assertion it is necessary to use the *check* command, which is shown in the next section. An assertion is only checked in response to check-requests, while a fact is always considered true.

- *fun*: A function *fun* is similar to a predicate, but a function is like a template for an expression. A function returns a value as an expression while a predicate only can return true or false.

Another important aspect to remember is that assertions can be *checked*, while predicates and functions can be *instantiated* or *simulated*.

2.4.4 Executing an analysis

Alloy has two commands to execute an analysis that implies constraint resolution:

- *run*. The *run* command specifies to the Alloy analyser that it should find an instance in accord with a predicate or a function. The shortest form of the *run* command only requires the name of the predicate or function, for example:

  ```
  run BusyDay
  ```

BusyDay will be analysed in a default scope of three, bounding the size of the sets. Furthermore, the bounds of the sets can be specified by specifying the numbers of each type explicitly, in the command, for instance:

```
run BusyDay for 4 but 1 BirthdayBook
```

This indicates an analysis with a scope of 4 for each top-level signature, *but* at most one for *BirthdayBook*.

In this case, *BusyDay* is a predicate that shows a situation when the birthday book has more than one birthday on a particular day. *BusyDay* has been declared as follows:

```
pred BusyDay (bb: BirthdayBook, d: Date){

      some cards: set Name | cards=(bb.date).d

        && !lone   cards

}
```

As a result of executing *run BusyDay*, the Alloy tool generates a valid instance, shown in Figure 2.1, where the case in which an instance of *BirthdayBook* has two relations *date* which link to the same occurrence of day (*Date0*) through the instances of name (*Name0* and *Name1*) can be seen.

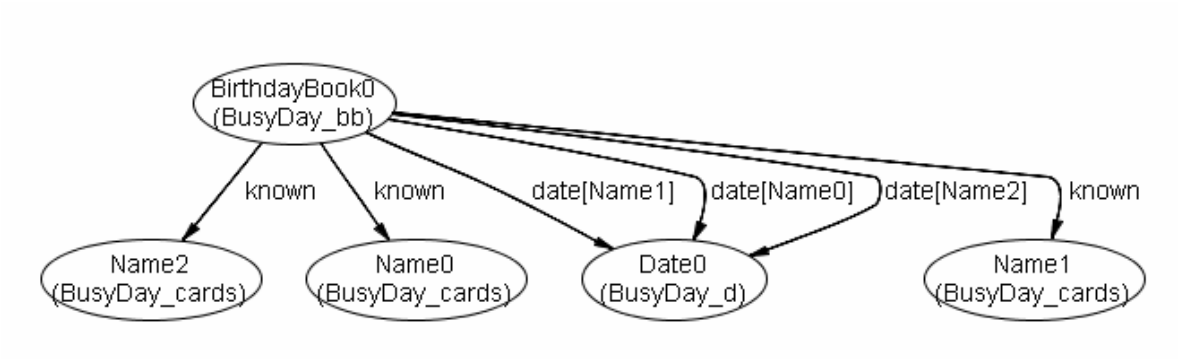

Figure 2.1 Instance generated by the execution of the *BusyDay* predicate

- *check*. This command is used to indicate that an assertion should be checked by the Alloy analyser. Syntactically, the *check* command has the same options as *run*.

  ```
  check DelIsUndo for 3 but 2 BirthdayBook
  ```

DelIsUndo is an assertion that verifies whether adding one birthday and then deleting it leaves the date relation in its original state. In this case, the assertion is not valid and generates the counterexample shown in Figure 2.2. The counterexample shows the case where, after adding one birthday, the *BirthdayBook* finish in the same state (*bb1* = *bb2*) due to adding an existing date; thus deleting a record after adding it does not guarantee that the *BirthdayBook* returns to its former state. Figure 2.2 shows this, where the *bb3.date* is not the same as the *bb1.date*. Here, remembering the expression of the assertion, *bb1* represents the initial state of *BirthdayBook*, *bb2* is the intermediate state (after the birthday was added), and *bb3* represents the final state of *BirthdayBook* (when the birthday has been deleted). Therefore the counterexample is saying that there is at least one case where adding and deleting a birthday does not leave the model in the original state: if the name added existed previously in the *BirthdayBook,* this and its date is overwritten, so deleting it makes the *BirthdayBook* enter a different state from the initial one.

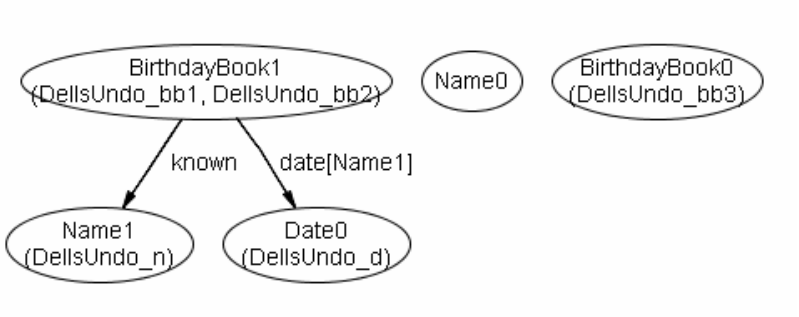

Figure 2.2. Counterexample generated by the execution of the *DelIsUndo* assertion.

In fact, the *addBirthday* predicate in Alloy differs from its equivalent Z schema because the *addBirthday* schema rules out the possibility of adding the same name twice. If it is desired to prevent the addition of the same name, it is necessary to include the next constraint in the *addBirthday* predicate:

```
n !in bb'.known
```

Subsequently, the execution of the *DelIsUndo* assertion will not generate a counterexample.

To conclude this section, it is important to say that neither the *run* nor *check* commands directly execute an action, in the sense of executing a program. They only act as an instruction to the compiler to identify what formula paragraphs should be checked, within a particular scope.

2.4.5 Metamodel

The software that implements the Alloy analyser has been improved. The design of the user interface is easier than the previous implementation. The visualisation of the graphs is quicker and it has a better presentation. Furthermore, it now includes a new feature to present a metamodel based on the source code. The Figure 2.3 shows the metamodel for the *BirthdayBook*; there, the relation *known* can be seen, from *BirthdayBook* to *Name*, and the relation between the sets *Name* and *Date* (*BirthdayBook.date*). It is relevant to note that every signature descends directly or indirectly from *univ*, which represents the predefined universal type, as is shown in Figure 2.3.

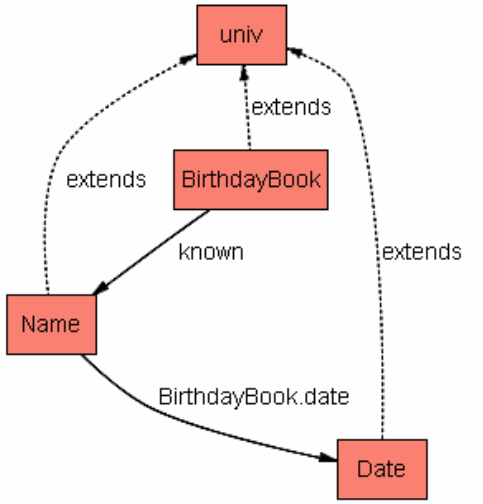

Figure 2.3. Metamodel for BirthdayBook

2.4.6 Summary

Although Alloy was based originally on Z, it now contains features that distinguish it from Z and other languages. While Z is defined in a classical set-based type system, Alloy supports subtyping and overloading [59]. An introduction to the Alloy analyser showing a little model was presented, based on the classic Z example of Spivey [41].

There are more characteristics of Alloy. A complete description can be seen in [36, 49] for the version 2, and in [52] and [58], which are the reference manual and tutorial for the version 3, respectively.

2.5 OCL

The Object Constraint Language (OCL) is a formal language created with the intention to support UML, giving the possibility to create constraints on a UML model in a formal way [60]; this language is part of UML since version 1.1. OCL is used in conjunction with UML and, like Alloy, avoids using mathematical symbols [7], using a syntax similar to object-oriented languages.

OCL can be used essentially to specify invariants on UML classes, to describe constraints, preconditions and postconditions on operations or as a query language [19]. It is for this reason that OCL expressions are written in a particular context and for a UML stereotype such as <<*invariant*>>, <<*precondition*>>, and <<*postcondition*>>.

2.5.1 Syntax overview

If an OCL expression is specified as an invariant of a type, the OCL expression must be true for all the instances of this type. The keyword *inv* defines the OCL expression to be an invariant constraint. The general syntax of the invariants can be seen as follows:

```
context classifierContext inv [name]:

     <oclExpression>
```

Where *classifierContext* represents the name of a classifier, *[name]* indicates an optional name for the invariant, and <*oclExpression*> represents the constraint definition.

On the other hand, precondition and postcondition expressions are attached to operations that can be defined both in the same context:

```
context typename::operationName(param1 : Type1, …)

 : ReturnType

     pre  [name]:   <oclExpression>

     post [name]:   <oclExpression>
```

Where the declaration of the context for the operation is similar to the declaration of an operation for programming languages and preconditions and postconditions can be declared (both or one of these) with an optional name. An optional keyword *result* can be used to denote the result of the operation.

Additionally, OCL has primitive types, collection operations and predefined OCL types that can be used to constrain the UML models. On the other hand, OCL offers the possibility of declaring a constraint as derivation rule, initial value, and body of query operation, using the keyword *derive*, *init*, and *body* respectively. The former is used in OCL to define how the value of a derived attribute or association is obtained. The second one is used to define initial values for attributes and associations. The

latter permits an expression (called the body expression) to specify the result of a query operation[2].

2.5.2 BirthdayBook example

Below, the same *Birthdaybook* example based on the original by Spivey[31, 41] showed before is presented. As OCL was created with the aim to formalise UML, OCL expressions should be used in conjunction with UML diagrams. For the example, a possible representation could be the showed in Figure 2.4, defining a *Birthdaybook* class containing a map between the names and dates for the birthdays. Clearly, assuming a declaration of the types NAME, and DATE.

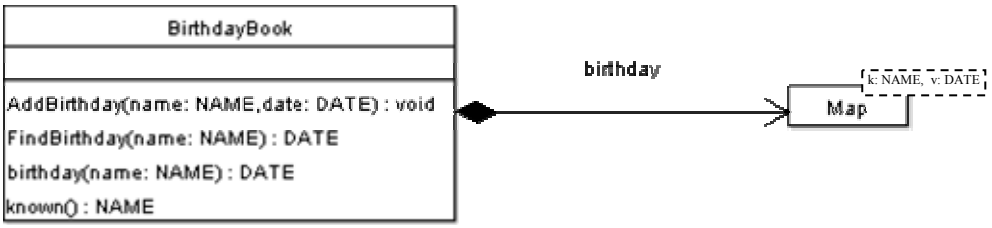

Figure 2.4. A possible representation of BirthdayBook

To represent this in OCL, an OCL tool was used to check the constraints and their correspondence with the UML diagram. A deeper analysis of OCL tools can be seen in [61], where the authors present a brief description of the most important categories of tools supporting OCL: syntactical analysis, type checking, logical consistency checking, dynamic invariant validation, dynamic pre/postcondition validation, test automation, and code verification and synthesis. In addition, Richters and Gogolla [60] present a comparison between OCL tools based on the above categorisation. In [62] a set of OCL tools are compared taking as criteria: degree of analysis capabilities of each application (syntactic analysis and type checking), facilities of the tool, capacity for dynamic validation, and OCL version supported.

USE[3] and ArgoUML were chosen because of their support for OCL in connection with the UML class diagram. USE was developed by Mark Richters [18, 63] at the University of Bremen, and ArgoUML [64] is an open source project available from 1998. Neither USE nor ArgoUML support class templates, so the original design for *BirthdayBook* could not be represented. Instead the function $NAME \rightarrow DATE$ was depicted as an association between NAME and DATE.

Additionally, from tool to tool, the OCL support varies and it was not possible to have exactly the same code for this little example. Figure 2.5 depicts the model using ArgoUML, showing the relationships and the two operations *addBirthday* and *findBirthday* from the example.

[2] A query operation is an operation that does not change the state of the system.

[3] Acronym of UML-based Specification Environment

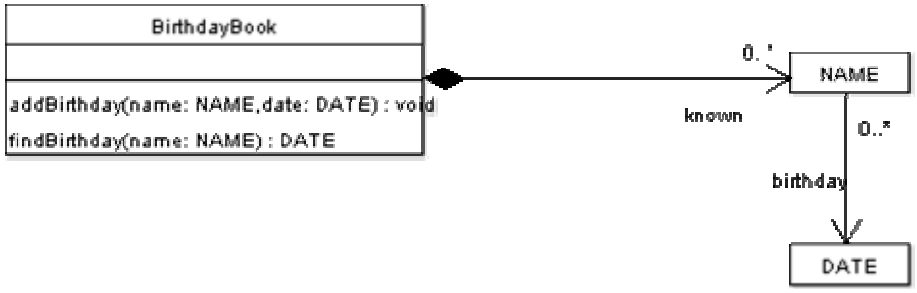

Figure 2.5. BirthdayBook modelled in ArgoUML

The OCL constraints using ArgoUML are shown below. The roles *known* and *birthday* are used directly since it was not possible to use the operations *known()* and *birthday()* to represent the corresponding elements for the Z example. The OCL constraints checked in ArgoUML are shown as follows:

```
context BirthdayBook::addBirthday (name: NAME;

 date: DATE)

     pre : self. known->excludes(name)

     post : self. known->includes(name)


context BirthdayBook::findBirthday (name: NAME): DATE

     pre findB : self. known -> includes (name)

     post findB_1 : result->includes(self.known.birthday)
```

As it was said before, the same example is represented in the USE tool. With USE, the operations *known()* and *birthday()* could be employed, declaring both as side effect-free operations. The Figure 2.6 shows the implementation for USE, in a similar solution as was presented for ArgoUML.

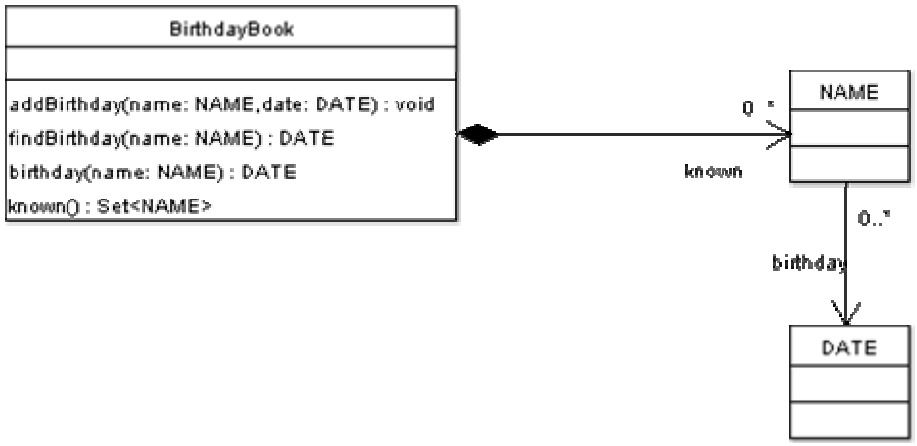

Figure 2.6. BirthdayBook modelled in USE

Because USE has no GUI to allow the user to design the UML class diagram, the model is represented in a textual way. The classes contain information about the attributes and operations, but not about the associations, which must be specified separately. The definition of the model for the USE tool is shown as follows:

```
model BBook

class BirthdayBook

operations

     known(): Set(NAME)= self.known

     birthday(name: NAME): DATE =

       self.known -> collect(name).birthday ->

       any(self.known -> collect(name).birthday->size=1)

     addBirthday (name: NAME, date: DATE)

     findBirthday ( name : NAME ) : DATE

end

class NAME

end

class DATE

end

-- Relationships

association aDate between

     NAME [*]

     DATE [1] role birthday

end

composition names between

     BirthdayBook [1]

     NAME [*] role known

end
```

Lastly, the constraints used for the model are shown below. It can be seen that *known()* and *birthday()* are used because both operations were declared as side effect-free, adding an OCL expression to the declaration of the operations.

```
-- Constraints

constraints

context BirthdayBook

     inv: self.known->includesAll( self.known() )

context BirthdayBook::addBirthday (name: NAME, date:
DATE)

     pre : self. known()->excludes(name)

     post : self. known()->includes(name)


context BirthdayBook::findBirthday (name:NAME) : DATE

     pre : self.known()->includes(name)

     post : result = self.birthday(name)
```

2.5.3 Summary

Since the creation of OCL a number of problems have been identified. In [65, 66] there have been mentioned for example problems with the definition of types, meta types, and expression types. Additionally, the author mentioned some restrictions to combine types for the creation of complex types, problems with the undefined value that, in opinion of Gogolla and Richters, are too strong for some cases and unclear for others (e.g. is undefined treated as false or true for a Boolean value?). Additional troubles were reported in [67] such as incompleteness of concepts (there is not full support for all the UML diagrams) and the possibility of creating ambiguous expressions. Finally, problems of representation can be appreciated even in this little example. The majority of the tools only support a subset of the UML diagrams, and additionally the support for OCL can be different from one tool to the next. Another problem is the poor semantic definition of OCL [68], which makes it difficult to interpret in an equivalent way for the different tools.

2.6 Process Algebra

The term process algebra or process calculus is used to define an axiomatic approach for processes. There is not a unique definition for processes although Baeten [69] says a process refers to the behaviour of a system. Process Algebras have been used to model concurrent systems [70]. Common concepts in the different process algebras are *process* (sometimes called agent) and *action* [71]. A process can be seen as any concurrent system with behaviour based in discrete actions. An action is considered something that happens instantaneously and it is atomic. An action is expressed in conjunction with other actions, using particular operations defined by the algebras.

Some of the principal process algebras comprise ACP, CCS, CSP, and more recently Pi-Calculus. The term process algebra was coined by Bergstra and Klop in the paper [72] where the Algebra of Communicating Processes (ACP) was presented. The Calculus of Communicating Systems (CCS) was proposed by Milner [73]. The contrasting calculus of Communicating Sequential Processes (CSP) was proposed by Hoare [74]. An extension and revision to CCS, the Pi-calculus was later proposed by Milner [75].

2.6.1 ACP

The Algebra of Communicating Processes is an algebra proposed in 1982 when Bergstra and Klop wanted to research a question about unguarded recursive equations [69]. The algebra is defined using a combination of instantaneous atomic actions and algebraic operators, in order to generate a variety of processes. These operators are used to represent union, concatenation and concurrency:

- Concatenation, also known as composition or sequencing, uses the symbol · and represents the order of the actions. For example, $a \cdot b \cdot c$ indicates that action a happens before action b and action b happens before action c.

- Union is used to specify a choice between actions, using the symbol + to represent the union. For example, $a+b$ represents that action a or action b can occur but not both of them.

- Concurrency is represented with the interleaving || and left-merge operator $\lfloor\!\lfloor$, where $p \parallel q$ allows all possible interleavings of actions in the processes p and q, whereas $p \lfloor\!\lfloor q$ always prefers the first action of p before the first action of q and otherwise behaves like ||.

These operators satisfy the following axioms (for all $a \in Action$, and $x,y,z \in Process$):

$$x+y=y+x$$

$$x+(y+z)=(x+y)+z$$

$$x+x = x$$

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$

$$(x+y) \cdot z = x \cdot z + y \cdot z$$

$$x \parallel y = (x \lfloor\!\lfloor y) + (y \lfloor\!\lfloor x)$$

$$(a.x) \lfloor\!\lfloor y = a.(x \parallel y)$$

$$(x+y) \parallel z = (x \parallel z) + (y \parallel z)$$

$$a \lfloor\!\lfloor y = a \cdot y$$

As was mentioned, these axioms just expressed the concatenation, union and concurrency (via the left-merge operator). These axioms represent the Basic Process Algebra, which was later extended to include communication as presented by Bergstra in [76].

2.6.2 CCS

Even though the Calculus of Communicating Systems was presented by Milner in 1973, it was not until 1980 that he published the book [73] that is now considered the definitive reference on CCS. In CCS a process is represented by a number of states representing the possible lines of action that can be realised. The states of the process are presented as dots (usually open dots), while the actions represent the transitions from a state to other [71]. The rules and axioms in CCS are provided as laws.

In CCS, 0 (nil) represents the most basic process, representing halting, or deadlock. CCS also provides an action prefixing operator, where a.P denotes that an action *a* can be prefixed to a process *P* to denote sequential composition of *a* and *P*. An action can be interpreted as an input or output communication on a port. By convention, *a* denotes input and $\bar{a}$ denotes output.

The choice operator proposed by Milner in CCS is +. It is commutative, associative and idempotent. Additionally, the CCS operator | represents parallel composition, where, for instance, the expression *P*|*Q* depicts two processes running in parallel. Communication between two processes happens when there is an action *a* in one process and a complementary action $\bar{a}$ in the other one.

2.6.3 CSP

CSP was proposed by Hoare in [77], initially without a formally defined semantics. Later a semantic model was proposed based on trace theory [78]. A new model was proposed and CSP changed its name to Theoretical CSP (TCSP) [79], which later was called again CSP.

The trivial element in CSP is the event, which is defined as instantaneous and indivisible. Events are notated in lowercase, for instance *x, y, z.* are events in CSP. Processes are notated in uppercase. There are also primitive processes such as *STOP* and *SKIP* to represent basic predefined behaviours.

CSP builds processes from actions using a prefix operator →, such that $x \rightarrow P$ denotes a process formed by prefixing the process *P* with the event *x*. CSP has two choice operators, for external and internal choice. The external choice operator , is defined, such that $(x \rightarrow P)$, $(y \rightarrow Q)$ denotes a choice between two processes, according to whether the environment supplies the event *x* or *y*, after which one of *P* or *Q* executes, respectively. The internal choice operator ⊓ makes a nondeterministic choice and may refuse events from the environment. A response is only mandatory if all prefixes are available. Concurrency is represented by the interleaving operator |||, such that P ||| Q denotes a nondeterministic choice between all possible interleavings of the actions of P and Q. The synchronising operator $\|_A$ forces its operands to synchronise, such that P $\|_A$ Q forces synchronised communication between P and Q on all the events in A.

2.7 Summary

Formal methods have been adopted widely and probably this tendency will continue in the future. But, this adoption has been slow and essentially only in critical systems.

A number of problems are commonly mentioned when people talk about formal methods, for example: difficulty of use, the lack of tools to help the developers to build software, and that use of formal methods implies much additional work. In fact, it is possible that some of these opinions are subjective, sometimes overestimating the difficulty and in other occasions underestimating the utility of formal methods. These problems, which had been mentioned initially by Hall [80] and later by Bowen [81, 82], are essentially caused by preconceived ideas that people have about formal methods.

Additionally, certain formal methods that are in common use have been presented. Originally the idea about OCL was to create a less intimidating formal language for users than other formal languages such as Z, but conceptually it results almost certainly a more complicated language [83]. Whilst OCL tools can evaluate constraints for given instances, Alloy can search instances and counterexamples in larges spaces. This is because the analysis made by OCL tools and Alloy is different. For OCL tools, users give an instance and the OCL tool checks if it satisfies the constraints, evaluating each subexpression to know if the constraint is valid [54]. So while Alloy performs a bounded exhaustive analysis of models, OCL tools have only been featured for parsing and simulation of their OCL models [84]. In addition, Alloy seems to be easier to use for developing declarative models.

On the other hand, OCL depends on UML notations for part of its representation and Alloy is independent and it is based completely on textual keywords. Wallace [84] predicted a visual layer representation for Alloy in order to facilitate the generation of models. Until now, the Alloy analyser tool can create a visual metamodel similar to the typical class diagram, but this is created from the Alloy code and the opposite process is not possible.

Process Algebras are used to model concurrent systems. Common concepts in the different process algebras such as *process* and *action* could be used to represent similar concepts in software modelling. This is the approach this project uses and is initially presented in Chapter 4.

In this chapter, an overview of formal methods was described with an emphasis on model based languages and process algebra languages. Examples of Z, Alloy and OCL were presented in order to identify the basic capabilities of the languages. The next chapter will discuss object-oriented methods and, in particular, the Discovery Method.

Chapter 3:
Object Oriented Methodologies and the Discovery Method

This chapter is presents a brief history of object-oriented methods and some problems that UML has had. An introduction to the Discovery Method is presented in its four phases: Business Modelling, Object Modelling, System Modelling, and Software Modelling. Chapter 4 will focus in the task models used in the Business Modelling phase.

3.1 Introduction

Object–oriented notations are commonly used in software design. So far the most used notation is UML. The acceptance of UML has been a success in the sense of merging the principal competing of object-oriented notations, but the problem of expensive mistakes in software is still there. This is in part because UML does not resolve any of the essential problems in software development. There still exists a big difference between the model, the code, and the real users' specifications. On many occasions, a model in UML is understood in different ways by different people in the development team.

UML is not the only object-oriented notation with these kinds of problems, but it is for now the most popular. Nonetheless, it is believed that it should be a more precise language. In addition, UML does not have good semantic representation. In fact, the semantics in UML are not really formal semantics; they are more like a meta-model that describes how a UML model should be constructed to be well-formed syntactically, and does not give the meaning of the UML notation [1-3].

There exist other less general notations, oriented to particular types of systems and having fewer ambiguity problems. The Discovery method [25, 26] is an object-oriented method mainly used for business modelling. It uses simple notations, some of them similar to UML notations, observing the original purpose for which the notations were conceived.

This chapter mentions the common problems found in UML and gives a description of the Discovery Method and its concern to offer a more concise working notation. In the next section, this chapter mentions briefly the history of object-oriented methods; section 3.3 mentions some of the problems identified when working with UML; the last section (3.4) explains the Discovery method.

3.2 A brief object-oriented history

In the 1980's, object-oriented programming (OOP) began to be used by software companies, but it was not until the end of this decade that several methodologies for object-oriented analysis and design emerged as a consequence of the old structured methodologies not integrating properly with the OOP style. In fact, in the history of object-oriented methodologies (OOM), three generations can be identified [85, 86].

The first generation originated in the late 1980s. In this generation, generally individuals or small groups devised the methods. In 1988, Sally Shlaer and Steve Mellor published their methodology with the name of Object-oriented Systems Analysis [87]. The Smalltalk community at Portland, Oregon contributed with the Class-Responsibility-Collaboration (CRC) technique in 1989 [88], and Responsibility-Driven Design in the same year [89, 90], based both on the behaviour of the objects. 1991 was a prolific year for OOM: Peter Coad and Ed Yourdon [91, 92] presented their own approach. In the same year, working in the research laboratories of General Electric, Jim Rumbaugh et al. [93] published the Object Modeling Technique (OMT). Additionally, the first version of Booch's methodology was published [94].

The second generation is identified from the early 1990s. These methods were still authored by small groups but ideas from other methods were taken. Ivar Jacobson, working for Ericsson, introduced the concept of Use Cases (1992) and his method was known as Objectory [95], which afterwards became the basis for the Rational Unified Process. The second versions of the methodologies of Booch and Rumbaugh (Booch 94, OMT-2) are more similar; for instance, Booch 94 [96] adds the concept of relationships and state graphs and OMT-2 [97] eliminates the data flow diagram from the functional model. Other important methods from this generation are Fusion [98, 99], MOSES [100], and BON [101].

The third generation of methodologies was generated in the mid-1990s from larger collaborative groups. In 1994, Rumbaugh left General Electric and he joined Booch in Rational Software in order to work initially on an "unified method". The next year, Rational Software bought Objectory SA and Jacobson came to work with them. In 1996 they proposed the Unified Modeling Language (UML) [1]. The final document of UML was submitted to the Object Management Group (OMG) in September of 1997 and in November of the same year was accepted as standard. However, UML was only a set of notations without an associated process.

Another approach that was submitted to the OMG in the same period was OPEN [86, 102] (Object-oriented Process, Environment and Notation). OPEN laid emphasis on the whole development process, rather than just the notation, but the OMG did not accept this proposal. OPEN provides flexibility: derived from the OPEN metamodel-based framework, an OPEN process can be tailored to suit individual domains or projects taking into account personal skills, organizational culture and requirements peculiar to each industry domain. OPEN [103] was initially created by the fusion of methods as MOSES [100], SOMA [104], Firesmith [105], and Synthesis [106], and more recently with ideas from BON [101], Ooram [107], and UML.

One problem with OPEN or other eclectic development methods (such as RUP [1]), which offer general guidance and a plan of possible activities to be carried out at

various points in the software lifecycle, is the necessity to be skilled in using the method with the intention of identifying the better techniques that can be used for a specific project. In reality, these methods focus more on the over-arching management process [85] than on the detailed technical process, which means that the business management aspects of building a system tend to dominate other concerns [108]. On the other hand, we should not ignore the technical process [86]; this includes all the classic design techniques from conventional Software Engineering, from requirements to implementation, using techniques and notations to develop object-oriented systems.

3.3 UML problems

The UML can be seen as an eclectic collection of diagrams representing different points of view of a model. Supporters of UML say that the benefit for the designer is that he can choose the diagram he needs for his design, having some degree of liberty to interpret these diagrams. But this freedom might be considered as a lack of direction, and could be the result of the lack of unified semantics in UML [109]. In fact, Booch [110] said the current UML specification does not restrict graphical formats and *"there really is no 'illegal' UML graphical syntax"*. In UML 2.0, it is possible to define "UML profiles", which may specify both extensions to the syntax, or restrictions on the possible uses of diagrams in particular models [111]. In particular, it is possible to define custom syntax, provided that this can be explained as stereotypes of basic elements within the UML metamodel.

Simons and Graham highlight how UML's two strengths are also its greatest weaknesses [112]. On the one hand, UML is *eclectic*, adopting model notations from many other approaches, without considering fully how these models fit together, nor what kinds of development process they should support. To this extent, UML is a political, rather than technical, compromise. On the other hand, UML is *universal*, in the sense that the same diagrams are used in every stage of the software lifecycle, with the consequence that the level of detail in each diagram can vary widely; and different developers may interpret the same diagrams differently.

In [1, 34, 35] the authors present a collection of problems detected in the use and misuse of UML 1.3 by developers, classified in terms of consistency, ambiguity, adequacy, and cognitive recognition problems. In the articles [112, 113], the authors report problems with the <<*include*>> and <<*extend*>> relationships; where, for instance, developers use <<*extend*>> to represent exceptions and alternatives, but the semantics of <<*extend*>> does not handle these cases, since the <<*extend*>> relationship is considered only an insertion and the flow of control returns to the point after the call. Both these authors [113] and Lano et al. [114] observe that the logical dependencies implicit in these use case relationships can result in complex control flow. Even the original authors and leading exponents of UML seem to disagree in some fundamental details about use case relationships, as was apparent at the OOPSLA panel session on use cases [115] when Jacobson said that the old <<*use*>> was a kind of generalisation, and Cockburn said it represented functional composition (see also [113]). It is clear that <<*include*>> is meant to be composition in UML 2.0. However, composing use cases eventually breaks the desired granularity of a use case as "a single complete interaction of the user with a system that delivers a result of observable value" [95]. It is also clear that <<*extend*>> is meant to be an

insertion of optional behaviour in UML 2.0. However, the direction of dependency, from the inserted case to the case that is being extended, runs counter to logical dependency, in which the behaviour of the whole depends on the behaviour of its parts [113, 114]. Perhaps in recognition of these problems, the original UML authors admitted that it was not considered possible to forward-engineer the control structure of systems from use cases [2]. Use cases may describe behaviour but not how this behaviour is implemented.

Elsewhere in UML, diagrams do not support the development of logically sound control structures as much as might be expected. In sequence diagrams, Simons and Graham mention some problems with the use of focus bar [112]. They indicate, to begin with, that the consistent use of the focus bar obviates the need for return arrows. Figure 3.1c shows the proper use of the focus bars with stack-frame semantics, in which it is clear that control returns to the calling block upon termination, whilst Figure 3.1a shows a second process thread semantics as was allowed in UML 1.1, in which each object encompasses its own thread of control and can only communicate through synchronisation (the message arrows therefore have a different meaning from invocation). However, Figure 3.1b depicts the misuse of the focus bar since it does not represent any sensible invocation semantics; unfortunately, it is a common mistake made by developers. The option to omit focus bars altogether means that sequence diagrams are little more than timing diagrams and have no procedural interpretation. An additional problem in sequence diagrams, mentioned by the authors, is that the idea of a normal course with its extensions only works for simple examples and multi-branching control logic is difficult to visualise. UML 2.0 has addressed this by introducing better control structures, which encapsulate logical fragments of a diagram corresponding to alternatives, loops, and even external diagrams [111].

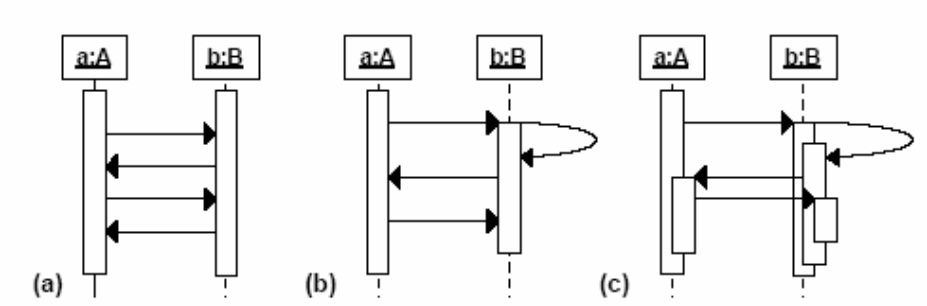

Figure 3.1 Thread, activation and stack-frame semantics focus bar [112]

The UML interaction diagrams (sequence and communication[4]) use two types of messages, procedural and non-procedural (They are also called nested flow control and flat flow control, respectively).The notation is shown in the Figure 3.2. While the procedural arrow has the semantics of method invocation, the non-procedural arrow is used as a kind of workflow in a flowchart because it represents the linear progression from step to step, without consideration for calls, returns and nested flows of control [2].

[4] Called collaboration diagram in UML 1.x

Figure 3.2 Procedural and non-procedural message flow

There are also inconsistencies with state and activity diagrams [112]. For instance, in practice, developers do not know if the initial pseudostate is really a true state, or simply a label identifying the intial transition to the first "real" state. Similarly, it is not clear whether the final pseudostate represents an accept state, in the sense of a Mealy machine, or a halt state after the last "real" state, in which the object is destroyed, or unreachable. Furthermore, both the initial and final pseudostates function as connectors for joining state machines to substate machines. In this case, the pseudsostates are not really states, but points midway along the initial, and final transitions to the substate machine.

Some of the complexity of the UML statechart arises because it is based on the Harel statechart [116], which is a combination of Mealy and Moore state machines. Figure 3.3 depicts the difference between these different approaches, contrasted with statechart notation.

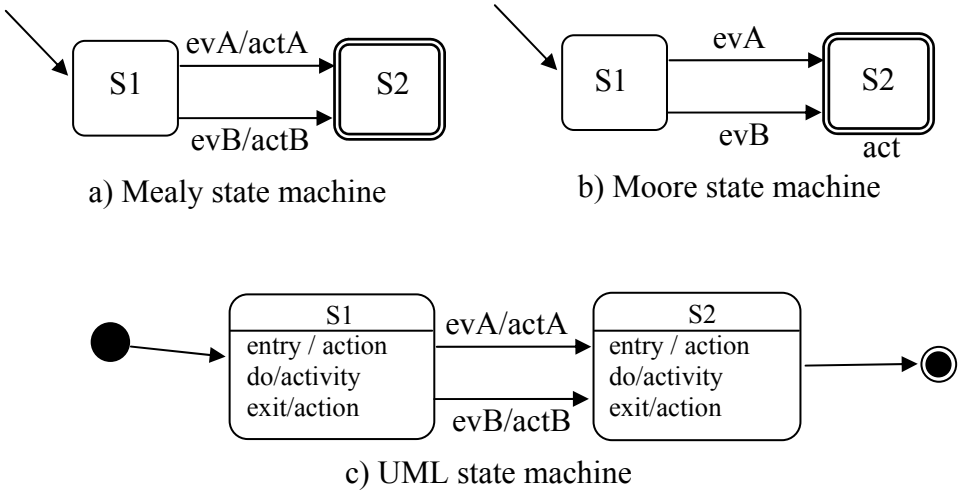

Figure 3.3 Mealy, Moore and UML state machines

In their original formulation, Mealy and Moore machines are apparently similar, in that states are quiescent and actions take place on the transitions, but whilst in the Mealy approach (Figure 3.3a) the triggered action depends on the transition taken, in the Moore state machine the triggered action depends on the reached state (Figure 3.3b). This makes it possible to adopt a different view, in which the action appears to take place in the state instead of on the transition. Once states are no longer quiescent, they become procedural, having a different kind of semantics, with the need for initialisation and self-termination. Since the standard states in UML statecharts are now potentially quite complex procedures, the UML statechart identifies as independent elements the initial and the final pseudostate. Actions may be triggered on the transitions, as in a Mealy machine. But it is also possible to have

internal operations in the states, only distantly inspired by the Moore machine, considering *entry* and *exit* events (and their activities), and internal transitions (such as the *do/activity,* which is commenced upon entry to the state, but which can be pre-empted by any exit transition) which represent transitions that do not change the state of the state machine. The final result is that, at this level, UML statecharts are more similar to the flowchart than to the original concept of state machine.

Some redundant constructions, such as state *entry* and *exit* actions, are notionally admitted to support the Moore machine viewpoint, but could be replaced by ordinary actions on transitions. This would obviate the need for special "internal transitions", which replicate standard re-entrant self-transitions, but which do not trigger the state entry and exit actions. The pre-emptable *do* activity that is carried out within a state tends to blur the notions of quiescent states and active procedures. These could be shown as substate machines without this extra notation. Also, the UML statechart allows the violation of state-encapsulation by crossing the boundary of a superstate with entry and exit arcs.

Simons [117] comments on a redundancy issue with statechart diagram notation, namely that statecharts allow concurrent transitions (i.e., *fork* and *join*) based on Petri nets, as well as a representation for concurrent composite states. Both of these notations denote the same concurrent semantics, so only one is strictly necessary. These cases are respectively exemplified in Figure 3.4a and Figure 3.4b.

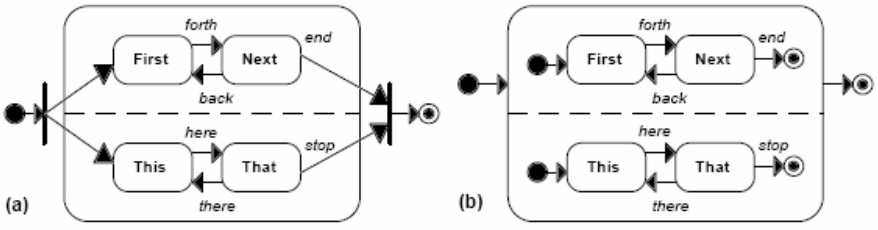

Figure 3.4 Equivalent UML models for concurrent substate machines [117]

In the use of class diagrams, Simons and Graham identify certain problems with decisions made during early stages [112]. For example, almost all the class diagrams created in the analysis phase have an excessive influence on the design. Something similar happens with the associations, which commonly are transferred from analysis to design producing poorly-coupled models. In addition, another interesting problem they identify is that the class diagram mixes up the notion of associations (Figure 3.5 (a)), that strictly describe data dependency, with functional dependencies, drawn as navigations (Figure 3.5 (b)) that strictly describe modular coupling. The class diagram may therefore confuse data dependency and functional dependency in the same model, preventing the developer from seeing how to transform a model into a more optimised design.

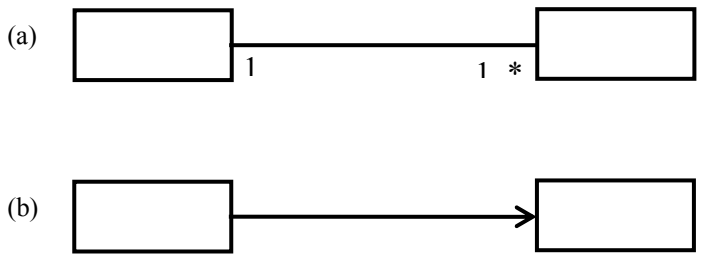

Figure 3.5 Association and navigation in UML class diagram

Whilst the notion of an association is inherited from the Entity-Relationship Modelling [118], the navigation arrow represents a functional dependency similar to the relationship expressed in Responsibility-Driven Design [119]. An Entity-Relationship diagram is used to optimise data dependency, by a process of converting all bidirectional many-to-many associations into simple many-to-one associations. A collaboration graph (in Responsibility-Driven Design [90]) is used to optimise functional coupling, by combining paths along which objects communicate with each other to access common services. Different kinds of transformations are intended for each model. Using both notations at the same time can be confusing, preventing the designer from seeing how to progress the design, or making it logically impossible to transform the design.

3.4 Discovery Method

The Discovery Method is an object-oriented methodology proposed formally in 1998 by Simons [25, 26]; it is considered by the author to be a method focused mostly on the technical process. From version 1, Discovery has been using a simple and semantically clearer notation based on UML, but changing some models where this is considered appropriate. In addition, it is consistent with the process model of OPEN [120], and has been tested in a number of industrial projects by MSc students at the University of Sheffield. The simple and unambiguous Discovery notation makes it an appropriate option to work with.

There are four overriding principles of the Discovery Method: *direction*, the need to have an intellectually linked sequence of design activities and products; *selectivity*, the desire to choose only those notations that are appropriate for a given design technique; *transformation*, a rejection of the "seamless" pressing of analysis models into design artefacts, but embracing traceable model transformations; and *engagement*, the desire to increase the bandwidth of communication between the customer and developer through clear notations and appropriate activities. In addition, the Discovery Method is organized into four phases; Business Modelling, Object Modelling, System Modelling, and Software Modelling[5].

In the Business Modelling phase, the idea is to get knowledge of the context of the system, capturing and analysing the requirements in order to reach a decision about the scope of the contract and costing. Object Modelling is the next phase and here the

[5] These phases were called Task Modelling, Object Modelling, System Modelling, and Language Modelling in [25, 26]

aim is to identify objects and modular units of design. For the third phase, System Modelling, it is needed to analyse the cross boundary and internal dependency (coupling and cohesion, respectively) and identify the natural subsystems. The last phase is Software Modelling, where work in the translation of designs into code in some specific programming language has to be done.

Contrary to UML, the Discovery Method tries to delay the creation of objects in the early stages, due to the fact that initial objects tend to persist throughout the rest of the design, introducing an early bias in the perception of the system and affecting the evolution of the project. The Discovery Method is based on techniques and notations mostly taken from existing methods, however it selects and evaluates each chosen technique and notation carefully, in order to use it for its single and original purpose. In this way, each successive analysis or design model is formed gradually from previous models.

Additionally, the Discovery Method uses the elements of its simplified notation in a consistent way, throughout the method, for example, if a symbol has a particular meaning in one kind of diagram, this symbol will have the same meaning in other diagrams, and wherever this symbol is used in other parts of the method. A clear example can be seen in the symbols for aggregation and generalisation, where these concepts have the same semantics in the Data Model and in the Task Structure Model, as shown in Figure 3.6. It eliminates the confusing <<*include*>> and <<*extend*>> used by UML in the Use Case diagram, showing a consistent behaviour for these structural relationships. Using the symbols consistently among different models will make it easier to comprehend the diagrams and eliminates confusing and complicated notations.

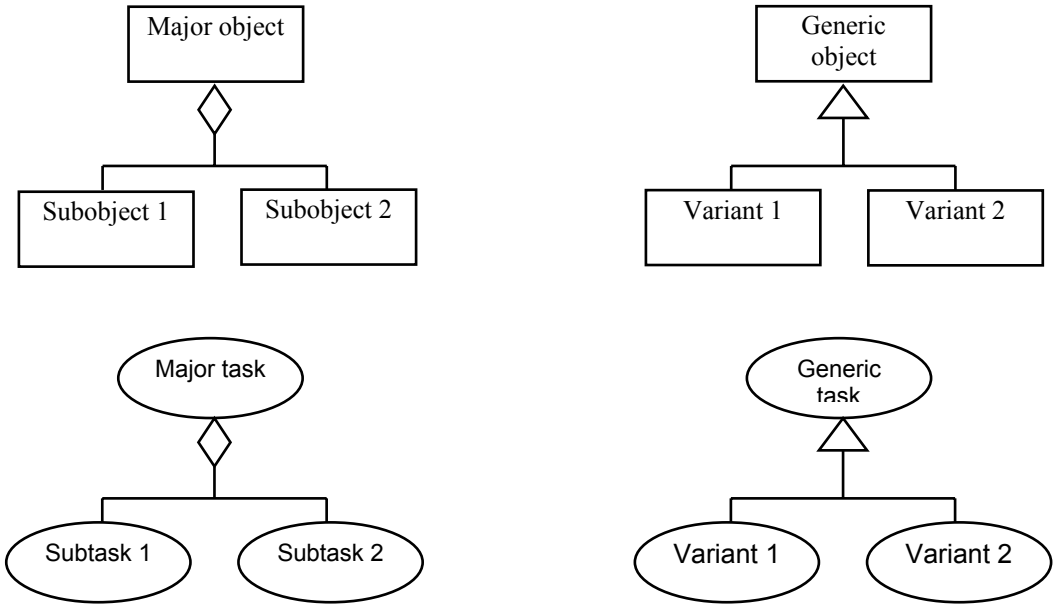

Figure 3.6 Aggregation and generalisation for Data and Task Structure Diagrams

The Discovery Method only has quiescent states in its State Model. The method does not consider it suitable to treat the states as active processing stages; in opposition to UML and Harel statecharts where, as it was mentioned before, *entry* and *exit* actions

are executed in the states, generating a kind of flowchart and not a statechart. The UML notation is slightly altered so that initial and final states are seen to be first-class states, rather than pseudostates. It is also more sensitive in distinguishing between final accept- and reject-states, using an independent symbol for each option. We can appreciate the notation in Figure 3.7.

Figure 3.7 Initial and final states are real states

The Discovery Method deploys various design techniques, which are used in their original intended context, which results in adopting subsets of simplified UML notations for different model specifications, but providing a greater linkage between models and adding clear semantics. Each technique is used for a single purpose and each model is built systematically, avoiding misapplication of the techniques out of their proper context as is frequent in other methodologies.

3.4.1 Business Modelling

The Business Modelling is the initial phase of the Discovery Method. In this phase, the goal is to explore and represent the requirements of the customer in a structured model of the business context where the system will work. The main activity is to identify the tasks that are part of the model, proposing an improved system in terms of business tasks and supporting the objectives of the client. It is also important in this phase to draw up a contract and a plan for incremental delivery of the system.

This phase consists of interviews, domain analysis, task analysis, and contract planning. Interviews are conducted by the "developer", a term denoting a person or a group of people from the software house, with the "customer", a term denoting the commissioning manager and potential users of the system. Interviews should use a non-directive technique. Discovery suggests three procedures for this objective:

- *Free Exploration* is the least directive technique and here the developer does not lead the customer to fixed choices, but allows him or her to express the most pressing business needs.

- In a second technique, called *Bluesky Wishlists* [121], the customer identifies the major stakeholders and their extreme preferences for the system. This technique should help to identify competing forces and, consequently, to recognise the possible constraints.

- The third technique is *Iterative Prompting*. It is a task-centred exploration trying to find the decomposition of tasks and their dependency. The customer should lead the discussion while the developer helps him using simple "wh-" questions (i.e., what?, who?, how?, why?, when?, where?). This technique was first used by Ian Graham [104] for task-centred analysis.

Task analysis, as a part of Business Modelling, consists of the three techniques *Task Sketching*, *Narrative Modelling*, and *Task Modelling*.

Task sketching may be done by using *Task Structure Sketching* or *Task Flow Sketching*. *Task Structure Sketching* is recommended to identify business tasks at a coarse scale. A task should capture an obvious business task rather than detailed system processes. The *Task Structure Diagram* is useful for representing tasks and their structural relationships such as aggregation and generalisation, which are explained later.

Additionally, *Task Flow Sketching* is a technique recommended to capture information about the order of execution of the tasks. Two sequential tasks are shown to be in a relationship by using a transition arrow that describes the control flow. Choice between tasks can be represented by diamonds splitting the flows, while an exception may be represented by a half-diamond splitting the normal flow from the exceptional flow. Identification of actors and their relationship with the tasks may also be done here. The *Task Structure Diagram* and *Task Flow Diagram* are explained in detail in chapter 4.

Narratives are used to describe the task of a business model. They are described by the customer but recorded by the developer. A narrative has to identify its purpose and the elements (actors and objects) that are part of the task. The description of the flow may include alternatives and exceptions. In addition, the preconditions and postconditions affecting the task are included.

Engaging in more detailed *Task Modelling* is the last part of the *Business Modelling* phase. Basically this consists of *Thematic Clustering*, *Logical Task Restructuring*, and establishing *Alternative Task Flows*. *Thematic Clustering* is the Discovery procedure for grouping tasks into more abstract tasks (or supertasks). There are two basic criteria for clustering: either by identifying tasks with a common goal, or identifying task with common actors and objects participating in a task. *Task Restructuring* is applied to improve the task structure design, mainly looking to reduce task dependency and isolate actors with many task participations. Finally, *Alternative Task Flows* is a procedure where the developer describes flows from the viewpoint of a particular object or actor.

3.4.2 Object Modelling

In the *Object Modelling* phase, the task descriptions from the *Business Modelling* phase are used to identify candidate objects in the business domain. In this stage, the analysis model, created in the Business Modelling phase and expressed as tasks, is transformed into a design model, expressed as collaborating objects. Object modelling is only partly an analysis activity where the object concepts are discovered in the business domain. Most of the activity in this phase is considered a design activity, with the invention of metaphors to denote units of software [122]. It is also in this phase that the developer creates decentralised component-based architectures and identifies patterns of collaboration.

The input for this phase is the business model, i.e., a set of task specifications. Object modelling uses Responsibility-Driven-Design (RDD) [89, 123] in a bottom-up approach for identifying object concepts and their responsibilities. The input for the *Object Modelling* phase is the specification of the main business tasks. This phase produces a set of candidate objects, their collaborations and responsibilities.

The architecture of business systems is modelled commonly as a three-layer structure, although this is not mandatory in the Discovery Method. The first layer, devoted to user interface modelling, is guided by the roles identified for the business stakeholders and the business tasks that they carry out.

The middle layer is used to represent the logic of the business. Business logic concepts are identified because they are mentioned in the narratives. *Gatekeeper* objects (another term used in the Discovery Method) are those objects with state that capture the business logic and are used to allow or prevent the execution of a task. Gatekeeper objects are often found as concepts named in the preconditions of the narratives. Task flow diagrams are drawn for each gatekeeper object, which helps the developer to describe how the object participates in the tasks. *State Diagrams* for the gatekeeper objects can be made by inverting the nodes and transitions of the Task Flow Diagrams. Gatekeeper state diagrams are important because they help to represent business processes and can be translated into software.

State Diagrams in the Discovery Method are traditional Mealy-style finite state machines formed by states and transitions, where initial and final states are real "first class" states. States are connected by transitions. A transition denotes the change from one state to another when a particular event takes place. The event could trigger an associated activity if this is defined. Where the event is not sufficient to select a unique transition in a state, transitions may also be guarded. A guard is a Boolean condition, which must be true for the transition to fire. The guards on events are preconditions. All guards in a set of preconditions must be mutually exclusive and exhaustive. There are two kinds of final states: reject- and accept states. An accept state represents the normal termination of the state diagram's execution. On the other hand, a reject state means a failure in the execution, specified by events and conditions defined in it. A diamond is used to denote a conditional branch in a transition, used only when the next state cannot be determined from the event and preconditions before the transition is fired. Each branch is guarded by a postcondition. Such guards are also mutually exclusive and exhaustive Boolean conditions; they are always depicted between square brackets. See Figure 3.8.

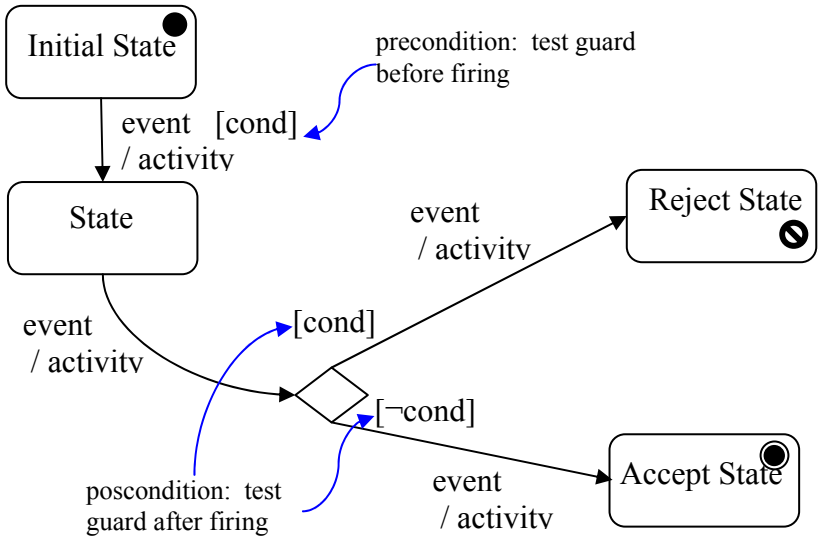

Figure 3.8 Notation for the State Diagram in the Discovery Method

The last layer is where the data model is represented. Data storage concepts are determined according to the needs of the business system. Typically, these consist of information concepts but gatekeeper objects are also major candidates, since they record the state of a business process. The data model can be constructed by using either Object-Association Modelling (OAM) or Event-Driven Design (EDD) [124, 125]. The aim of both approaches is transforming the data to a normalised set of tables.

As mentioned before, the output of this phase is a set of candidate objects. Each object should have a limited function and depend on other collaborator objects to achieve its responsibilities. The developer should write *Object Role Cards*, similar to CRC cards [123], to record the responsibilities, collaborators and attributes of the candidate objects.

The Discovery Method does not use a single class diagram to model relationships between the types of candidate object concepts. Instead, there are two separate models, the *Data Model*, which consists of record types related by associations, and the *Collaboration Diagram*, which consists of functional classes linked by collaborations, drawn as navigable associations. The notation is based on UML's class diagram notation but attempts to keep things simple using a reduced number of elements.

Figure 3.9 and Figure 3.10 show the most common elements used in both the Data Model and the Collaboration Diagram respectively.

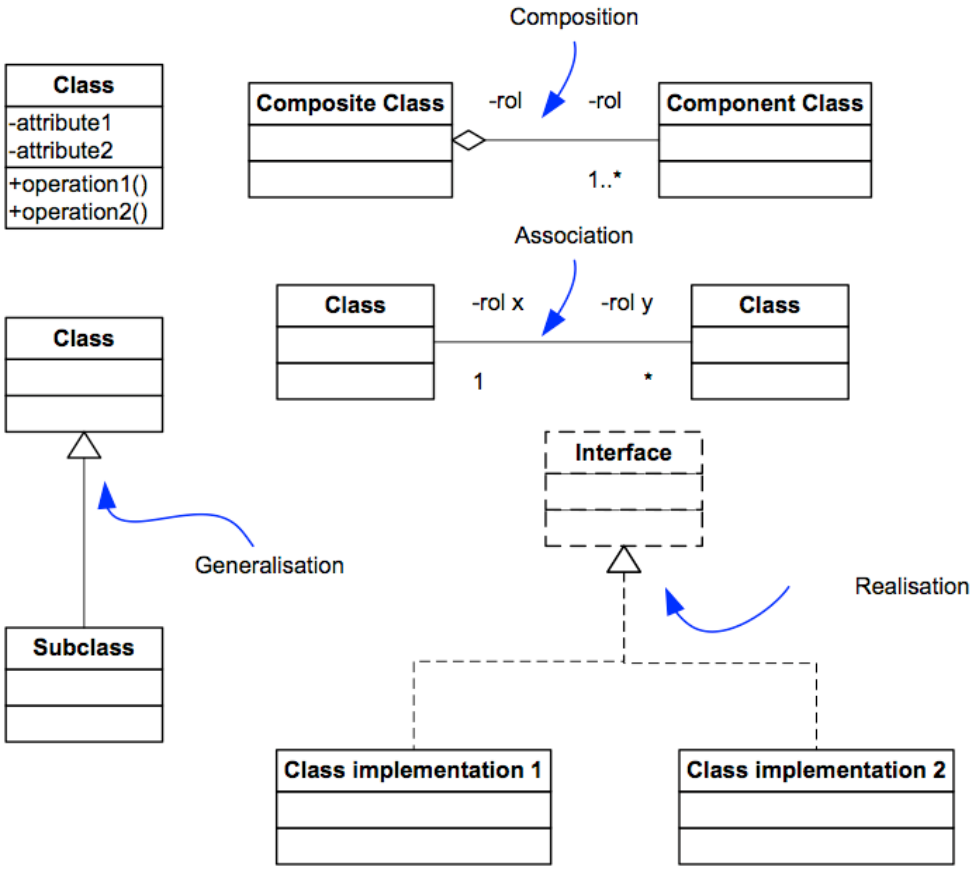

Figure 3.9 Notation for Data Diagram in the Discovery Method

Most elements are familiar from UML notation. The symbol for an interface type is drawn differently, as a dashed box, like the dashed shaft of the realisation arrow, instead of the normal box with the <<*interface*>> stereotype. This is part of the Discovery Method's UML profile, which extends the allowed notation in certain ways. The reason for this change is to notate the difference between abstract and concrete concepts uniformly. Dashed outlines are also used to notate the difference between abstract *Goals* (which have dashed ellipsoid outlines) from concrete *Tasks* (which have solid outlines).

As can be seen in Figure 3.10, the *Collaboration Diagram* is very similar to the *Data Diagram*, but it shows not vague associations. This diagram represents only unidirectional connections remarking how messages are sent from class to class.

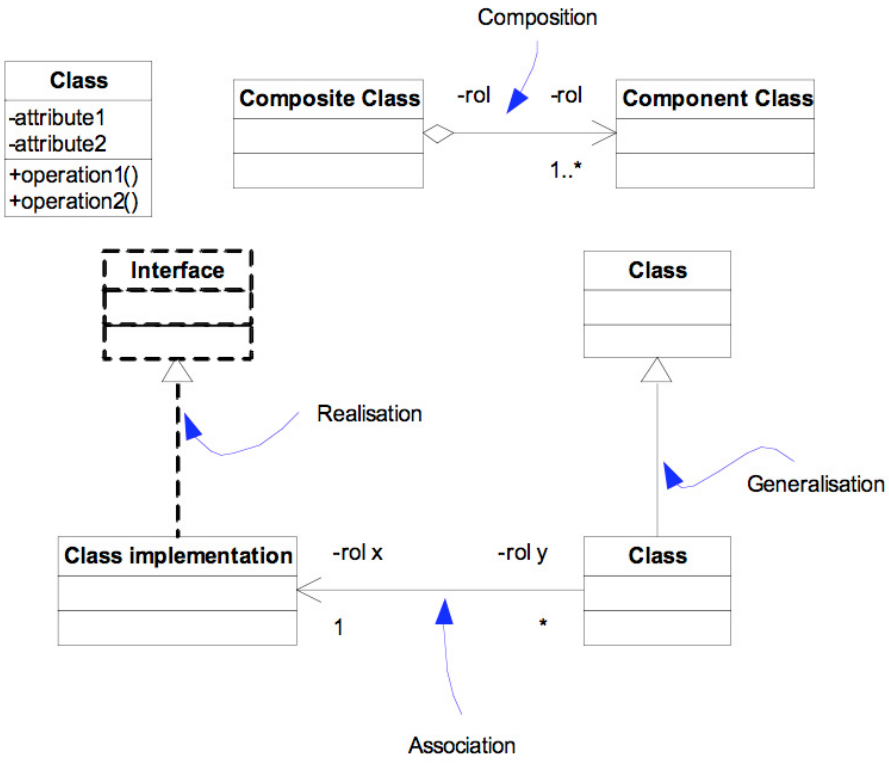

Figure 3.10 Notation for Collaboration Diagram in the Discovery Method

3.4.3 System Modelling

The *System Modelling* phase has the aim of identifying the optimal system architecture, discovering natural layers and subsystems, helping to modularise the design of the system, consequently facilitating the reduction of the strongly coupled graph of object roles. Additionally, the *System Modelling* phase contributes to the construction or maintaining of a framework. If a previous framework exists, the recent design of the system has to be merged with this framework.

For this phase the input is a set of object roles. Object roles are highly coupled by collaboration with other roles. Object roles that are densely coupled should be decoupled as part of the activities for this phase. Each object role can be potentially a

class or an interface and their responsibilities have to be balanced. The output for the system modelling phase is the optimal design and a specialised framework.

System modelling seeks to create a general picture of functional dependency among the candidate classes using the collaboration diagram proposed by Wirfs-Brock et al. [90] where the candidate classes that have a relationship of dependency are linked by an arrow. This diagram is not the same thing as the old UML diagram with the homonymous name, which now, since UML 2.0, is called the Communication Diagram instead. Figure 3.11 shows the Collaboration Diagram notation for the Discovery Method. A Collaboration Diagram is a graph showing the communication paths among classes. A collaboration arrow is used to represent a functional dependency, which means, in practical terms, that an object of the source class sends a message to another object of the destination class. The arrow points to the class receiving the message. In the figure, class *A* has a functional dependency on classes *B*, *C*, and *D*, while class *B* and *D* have a dependency on class *C*. Functional dependency can be weak or strong. A strong functional dependency may evolve into a permanent relationship between the classes, i.e., into a directed association or composition relationship, implemented using a reference. Part of the activity in *System Modelling* is designed to discover which collaborations are robust and should be encoded this way. Other collaborations will disappear after system transformation.

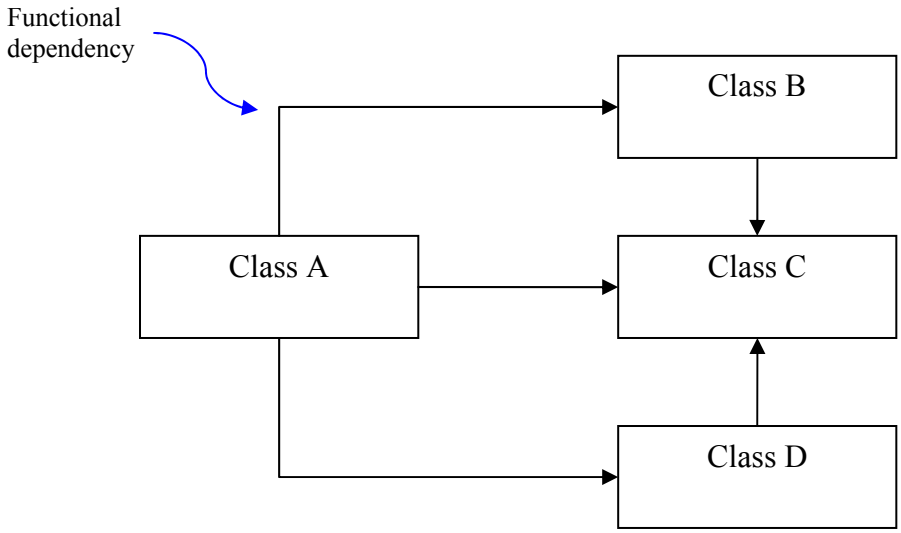

Figure 3.11 Notation for the Collaboration Diagram in the Discovery Method

Another important activity is to apply three kinds of design transformations based on [90]: the aggregation transformation, server generalisation, and client generalisation. With each transformation the flow of control changes and the object role cards have to be updated. These transformations tend to identify new intermediate abstractions and, in consequence, reduce the number of direct collaborations. Typically, this transformation process produces instances of recognisable design patterns [126], promoting a high-quality, generic and decoupled design for the system [125]. At this point the design of the system is considered to have the maturity of a white-box framework. It consists of many levels of specialisation, with plug-in points for new classes, which expect to override some of the general methods provided for the system. If some pre-existing framework has already been developed, the current

system is compared against this. At this point the developer has to decide whether it is better to deliver the current system to the optimal design, or adapt the system to the pre-existing framework. This choice is taken after considering how mature the pre-existing framework is. A framework usually starts to stabilise after three or more systems of the same kind are built [Simons, pers. comm.]. Once the framework has stabilised, it is possible to convert it in a black-box framework. This is one in which the plug-in points are converted into explicit typed interfaces. A black-box framework is less flexible to adaptation, but safer for type checking the inserted components.

3.4.4 Software Modelling

The aim of the *Software Modelling* phase is to transform the system design obtained in the last phase into source code in an object-oriented language. The election of the programming language is dependent on the non-functional requirements or the availability of existing frameworks.

The classes, interfaces and attributes of the design are translated to the language almost directly, but it is also possible to do particular translations of some design concepts, presented as code idioms. Responsibilities for the classes are implemented by several methods. References between classes can be implemented as references if the connections are long-term, or, for short-term references, using method arguments. Pre- and post-conditions are coded like executable assertions. The pre- and postconditions defined in the narratives should be codified into executable assertions in order to preserve the semantics of the operations.

Three kinds of testing are suggested to be used with diagrams in the Discovery method: protocol testing using state diagrams, flowgraph testing using narratives or communication diagrams, and acceptance testing.

3.5 Summary

In the previous chapter an introduction to formal methods was provided. In this chapter, a brief history of object-oriented methods was presented and some of the problems associated with UML were identified. An introduction to the Discovery Method was offered, organised according to its four phases: Business Modelling, Object Modelling, System Modelling, and Software Modelling. The next chapter will focus on the task models used in the Business Modelling phase.

Chapter 4:
The Informal Semantics for the Task Models

The previous chapter gave an introduction to object-oriented methods and compared the Discovery Method with other methods. In this chapter, the informal semantics for the Task Structure and Task Flow Diagrams are explained, as well as an introduction to the approximation used to define the formal semantics. Additionally, the possibility of using Alloy to define and verify the abstract syntax on the diagrams in the Discovery Method is explored. This approach was previously published in [127].

4.1 The informal semantics for the Task Diagrams

The Business Modelling phase is task-oriented. A task is defined in the Discovery Method as something that "has the specific sense of an activity carried out by stakeholders that has a business purpose" [Simons, pers. comm.]. This task-based exploration will lead eventually towards the two kinds of Task Diagrams: *The Task Structure* and *Task Flow Diagrams*.

The Task Model in the Discovery Method consists of the Task Structure diagram and the Task Flow diagram. The former is used to represent structural relationships between tasks: aggregation and generalisation. In contrast, the Task flow diagram is able to depict workflow relationships between tasks. Both diagrams have a number of limited correspondences [Simons, pers. comm.]. A set of generalisations in the task structure is only consistent with a selection between the specialised tasks in a Task Flow diagram. In addition, having an aggregation of tasks in a Task Structure diagram is consistent with sequence, selection, repetition and parallel composition of these tasks in a Task Flow diagram.

Two different approaches were used in this research to represent these diagrams. While Alloy is presented here to represent Task Structure diagrams, for the Task Flow diagrams a denotational semantics approach is used.

4.1.1 Task Structure Diagram

Based on the interviews, the developer discovers a collection of tasks and identifies the relationships between one task and another. He also will recognise the stakeholders involved with the tasks and these will be presented as stick figures, known as actors. The difference between a stakeholder and an actor is that, whereas a

stakeholder is a specific person with a unique set of priorities about the way the eventual system should operate, an actor relates to one formal role played by one or more stakeholders in the system. The notation used for the Task Diagrams in the Discovery Method is simple. The elements are taken from the UML but are presented in a more concise and consistent form that is, in some parts, unique to the Discovery Method. Figure 4.1 shows the elements used to relate tasks with actors and objects in the *Task Structure Diagram*.

A *task* is represented as a labelled ellipse, using the UML notation for use cases, but standing for all kinds of business activity ranging from a small-grained use case, up to a large-grained business process. Actors that participate in tasks are drawn as stick figures; and participating objects as rectangles. Relationships between actors or objects with tasks can be represented. Relationships between actors or objects with tasks are called *participation*, drawn as a simple straight line linking the elements. It indicates that the actor or object is involved with the task. In the case the actor or object is not just involved but also responsible for the task, this is shown with a small filled circle drawn at the task-end of the relationship. This is known as *ownership*.

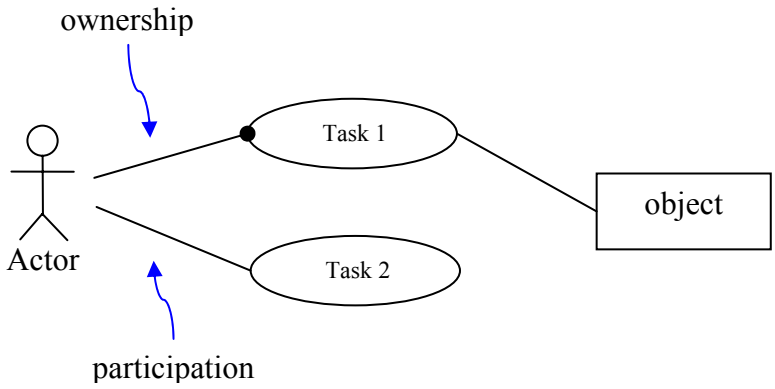

Figure 4.1 Basic elements of Task Structure Diagrams

The elements mentioned above can be insufficient if a more detailed diagram is required. The *Task Structure Diagram* utilises the generalisation and aggregation relationships, the two main structural relationships in UML, to define structural relationships between tasks. Aggregation and generalisation are used with the same meaning, avoiding the confusing structural relationships in UML use cases shown in Chapter 3. Figure 4.2 shows the generalization and aggregation notation in the *Task Structure Diagram*.

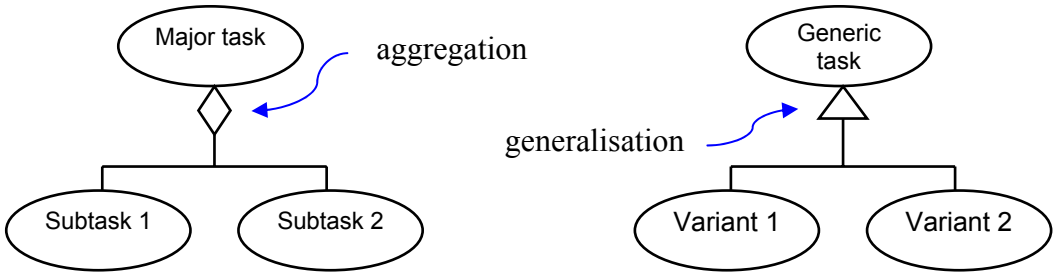

Figure 4.2 Structural relationships in the Task Structure Diagram

Aggregation, represented with a diamond arrowhead, indicates a major task divided into smaller sub-tasks. Aggregation specifies a whole-parts relationship where the whole indicated by the diamond arrowhead is formed by an encapsulated set of parts, at the other end of the relationship. Generalisation, presented using a triangle arrowhead, depicts a general task on the side of the arrow, and specialised tasks at the other end of the relationship. Generalisation describes a general-specific relationship where a more general abstract task generalises over a collection of more specific concrete tasks.

4.1.2 Task Flow Diagram

At some point during the process of identifying the tasks and structural relationships for the different actor viewpoints in the model, it will also be necessary to represent workflow relationships. The workflow is represented in the Discovery Method using the *Task Flow Diagram*. It depicts the order in which the tasks are realised in the business, expressing also the logical dependency between tasks. While the notation used in the Discovery Method is largely based on the Activity Diagram of UML, it maintains consistently the labelled ellipse notation for tasks. Figure 4.3 shows the notation for the *Task Flow Diagram*.

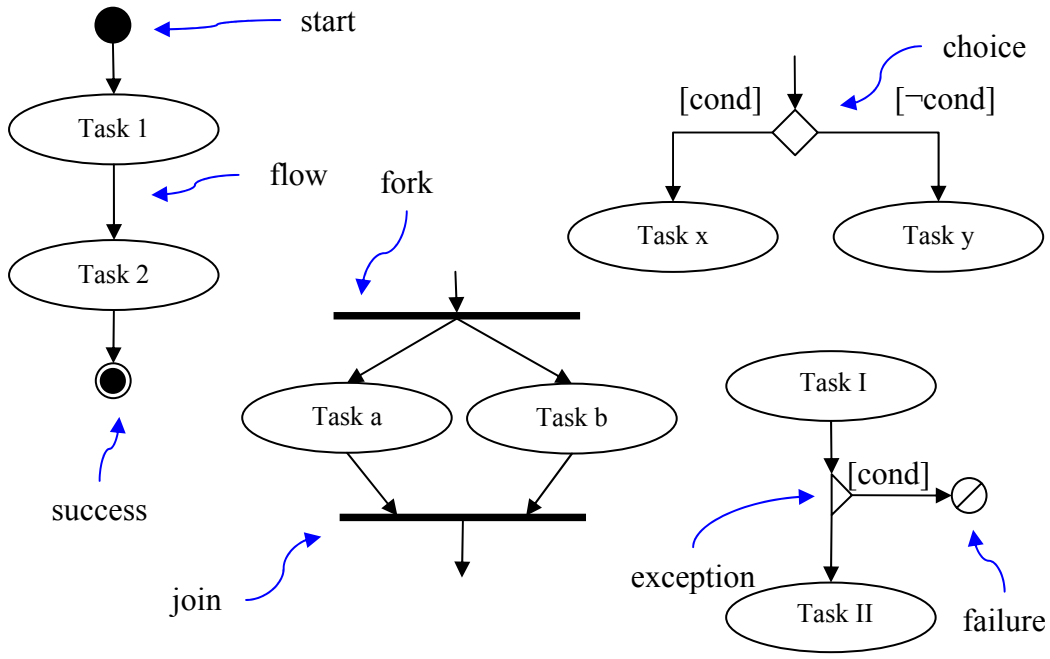

Figure 4.3 Elements of the Task Flow Diagram

Tasks are connected by an arrow indicating the direction of the flow. A choice is represented by a diamond; and an exception, a special case of a choice, is represented using a half-diamond symbol. The full diamond is used to split the flow in two or more alternative flows, whereas the half-diamond symbol represents the choice between continuing the normal flow or branching to the exceptional flow. Whereas mutually exclusive and exhaustive guard conditions must always be given for each branch of a standard choice, only the failure condition need be notated in an exceptional choice, where the continuation condition is understood to be the logical complement. *Start* and *end* symbols are the standard icons used elsewhere in flowcharts and state diagrams. There is also a particular kind of end identified as *fail*.

Fail is notated as a circle crossed by a diagonal line and represents, in the Discovery Method, exit with failure from the process described by the diagram. By contrast, the traditional *end* symbol represents exit with success from the current diagram.

Finally, the *Task Flow Diagram* in the Discovery method allows the representation of parallel tasks. This representation in the diagram is necessary because business processes, just like other kind of processes, are sometimes independent from other processes and, consequently, could be performed concurrently. The *Task Flow Diagram* employs the *fork* and *join* symbols to delimit two or more potentially parallel flows. The *fork* and *join* symbols are common to different notations used for flow and state diagrams. A comparison of different statecharts can be seen in [128]. A *fork* is a transition with one source task and multiple target tasks. A *join* is a transition from multiple source tasks to one target task. When a fork transition is taken, all of the target tasks after the fork transition are understood to begin simultaneously. Tasks in each subflow are executed sequentially, assuming there are no more parallel tasks defined, but the interleaved order of execution of each concurrent subflow is undetermined. The transition from the join to the next element is only taken when all the subflows have finished successfully. Forks and joins have to be balanced: for each *fork* a corresponding *join* symbol closing the parallel tasks section should exist. Figure 4.4 shows the use of parallel tasks in a Task Flow diagram. After the task *Get bug report* terminates, two parallel flows are initiated in the diagram. *Fix bug* and *Rollout new release* are task executed sequentially in one flow, while *Log bug* is executed in the second flow. Each flow may finish independently and the execution of the *Notify client* task can be made just after the two flows have synchronised in the *join fork*.

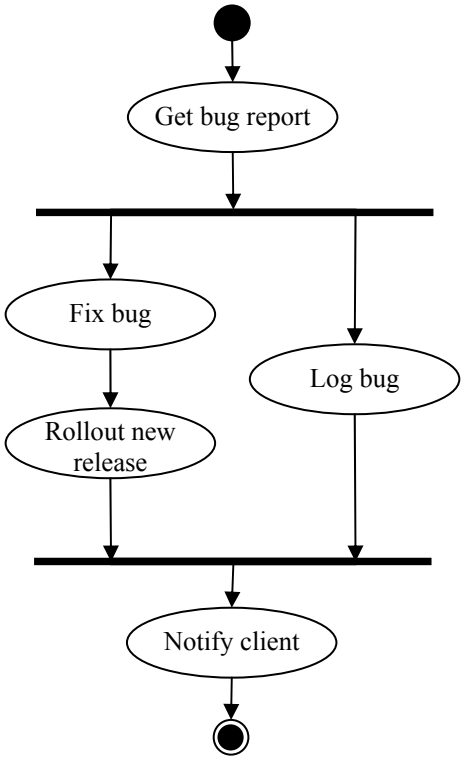

Figure 4.4 Example showing parallel tasks (Modified from [129])

A particular feature of the workflow model supported by the Discovery Method is the interaction of concurrent flows and task exit points. Both early success and early failure in one of several concurrent flows may result in the pre-emption of the other flows, a feature, which must later be modelled in the semantics.

4.2 The Alloy approach

Different techniques could be used to formalise the notation used by the Discovery Method. Chapter 2 mentions some of the formal methods considered in this research. In that chapter Alloy [34, 50], a language originally inspired by Z and based in first-order logic [49], was mentioned. In an initial experiment, Alloy was utilised to represent the abstract syntax of diagrams used in all of the Discovery Method and the abstract syntax tree was used to check some example concrete models [127], which were judged according to whether they conformed to the rules of the abstract syntax. An abstract syntax tree for all the diagrams in the Discovery Method was generated (see Figure 4.15).

4.2.1 Methodology

The abstract syntax was determined by examining each design model used in the Discovery Method in turn, then describing each model element and the constraints upon that element. Initially, there was some freedom to develop either a single abstract syntax, or a collection of syntaxes, one for each type of model.

Alloy contains certain built-in predicates that were useful when checking properties of the abstract syntax. For example, some models had the property of being directed acyclic graphs (DAGs). Provided that a relation could be constructed to generate the transitive graph, the built-in *dag()* constraint could be applied to this expression.

Successive versions of the abstract syntax specification were tested by proposing *check* assertions in Alloy, to check deliberate counterexamples which encoded violations of desired properties of the abstract syntax, for example that an *Object* is a composition (exclusive aggregation) of itself, recursively. When these were *checked*, Alloy would sometimes not detect the expected syntactic violation, indicating that the abstract syntax did not yet encode sufficient invariant properties to rule out malformed diagrams.

Later, when checking diagram instances against the abstract syntax, the model checking strategy was switched from using a refutation approach to using a predicate satisfaction approach, whereby diagram instances were encoded as predicates and the Alloy analyzer had to satisfy one instance of each predicate, to indicate that a diagram was valid. The reasons for this change are described in the evaluation of using Alloy as a diagram-checking tool below.

4.2.2 Abstract syntax

The experiment was geared toward the use of Discovery and Alloy, used as a supporting formal method, with the aim of defining the formal representation for Discovery.

The abstract syntax for the notations of the Discovery Method was coded in Alloy with the aim of facilitating the mapping between the notation and the semantic domain [27]. The refutation approach was used to test the specification. The abstract syntax model also included well-formedness rules or static semantics [130], which govern the correctness of Discovery models. Checking for well-formedness is traditionally made at a diagrammatical level using a BNF specification, but Alloy was used, trying to define the whole abstract syntax and looking for an appropriate representation of the model instances of Discovery and experimenting with the model checking supported by Alloy.

The project involved the construction of a unified abstract syntax for the five principal Discovery models (Task Structure, Task Flow, Object, State and Collaboration models), which includes well-formedness rules derived naturally from the diagram notations. The abstract syntaxes were combined for each model to have single definitions of the common elements with shared properties. A similar strategy for UML has been recommended by the 2U group in their UML 2.0 proposal [131]. Evans et al. actually propose two abstract syntaxes to support all the concrete syntax of UML [132], separating the abstract syntax describing structure from that describing behaviour. Figure 4.5 shows the chosen abstract syntax architecture, with four layers: the *System view*, the *Model view*, the *Diagram view* and the base level for the elements of Discovery notation.

The *System view* gives a complete representation of a specification, formed by a collection of models in the Discovery Method. This view includes at most one model of each kind and maintains the relationships between the different models. The *Model view* is used to define the different models supported by Discovery. At this level, each model has *n* diagrams and the *Model view* maintains the consistency between these different diagrams.

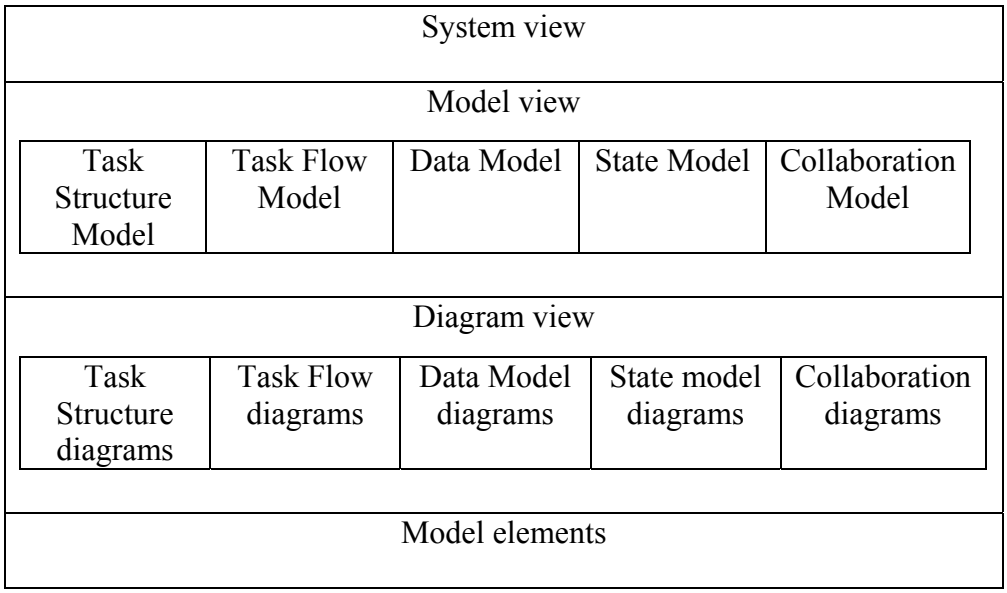

Figure 4.5 General structure of the abstract syntax

The *Diagram view* specifies single diagrams without concern for their interrelation, since the purpose at this level is to ensure that diagrams use the appropriate elements

of Discovery's notation. The lowest level is used to specify all the relevant elements of the Discovery notation and their basic relationships.

With this layering of models and diagrams, it is possible to check, at different levels of detail:

- Each diagram separately.
- Each model independently.
- The whole system specification.

4.2.3 Checking visual models with Alloy

Visual models are modelled in the abstract syntax and checked using the predicate satisfaction approach mentioned in section 4.2.1. The abstract syntax model supports the definition of generic syntax constraints, together with the specific constraints relating to a particular diagram, model or system. While the Alloy representation may be checked for all three views shown in Figure 4.5, the *Diagram view* must always be included, since this declares the relevant primitive elements. The strategy followed is to encode the general constraints for each type of diagram in one Alloy signature, and then to encode a specific diagram as a subtype signature in Alloy. The reasons for this are discussed below in section 4.2.4.

Figure 4.6 shows the signature *TaskStDiagramView*, defining the general properties of a *Task Structure Diagram* in Alloy. This basically declares the sets of elements that can possibly be part of the diagram. The relationships among these elements are defined at the lowest level of the abstract syntax graph (see the metamodel in Figure 4.15).

```
sig TaskStDiagramView extends DiagramView{
  task: set Task,
  goal: set Goal,
  gen: set Generalisation,
  real: set Realisation,
  agg: set Aggregation – Composition,
  comp: set Composition,
  actor: set Actor,
  obj: set Object - AssociationClass,
  parti: set Participation
}
```

Figure 4.6 Task Structure Diagram elements

Given the above, we may encode a specific *Task Structure Diagram*, such as the sketch of a Library's circulation system in Figure 4.7. The corresponding Alloy signature s*CirculationTS*, which represents the diagram instance, is given in Figure 4.8. This signature extends the basic *TaskStDiagramView* signature. Signature extension means that the derived signature has all of the properties of the base signature, similar to the notion of inheritance in object-oriented programming. In the upper declaration area, the particular elements of the diagram are declared. These are all expressed in terms of diagram element types inherited from the generic signature.

In the lower predicate area, constraints are defined on the declared elements. One constraint is used to specify the aggregation relationship, linking tasks to their corresponding source or target tasks in the structure. A similar constraint is created to define the participation linking the actor and the top-level task.

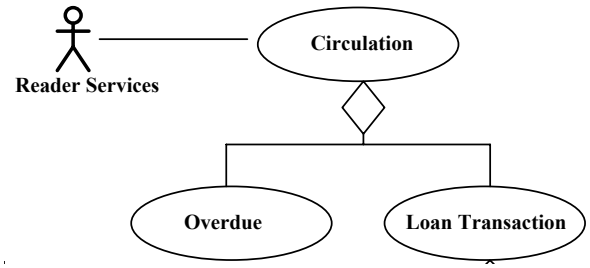

Figure 4.7 Circulation Task Structure Diagram

```
sig sCirculationTS extends
   TaskStDiagramView {
 part circulationTask, overdueTask,
   loanTransactionTask: task,
 readerServicesActor: actor,
 part p: parti,
 circAgg: agg
}{
 // Aggregation
 circulationTask in circAgg.head and
   overdueTask + loanTransactionTask
   in circAgg.tail
 #circAgg.tail=2
 // participation
 circulationTask in p.tact and
   readerServicesActor in p.user
}
```

Figure 4.8 Encoding the Circulation Task Structure Diagram

The abstract syntax for further *Task Structure Diagrams* may be specified. Figure 4.9, for example, shows a diagram that represents a more detailed elaboration of one of the tasks in the Task Structure diagram given in Figure 4.7. Eventually, the two independently-created diagrams should be made consistent within the same Task Structure model.

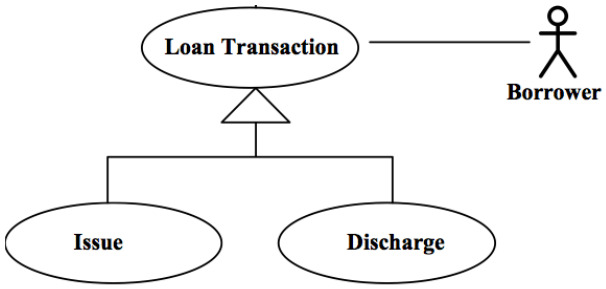

Figure 4.9 Loan Transaction Task Structure Diagram

The corresponding signature *sLoanTransactionTS* is shown in Figure 4.10. The specification is constructed in a similar way as before, but this time describes a generalisation instead of an aggregation relationship.

With these two definitions, there is enough information to check each diagram separately, to demonstrate that they each conform to the legal syntax of a *Task Structure Diagram*. However, it is more interesting to treat them as part of the same Task Structure model and check them together. To achieve this, a new Alloy specification must be constructed, representing the *Model view*, within which the two diagrams are merged on their common element (the *Loan Transaction* task).

```
sig sLoanTransactionTS extends
   TaskStDiagramView {
 part loanTransactionTask, issueTask,
   dischargeTask: task,
 borrowerActor: actor,
 p: parti,
 loanGener: gen
}{
// generalisation
loanTransactionTask in loanGener.head
  and issueTask + dischargeTask in
  loanGener.tail
#loanGener.tail=2
// participation
loanTransactionTask in p.tact and
  borrowerActor in p.user
}
```

Figure 4.10 Encoding the Loan Transaction Task Structure diagram

Figure 4.11 shows the signature *sCirculationModel*, representing a particular Task Structure model for the whole circulation subsystem, which merges the above diagrams consistently. The signature extends a generic *TaskStModel* signature (whose detail is not given here) and specifies that the diagrams *sCirculationTS* and *sLoanTransactionTS* are part of the model. All the information pertaining to the *Model view* is inherited from *TaskStModel* and the individual diagrams were specified above in the *Diagram view*, so apart from linking the diagrams to the model, there is only a need to assert which elements are common to both diagrams. For instance, the last clause in Figure 4.11 defines the fact that the intersection between the tasks from the Circulation Task diagram and the Loan Transaction diagram is equal to the Loan Transaction task.

```
sig sCirculationModel extends TaskStModel {
}{
sCirculationTS in tm
sLoanTransactionTS in tm
sCirculationTS.loanTransactionTask
  = LoanTransactionTS.loanTransactionTask
sCirculationTS.task &  sLoanTransactionTS.task
 = sLoanTransactionTS.loanTransactionTask
}
```

Figure 4.11 Encoding the Task Structure model

Having defined a particular Task Structure model consisting of two Task Structure diagrams, it is possible to check the consistency of these against the rules of the abstract syntax. In the following, the second of the two checking strategies is used, instantiating a predicate, rather than the model refutation approach. If Alloy cannot find a valid instance, this will mean that our model does not conform to all the syntax constraints defined for the Discovery notation. Figure 4.12 illustrates the Alloy code that is executed to validate the model. This consists of a dummy predicate *circulationModel()* which is run for an exactly-specified scope, within which Alloy must find all the elements of the model. The scope is an enumeration of each element and relationship used in the model under test. In a smaller scope Alloy cannot generate a valid instance, whilst in a larger scope Alloy will create additional elements, making the instance valid with the whole abstract syntax, but not equivalent to our model. Indicating the exact scope is necessary if satisfaction is to be interpreted as validating the model. However, this also has the useful effect of limiting the state space searched by Alloy for a valid instance.

```
pred circulationModel(){}
run circulationModel
 for 1 but
 exactly 1 Model,
  exactly 1 TaskStModel,
    exactly 1 sCirculationModel,
 exactly 2 DiagramView,
    exactly 2 TaskStDiagramView,
     exactly 1 sLoanTransactionTS,
     exactly 1 sCirculationTS,
 exactly 4 Relationship,
    exactly 2 Structure,
      exactly 1 Generalisation,
     exactly 1 Aggregation,
    exactly 2 Participation,
 exactly 7  Node,
    exactly 5 StateAndTask,
      exactly 5 TaskActivity,
       exactly 5 Task,
    exactly 2 Actor,
 0 Transition,
 0 TaskFlowElement,
 0 Member
```

Figure 4.12 Empty predicate and exact scope specified for the run command

When the above *run* command is executed, Alloy finds the unique instance, indicating that our example is in fact consistent with the Discovery abstract syntax. What Alloy does is to satisfy the empty predicate (a trivial task in itself) in conjunction with making the particular diagram and model specifications consistent with the general syntax specifications, within a scope that only has one possible solution, if any. Alloy presents its result either as a graph of linked signature instances (similar to the metamodel graph in Figure 4.15), or as a browseable tree (as shown in Figure 4.13).

Figure 4.13 shows the tree view generated by Alloy for the example presented above, whose structure can be inspected interactively if you want to examine the result. The

fact that Alloy finds an instance at all demonstrates that the example is valid. If no result is returned, this means that the tested model is invalid.

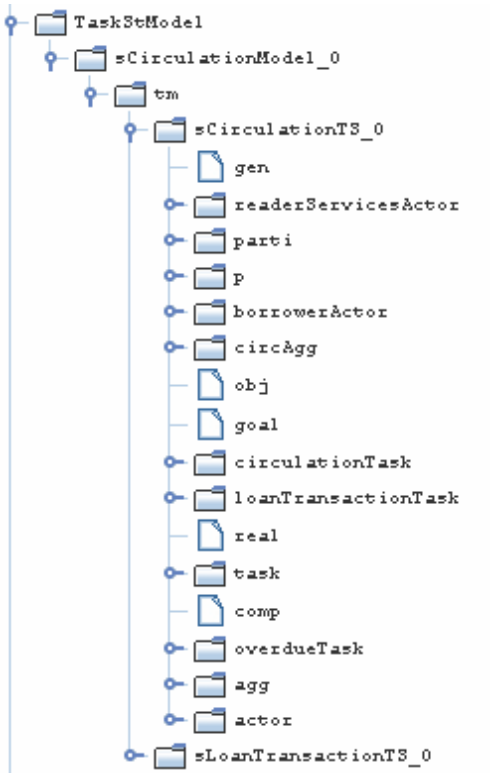

Figure 4.13 Solution generated by Alloy

Figure 4.14 illustrates a second interesting example, for which we would expect no consistent solution to be found by Alloy. It is possible to verify that the individual exemplar diagrams (a) and (b) are syntactically correct in the *Diagram view*, but when both diagrams are included within the same data model in the *Model view*, Alloy cannot find a valid instance. This is because the Z class is defined as a component of two different classes in the same model, something which violates the specification for a UML *composition*, which requires the composed elements to be uniquely-owned parts of the whole.

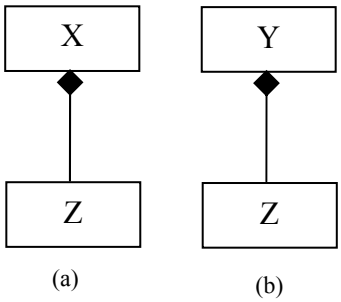

Figure 4.14 Two diagrams creating an inconsistent Data Model

4.2.4 Evaluating Alloy

While Alloy is very effective in modelling and analysing simple, lightweight formal specifications written in a Z-like style, this experiment found that it is more difficult to use as the basis for model checking the syntax and static semantics of a design

notation. At various times, the construction of the specification was forced into work-arounds to constrain the searching behaviour of the analyzer. The following gives a flavour of some of the unexpected discoveries while modelling in Alloy.

Initially, a separate abstract syntax for each type of model used in the Discovery Method was developed. Therefore, for example, the Task Structure model had distinct generalisation and aggregation relationships from those in the Data model, although in the Discovery Method these are each single kinds of relationship, with a uniform semantics across all model types. This meant that the Alloy signatures for *Generalisation* and *Aggregation* were short and the scopes, within which model instances were checked, were quite small. However, when models of different types were combined, this required a set of translations from one abstract model syntax to another.

In the second version, all the abstract syntaxes for the different model types were unified, such that a single *Aggregation* relationship existed for all types of model. This was more in keeping with the philosophy of the Discovery Method. However, the Alloy signature for *Aggregation* was made more complicated by the need to assert extra constraints that it either related two *Tasks*, or two *Objects* and not one of each. Alloy lends itself to creating hierarchies of disjoint subtypes in its abstract syntax, using the *extends* notation. This initially fostered a meta-modelling style of construction, whereby all syntax elements descended from a common *ModelElement* root, similar to the MMF [6]. However, this had the unexpected consequence of requiring vastly larger scopes within which to search for model instances, since Alloy interprets all scope instructions as relating to the base instances in any tree. As a necessity, the syntax tree was broken down into a series of shorter trees (see Figure 4.15), losing the abstraction over all model elements.

Once the abstract syntax had been fully validated using *check* assertions, Alloy representations of diagram instances were developed. Initially, a diagram instance was represented as an Alloy *predicate*, to be evaluated against generated instances of the abstract syntax. Eventually, this proved to be unwieldy, requiring the repetition of constraints whenever a part of the predicate referenced the same sub-elements in the diagram. In the second version, diagram instances were constructed as subtypes of the canonical abstract syntax types, a strange but economical encoding, which avoided such repetition of constraints. The eventual predicate to check was then trivial (empty), since all the analyser had to do was find one instance of the diagram itself. To control this, the scope was set to generate exactly one instance of each model element present in the diagram, a brute force approach to ensure that Alloy did not over-generate elements of the diagram. If the search to satisfy the trivial predicate generated a single matching instance of the diagram, then this represented success in satisfying the abstract syntax. The execution was able to find single instances of consistently-merged diagrams. The attempt to find an instance of mutually inconsistent diagrams failed, as expected, although no useful information could be reported about the detected inconsistency.

4.2.5 Conclusions on the Alloy approach

To summarise this section, experiences using the Alloy analyzer to check an abstract syntax for the notation of the Discovery Method were presented. This section described how the experiment used different approaches to design the abstract syntax

and to represent the diagram instances in Alloy, commenting on the naturalness, or otherwise, of the chosen encodings.

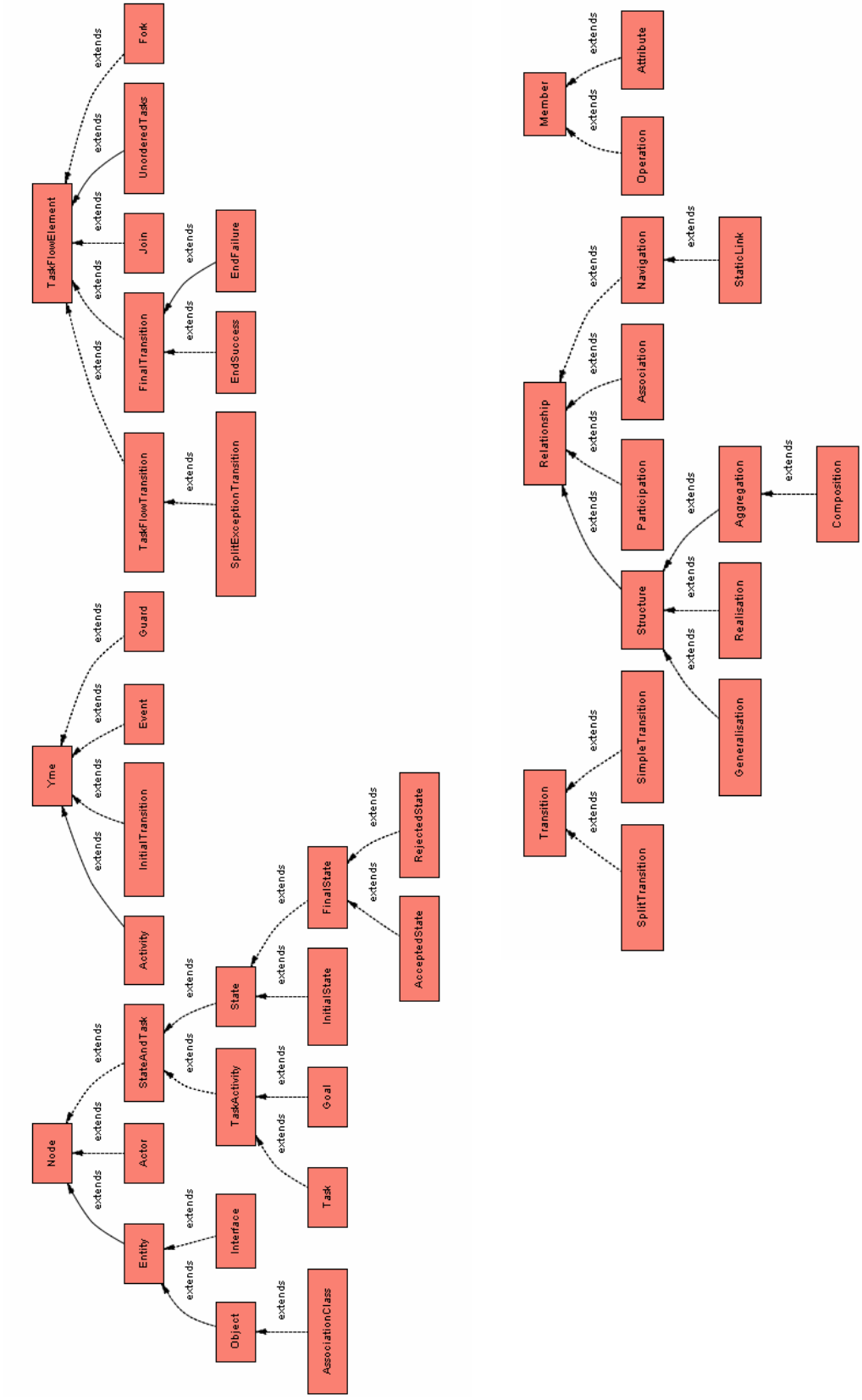

Figure 4.15 Abstract syntax metamodel

A complete example of a valid model for Discovery (a Task Structure model) and the result generated by Alloy were illustrated, showing that the basic approach is feasible. The time taken to validate larger models with an exact scope is in the order of minutes. In addition, a counter-example of an invalid model (a Data model) was illustrated, for which Alloy correctly found no instance.

Additionally, impressions of Alloy as a candidate tool for checking the consistency of multiple diagrams in software engineering notations were given. The feeling is that this is perhaps not an ideal deployment of Alloy. The searching behaviour of the constraint solver had to be carefully controlled. The notion of a single hierarchy of model elements in the abstract syntax specification was abandoned, since this gave rise to underconstrained instance generation.

As was mentioned above, even when we were able to represent the Task Structure model in Alloy, the results were not what we expected. In addition, in the objectives of the research it was established that we wanted to represent the semantics for the Task Flow diagrams in term of traces, so a process algebra style suited better for this kind of diagram.

4.3 From the Task Flow Diagram to the Task Algebra

In order to give a formal representation for the *Task Flow Diagram*, the following chapter will present a Task Algebra, which is first introduced here. The Task Algebra syntax is a straight translation from the *Task Flow Diagram*, the abstract representation is almost a direct copy of the diagram notation. There are structures for every basic structure allowed in the *Task Flow Diagram*: sequence, selection, parallelism, and repetition. Additionally encapsulation is considered as a formal structure for delimiting the scope of the diagrams.

The syntax for this algebra and its axioms will be explained in detail in chapter 5, followed by the semantics in chapter 6. In this section, the correspondence between the *Task Flow Diagrams* and the Task Algebra is explained by comparing simple examples with the algebra notation. The idea is to introduce the algebra as well as showing how clear the translation between the diagram and the algebra is.

4.3.1 Sequence of tasks

Figure 4.16, for instance, depicts a sequence of tasks. The equivalent expression in the Task Algebra for the task model is *{a; b; c}*.

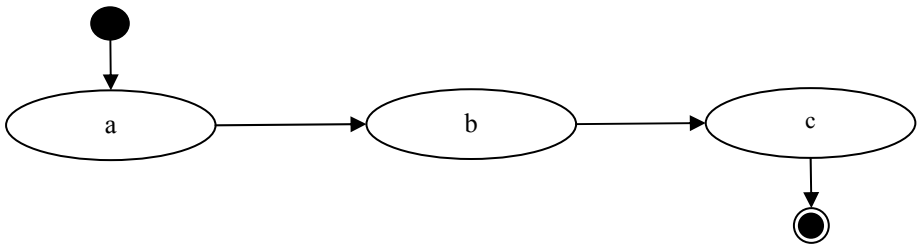

Figure 4.16 Sequence of tasks in the Task Flow Model

As can be seen, the start and end elements of the diagram have no direct representation because they are implicit from the scope of the curly brackets. While in the *Task Flow Diagrams*, order of execution of the tasks is specified by the arrows, in the Task Algebra the order of execution in sequences depends of the position of the task name in the expression. Tasks are executed from the left to the right and separated by semicolons.

4.3.2 Selection

Figure 4.17 depicts the choice among three tasks *a*, *b*, and *c*. This figure shows the selection symbol that can be used when two or more choices are needed. Guards are mutually exclusive and exhaustive. The *Task Flow Diagram* showed in Figure 4.17 may be presented in the Task Algebra as the following expression: *{a+b+c}*. As can be seen, the selection is represented by the symbol '+', and the conditions are not represented in the algebra.

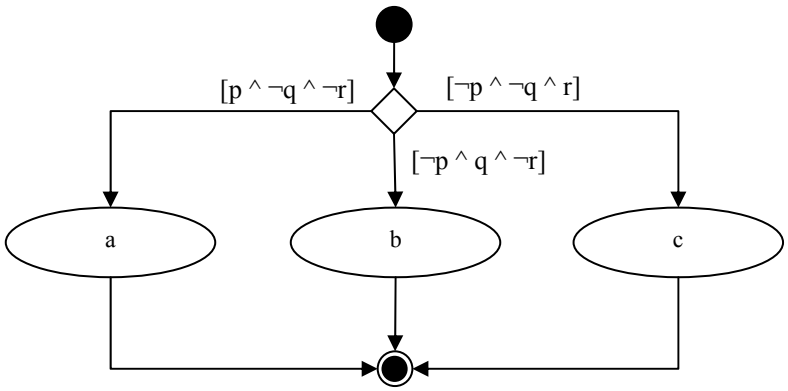

Figure 4.17 Selection in the Task Flow Diagram

The binary selection in Figure 4.18 is also represented with the symbol '+'. The guards in the binary selections are also mutually exclusive and exhaustive and, in consequence, one of the guards could be omitted. Figure 4.18a shows a normal binary selection where in case of *p* the task *a* could be reached, and task *b* could be reached if ¬*p*. *{a+b}* is the matching representation of Figure 4.18a in the Task Algebra. Figure 4.18b introduces the *fail* symbol using a small circle with a diagonal line crossing it. This symbol represents the exit with failure of the execution in the flow. *{ϕ + x}* is the matching representation of Figure 4.18b in the Task Algebra, where *ϕ* is for *fail* in the algebra. The *fail* symbol is employed in association with the half-diamond utilised to represent exceptions.

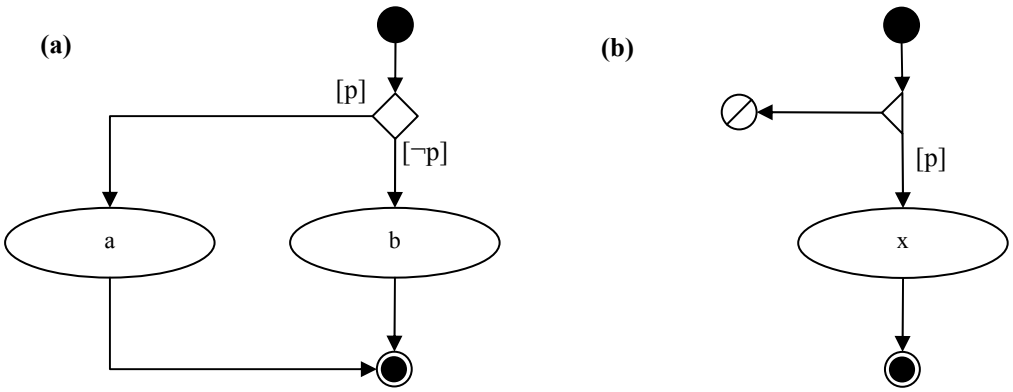

Figure 4.18 Binary selection in the Task Flow Diagram

It was mentioned before that guards have no direct representation in the abstract syntax, because they are initially not relevant. The same could be argued about the *start* and *end* symbols, which might initially be encoded implicitly in the structure of algebraic constructions. However, it is later found necessary to represent end symbols explicitly, because of the different behaviour of normal exit and failure. *Fail* is used to specify an end inside a context, but gives additional information about the global failure of the execution of the flow. A normal end is considered as success and, if it is necessary to include it explicitly in the expression, the σ symbol called *succeed* is used. The σ symbol is needed if there is a requirement to represent a race-condition (multiple parallel threads with one of them expecting to win); also is useful to express an unary selection using the binary selection from the algebra. However, probably the most important difference is that, while σ indicates the success of the execution of the actual diagram, ϕ represents a failure in the execution of the flow from that point and beyond. This means that even if the diagram is part of another diagram, no further tasks are executed after a failure.

4.3.3 Repetition

Repetition is shown in Figure 4.19 in the form of until- and while-loop. Like in the selection, the guard is implicit and there is not direct representation. Figure 4.19a shows the repetition of the task *a* until *p* was false. Its corresponding expression in the Task Algebra is $\mu x.(a\ ;\ \varepsilon + x)$. The while-loop is shown in Figure 4.19b and its representation in the algebra is $\mu x.(\varepsilon + a\ ;\ x)$. In the Task Algebra, ε represents the empty activity.

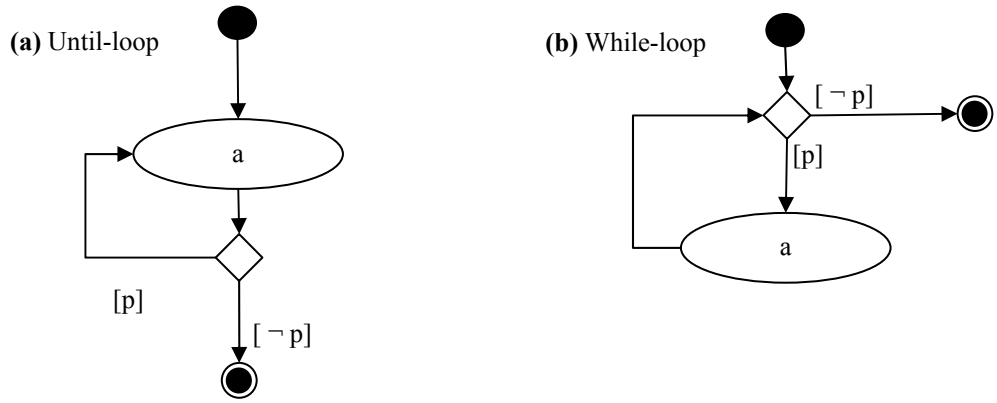

Figure 4.19 Repetition in the Task Flow Model

4.3.4 Parallel composition

Parallel composition is shown in Figure 4.20. The diagram depicts two sequential compositions executing in parallel. The Task Algebra expression for this *Task Flow Diagram* is *{(a; b) || (c; d)}*.

In the next section, an example is depicted with its equivalent expression in the Task Algebra.

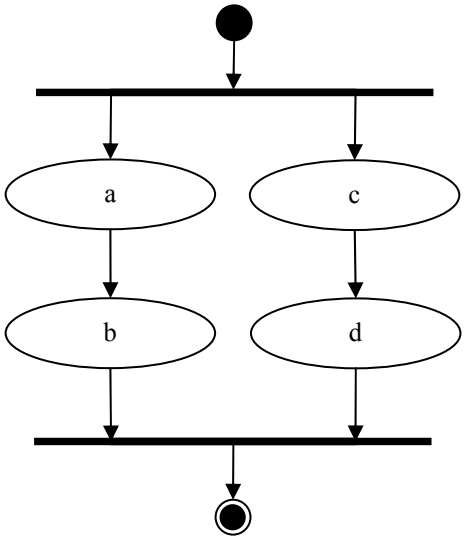

Figure 4.20 Parallel composition in the Task Flow Diagram

4.3.5 The eating routine example

In order to appreciate the translation from the *Task Flow Diagram* to the Task Algebra an example is shown in Figure 4.21. The diagram represents the flow of tasks for having dinner and choosing between cooking it in the house or going out.

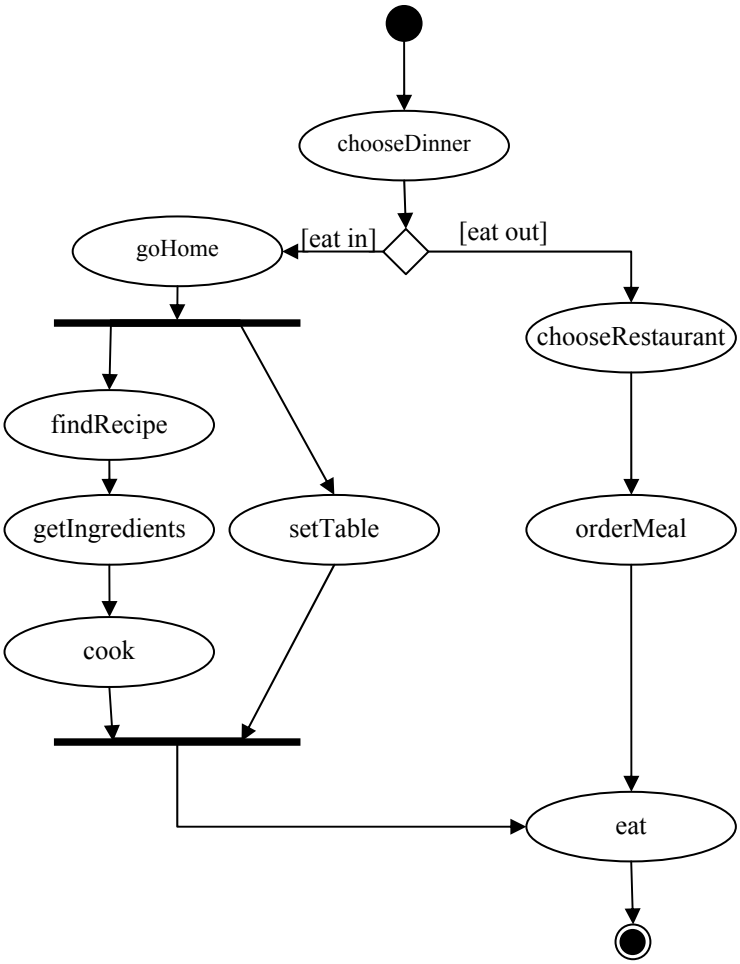

Figure 4.21 Task Flow Diagram showing the process of doing dinner

As can be seen, the diagram uses a combination of sequences, a choice, and concurrent tasks. Almost all of this information can be represented in the Task Algebra. The only elements that are not translated into the algebra are the guard conditions on each branch. The expression in the Task Algebra may be as follows:

```
chooseDinner;(goHome;((findRecipe; getIngredients; cook)
|| setTable)) + (chooseRestaurant; orderMeal); eat
```

In this case, it is important to mention that expressions in the Task Algebra are right-associative but parentheses are allowed where this is considered necessary. In this example, the sequence of three tasks (*findRecipe*, *getIngredients*, and *cook*), being part of the parallel composition, is grouped using parentheses. The same bracketing approach is used to cluster the tasks on each side of the selection operator. Start and exit with success are implicit but the scope of the enclosing task (which we shall call *Dinner*) is specified explicitly using curly brackets:

```
{ chooseDinner;(goHome;((findRecipe; getIngredients;
cook) || setTable)) + (chooseRestaurant; orderMeal); eat
}
```

The use of curly brackets defines the boundary of tasks and affects the scope of the behaviour of early exit tokens *success*, and *fail*, which jump to the nearest boundary.

Therefore, the behaviour of an expression enclosed in curly brackets will typically differ from one which is not so enclosed. The other effect of bracketing flows in this way is to define larger composite tasks. Now, supposing that the expression above is bracketed, this may be named as a compound task called *Dinner* and then used as part of another *Task Flow Diagram*, as shown in Figure 4.22.

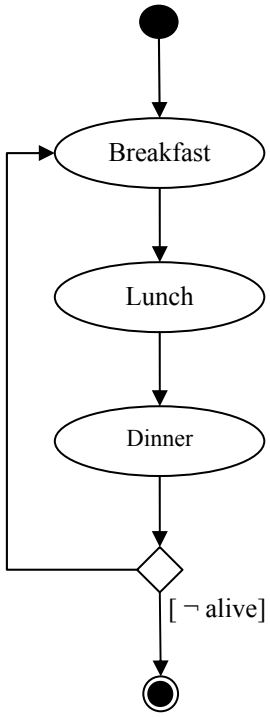

Figure 4.22 Until-loop repetition in the Task Flow Diagram

The diagram above uses the repetition structure containing a sequence of the tasks: *Breakfast*, *Lunch*, and *Dinner*. The execution of the sequence is guaranteed at least once because the structure is an until-loop setting the condition at the end. The alternative construction, a while-loop, it is also allowed by the *Task Flow Diagram*. The corresponding expression for Figure 4.22 in the Task algebra should be:

```
μx.( (breakfast; lunch; dinner); ε+x)
```

The sequence has to be between parentheses because the syntax, explained in detail in the next chapter, is defined for one *Activity* and the right-associativity will derive in a syntax error for no atomic tasks.

4.4 Summary

This chapter explained the informal semantics for the *Task Structure* and *Task Flow Diagrams*, as well as an introduction to the approximation used to define the formal semantics. Additionally, an experimental approach using Alloy to define and verify the abstract syntax of the diagrams in the Discovery Method is discussed, but this approach was eventually abandoned, due to problems in restricting the scope of the analyzer. The next chapter will present formally the abstract syntax and chapter 6 will explicate the formal semantics for the *Task Flow Diagrams* in the Discovery Method.

Chapter 5:
An Abstract Syntax Representation for the Task Flow Model

The previous chapter presented the informal semantics for the Task Structure and Task Flow Diagrams. In this chapter the abstract task algebra for the Task Flow Model is presented. The definition of the task algebra is initially depicted in Backus Naur form. Subsequently, the set of axioms constraining the initial definition are given and some examples are provided in order to show how the algebra works as expected for basic elements. The abstract task algebra is based in simple and compound tasks using operators such as sequence, selection, and parallel composition. Repetition is defined with recursion in the form of while- and until-loops.

5.1 Introduction

A simple task in the Discovery Method [4, 5, 63] is the smallest unit of work with a business goal. In this context tasks are categorised as simple and compound tasks. A simple task is the minimal representation of a task in the model. A compound task can be formed by either simple or compound tasks in combination with operators defining the structure of the Task Flow Model.

Simple tasks are understood to consist of a series of steps and therefore are always considered to be intervals, from a temporal perspective. The internal atomic steps of the simple tasks are not relevant for the Task Flow Model and for that reason this information is not present in the abstract syntax representation.

In addition to simple tasks and compound tasks, the abstract syntax also requires the definition of three instantaneous events. These may form part of a compound task in the abstract syntax.

5.2 The Abstract Syntax

The basic elements of the abstract syntax are the simple task, which is defined using a unique name to distinguish from others; ε representing the empty activity; and the success σ and failure ϕ symbols representing a finished activity.

Simple and compound tasks are combined using the operators that construct the structures allowed in the Task Flow Model. The basic syntax structures for the Task

Flow Model are sequential composition, selection, parallel composition, repetition, and encapsulation:

- **Sequential composition** defines the chronological order of execution for a task or a group of tasks from the left to the right and ';' is used as the operator.
- **Selection** is represented with the symbol '+' and it means that there is a choice between the operands.
- **Parallel composition** defines the simultaneous execution of the elements in the expression. It is represented by the symbol '||'.
- **Repetition** allows the reiteration of an expression in the form of an until-loop and while-loop structure. It is represented using the μx fixpoint.
- Finally, **encapsulation** is used to group a set of tasks and structures. This constructs a compound task and is represented using curly brackets *'{' '}'*.

The abstract syntax has the following definition in Backus Naur form:

```
Activity  ::= ε                    -- empty activity
        | σ                        -- succeed
        | ϕ                        -- fail
        | Task                     -- a single task
        | Activity ; Activity      -- a sequence of activity
        | Activity + Activity      -- a selection of activity
        | Activity || Activity     -- parallel activity
        | μx.(Activity ; ε + x)    -- until-loop activity
        | μx.(ε + Activity ; x)    -- while-loop activity

Task::= Simple                     -- a simple task
        | { Activity }             -- encapsulated activity
```

A task can be either a simple or a compound task. Compound tasks are defined between brackets '{' and '}', and this is also called encapsulation because it introduces a different context for the execution of the structure inside it. Curly brackets are used in the syntax context to represent diagrams and sub-diagrams but also have implications for the semantics that will be explained later. Also, parentheses can be used to help comprehension or to change the associativity of the expressions. Expressions associate to the right by default.

The abstract syntax represents in a simple way every basic structure used for the Task Flow Diagram. For instance, supposing three tasks *a*, *b* and c; a sequence composition of these elements can be specified as follows:

$$a;b;c$$

Which means the execution of *a*, then *b*, and then *c*. The selection operator '+' should be used for representing the choice among tasks:

$$a + b + c$$

The concurrent execution of these three tasks may be represented using the parallel composition operator '||':

$$a \parallel b \parallel c$$

Meaning that *a*, *b* and *c* are executed simultaneously and may terminate in any order. Finally, the repetition operator works either as an until-loop or a while-loop. The difference between each repetition is, as can be supposed, that the until-loop structure guarantees at least one execution of the activity in the repetition:

$$\mu x.(a; \varepsilon + x)$$

Repetition is modelled using recursion. In the example above, μx binds x to the expression $(a;\varepsilon+x)$, where a occurs at least once and, if under the choice of x, the expression is expanded (i.e. the expression is repeated recursively, x being the fixed-point of bound by μ.) The next example shows a while-loop:

$$\mu x.(\varepsilon + a; x)$$

As in the until-loop, μx binds x to the expression $(\varepsilon+a; x)$, but the choice is put in front of the expression to be repeated.

5.3 Task Model Constructions

Just as the graphical structures of the Task Flow Model can be composed, basic definitions in the abstract syntax may form complex expressions. The abstract syntax definition can be considered like a Universal Algebra which, to accomplish an accurate representation of the diagram syntax, has to be limited by axioms. The abstract syntax definition and its axioms form an Ideal or Quotient Algebra.

5.3.1 Simple task

As it was explained before, a simple task is the minimal representation of a task in the abstract syntax with significance for the expressions; whilst an Activity is formed using a combination of operators (sequence, selection parallel composition, and repetition), simple tasks, empty activity, end with success and end with fail. Empty and finished activities are vacuous activities. Empty is represented with ε, success with σ and fail using ϕ. The fact that simple tasks cannot be vacuous activities is formalised in the next axioms:

(sp.1) $\forall a \in Simple \bullet a \neq \varepsilon \wedge a \neq \phi \wedge a \neq \sigma$

(sp.2) $\forall a \in Simple \bullet \forall y, z \in Activity \bullet a \neq (y; z) \wedge a \neq (y + z) \wedge a \neq (y \parallel z) \wedge a \neq \mu x.(y; \varepsilon + x)$

Simple tasks are different from *succeed*, *fail* and empty activities because simple tasks represent processes with interval duration different from zero. *Succeed*, *fail* and empty activities are considered instantaneous events.

5.3.2 Empty activity

As was said before, the symbol ε is used to represent the empty activity. It is needed because the selection is a binary operator and ε is used to characterise the empty branch, where in combination with the selection operator it is used as the choice between doing something or nothing.

As a result of the existence of this element, a set of axioms must be defined to interpret the meaning of the empty activity when it is a part of other kinds of expression. These rules are specified within each operator description.

5.3.3 Finished activity

The finished activity is necessary to represent situations when an activity should terminate before the normal end. The abstract syntax representation allows two kind of finished activity: *succeed* and *fail*. *Succeed* is useful to represent an early exit from within an expression, returning the control to the higher scope. On the other hand, *fail* is used to represent the termination of all tasks, and the failure is propagated to the higher levels.

σ and ϕ are considered instantaneous events. Similar to the empty activity, the finished activities have an effect in many operator constructions, and they will be defined later.

5.3.4 Sequential composition

Sequential composition is defined as the consecutive execution of activities, from the left to the right. Tasks are separated by ';'. For example:

$$a;b;c;d \Leftrightarrow a;(b;(c;d))$$

The intuitive meaning is that first *a* will be executed, then *b*, and so on until the task *d*. Parentheses can be used to group elements but the meaning is not altered whatsoever. An associative axiom is defined to support this notion. Axioms for distribution, empty sequence and finished activity are also defined. Commutativity and idempotence properties are not considered for sequences:

(s.1) $\forall a,b,c \in Activity \bullet a;(b;c) \Leftrightarrow (a;b);c$ -- associative sequence

(s.2) $\forall a,b,c \in Activity \bullet (a+b);c \Leftrightarrow (a;c)+(b;c)$ -- right distributivity of sequence over selection

(s.3) $\forall a \in Activity \bullet a;\varepsilon \Leftrightarrow \varepsilon;a \Leftrightarrow a$ -- empty sequence

(s.4) $\forall a \in Activity \bullet \phi;a \Leftrightarrow \phi$ -- *fail* on sequence

(s.5) $\forall a \in Activity \bullet \sigma;a \Leftrightarrow \sigma$ -- *succeed* on sequence

Rule (s.2) defines that a right sequence is distributed over a left selection. Left distribution of sequence over selection is not allowed because, as in ACP [69, 133], left sequence distribution changes the point where the choice is made. It follows that:

$\forall a, b, c \in Activity \bullet a;(b+c) \neq (a;b)+(a;c)$

Because in the expression *a;(b+c)*, initially *a* is executed and then the choice between *b* and *c* is made; while in the expression *(a;b)+(a;c)* the choice is first and afterwards *a* is executed. The difference in the branching position can be easily appreciated in Figure 5.1.

Figure 5.1. State transition diagram for expressions a;(b+c) and (a;b)+(a;c)

Empty and finished activities may coexist in an expression, in which case the rule (s.3) is confluent with rules (s.4) and (s.5) (i.e., the expression can be reduced in another way) and may interact. All posible basic cases where these rules can be used confluently are shown as follows:

a) $\phi;\varepsilon$

$\Rightarrow \phi$ -- by applying (s.3) or (s.4)

b) $\sigma;\varepsilon$

$\Rightarrow \sigma$ -- by applying (s.3) or (s.5)

5.3.5 Selection

The selection of activities is performed with the '+' operator. It represents the choice among a group of activities, for instance:

$a+b+c+d \Leftrightarrow a+(b+(c+d))$

Intuitively each branch is evaluated from the left to the right. Guards are implicit and are not represented in the syntax. The guards are supposed to be mutually exclusive and exhaustive. When a guard is satisfied the left activity is executed and the right branch is discarded, otherwise the left activity is discarded and the next guard is verified. Logically, the last guard does not need to be checked and the order in which the branches are considered is irrelevant.

The '+' operator will be preserved in the semantics in the form of a commit symbol '↓', which was included to represent the commit point for a selection. This is already a simplified version of the selection, where the guard conditions are not present but mentioned as a future extension for the algebra. Even so, we think this approach

represents better the flow of tasks, making clear where the selection happened. Figure 5.1 above presented a case where the difference in the branching position can be seen.

The axioms defined for the selection operator are:

(sel.1) $\forall a,b,c \in Activity \bullet (a+b)+c \Leftrightarrow a+(b+c) \Leftrightarrow a+b+c$

-- Associative selection

(sel.2) $\forall a,b \in Activity \bullet a+b \Leftrightarrow b+a$ -- commutative selection

(sel.3) $\forall a \in Activity \bullet a+a \Leftrightarrow a$ -- idempotent selection

In the case of the empty activity, it is also possible to reduce the expression if both sides have the empty activity by the idempotent rule (sel.3). But, if just one of the elements (right or left) is ε, then the selection has no reductions. $\forall a \in Activity$, the following expressions are irreducible:

$a+\varepsilon$ -- irreducible selection of empty activity or activity

The same applies to the finished activities, where the selection between any of the finished activities or a general *Activity* has no reduction:

$\phi + a$ -- irreducible selection of *fail* or activity

$\sigma + a$ -- irreducible selection of *succeed* or activity

$\varepsilon + \phi$ -- irreducible selection of empty activity or *fail*

$\varepsilon + \sigma$ -- irreducible selection of empty activity or *succeed*

As described above, selection interacts with sequences and the right distributivity axiom may be applied. Its interaction with parallel composition is shown below.

5.3.6 Parallel composition

Parallel composition is defined as the simultaneous execution of all its tasks and it is represented with the operator '||'. An example is the expression:

a || b || c || d $\Leftrightarrow$ a || (b || (c || d))

Intuitively it expresses that the elements *a*, *b*, *c*, and *d* are initiated at the same time and executed simultaneously. The end of any of them is non-deterministic. Like the last operators, a set of axioms are defined:

(p.1) $\forall a,b,c \in Activity \bullet (a \,||\, b) \,||\, c \Leftrightarrow a \,||\, (b \,||\, c)$

-- Associative parallel composition

(p.2) $\forall a,b \in Activity \bullet a \,||\, b \Leftrightarrow b \,||\, a$ -- Commutative composition

(p.3) $\forall a, b, c \in Activity \bullet (a + b) \,||\, c \Leftrightarrow (a \,||\, c) + (b \,||\, c)$

-- right distributivity of concurrency over selection

(p.4) $\forall a \in Activity \bullet a \,||\, \varepsilon \Leftrightarrow a$ -- instant synchronisation

(p.5) $\forall a \in Activity \bullet a \,||\, \phi \Leftrightarrow \phi$ **if** a ≠ σ -- instant failure

(p.6) $\forall a \in Activity \bullet a \,||\, \sigma \Leftrightarrow \sigma$ -- instant success

The associative and commutative axioms (p.1, p.2) reflect the nondeterministic order of concurrent activity. Also, it is possible to do right and left distribution of concurrent composition over selection, but only the one axiom is necessary. Right distribution over selection is defined in (p.3) and left distribution is derived by applying (p.1) and (p.3):

$\forall a, b, c \in Activity \bullet a \,||\, (b + c) \Leftrightarrow (a \,||\, b) + (a \,||\, c)$ -- left distribution of concurrency over selection, by axiom (p.1) and (p.3)

In both, the axiom and the derived rule, simultaneously may occur any of the activity elements *a*, *b* or *c*.

The use of instant events such as ε, σ and ϕ may occur too in combination with parallel composition. Axioms (p.4), (p.5) and (p.6) define instant synchronisation, *fail* and *succeed* respectively. Whilst (p.4) performs the elimination of ε whether it is on the right or the left of the parallel operator, (p.5) and (p.6) establish that any activity in parallel composition with *fail* or *succeed* is equivalent to just itself. Although the parallelism is resolved as the simultaneous execution of simple activities (i.e. concurrency between a single task and an *Activity* means that the single task could occur at any time among all the simple actions of such Activity), *Succeed* and *fail* are considered as instantaneous events and they have priority over the elements of the *Activity*. In addition, *succeed* has a major priority than *fail*, therefore in the case of a parallel composition between these two elements *succeed* will prevail (p.6). *Succeed* and *fail* are considered as instantaneous since it was wanted to differentiate them from a time consuming task, for this reason it was necessary to give them preference over the tasks. The decision of having priorities on some elements prevents our algebra from satisfying the interleaving semantics. In particular, we can say the algebra does not satisfy the expression:

```
a||b = (a;b) + (b;a)
```

where *a* and *b* can be any valid symbol in the algebra. With the non-interleaving semantics the difference between the elements can be represented.

Logically, this set of rules is confluent, which can be easily proved. The specific cases involving the symbols σ, ϕ and ε can be resolved using any of the rules defined for each symbol to work with parallel composition. All posible basic cases where these rules are used confluently are shown as follows:

a) $\phi \,||\, \varepsilon$
$\Rightarrow \phi$ -- by (p.4)
or
$\phi \,||\, \varepsilon$
$\Rightarrow \varepsilon \,||\, \phi$ -- by (p.2)
$\Rightarrow \phi$ -- by (p.5)
b) $\varepsilon \,||\, \phi$
$\Rightarrow \phi$ -- by (p.5)
or
$\varepsilon \,||\, \phi$
$\Rightarrow \phi \,||\, \varepsilon$ -- by (p.2)
$\Rightarrow \phi$ -- by (p.4)

c) $\sigma \,||\, \varepsilon$
$\Rightarrow \sigma$ -- by (p.4)
or
$\sigma \,||\, \varepsilon$
$\Rightarrow \varepsilon \,||\, \sigma$ -- by (p.2)
$\Rightarrow \sigma$ -- by (p.6)
d) $\varepsilon \,||\, \sigma$
$\Rightarrow \sigma$ -- by (p.6)
or
$\varepsilon \,||\, \sigma$
$\Rightarrow \sigma \,||\, \varepsilon$ -- by (p.2)
$\Rightarrow \sigma$ -- by (p.4)

The next ones are examples involving instantaneous events showing equivalent expressions:

$\forall a,b \in Activity \bullet (a \,||\, b) \,||\, \varepsilon \Leftrightarrow a \,||\, (\varepsilon \,||\, b) \Leftrightarrow a \,||\, b \,||\, \varepsilon \Leftrightarrow a \,||\, b$

-- by (p.1) and (p.4)

$\forall a \in Activity \bullet a \,||\, \varepsilon \Leftrightarrow \varepsilon \,||\, a \Leftrightarrow a$ -- by (p.2) and (p.4)

$\forall a \in Activity \bullet a \,||\, \phi \Leftrightarrow \phi \,||\, a \Leftrightarrow \phi$ **if** a ≠ σ -- by (p.2) and (p.5)

$\forall a \in Activity \bullet a \,||\, \sigma \Leftrightarrow \sigma \,||\, a \Leftrightarrow \sigma$ -- by (p.2) and (p.6)

$\forall a,b \in Activity \bullet (a \,||\, b) \,||\, \phi \Leftrightarrow a \,||\, (\phi \,||\, b) \Leftrightarrow a \,||\, b \,||\, \phi \Leftrightarrow \phi$

if a ≠ σ ∧ b ≠ σ -- by (p.1) and (p.5)

$\forall a,b \in Activity \bullet (a \,||\, b) \,||\, \sigma \Leftrightarrow a \,||\, (\sigma \,||\, b) \Leftrightarrow a \,||\, b \,||\, \sigma \Leftrightarrow \sigma$ -- by (p.1) and (p.6)

$\forall a,b \in Activity \bullet (a + b) \,||\, \varepsilon \Leftrightarrow (a \,||\, \varepsilon) + (b \,||\, \varepsilon) \Leftrightarrow a + b$ -- by (p.3) and (p.4)

$\forall a,b \in Activity \bullet (a + b) \,||\, \phi \Leftrightarrow (a \,||\, \phi) + (b \,||\, \phi) \Leftrightarrow \phi$

if a ≠ σ ∧ b ≠ σ -- by (p.3) and (p.5)

5.3.7 Repetition

Repetition of tasks is defined as an until- and while-loop. The structures in the abstract syntax are constructed using recursion. The until-loop is formed by an *Activity* followed by an option of continuing or repeating *x*:

$$\mu x.(a; \varepsilon + x)$$

Intuitively it can be seen that the *Activity* is repeated as long as ε is not chosen. When ε is chosen (i.e. the end state of the recursion function is reached) the recursion terminates, which means that the next activity outside of the until-loop may be executed. The choice of the fixed-point *x* results in expanding unrolling the expression.

The until-loop has only one axiom specifying the unrolling of the recursions on the loop:

(r.1) $\forall a \in Activity \bullet \mu x.(a; \varepsilon + x) \Leftrightarrow a; \varepsilon + \mu x.(a; \varepsilon + x)$

-- unrolling one cycle of until-loop repetition

This rule can be applied as many times as necessary resulting possibly in an infinite repetition of the activity and the option to continue or repeat:

$\mu x.(a; \varepsilon + x) \Rightarrow a; \varepsilon + \mu x.(a; \varepsilon + x) \Rightarrow a; \varepsilon + (a; \varepsilon + \mu x.(a; \varepsilon + x)) \Rightarrow ...$ -- by (r.1)

Additionally, there are three special cases where the expression may be reduced, those ones when any of the instantaneous events is involved. In one case an until-loop containing just the empty element ε can be reduced just to ε:

$$\mu x.(\varepsilon; \varepsilon + x) \Rightarrow \varepsilon; \varepsilon + \mu x.(\varepsilon; \varepsilon + x) \Rightarrow \varepsilon; \varepsilon + (\varepsilon; \varepsilon + \mu x.(\varepsilon; \varepsilon + x)) \Rightarrow ... \Rightarrow \varepsilon$$

-- by (r.1) and (s.3)

The reduction of empty sequences can be made by the axiom (s.3). The recursion keeps going infinitely or finishes when the ε in the selection is chosen.

On the other hand, if the activity in the until-loop contains just ϕ or σ, the expression may be reduced and the recursion is eliminated:

$\mu x.(\phi; \varepsilon + x) \Rightarrow \phi; \varepsilon + \mu x.(\phi; \varepsilon + x) \Rightarrow ... \Rightarrow \phi$ -- by (r.1) and (s.4)

$\mu x.(\sigma; \varepsilon + x) \Rightarrow \sigma; \varepsilon + \mu x.(\sigma; \varepsilon + x) \Rightarrow ... \Rightarrow \sigma$ -- by (r.1) and (s.5)

As the examples above show, it is possible to reduce the until-loop using the axioms (s.4) or (s.5) already defined.

Alternatively, the while-loop is formed by the option of doing an *Activity* followed by repeating *x*, or the option of finishing the execution of the loop:

$$\mu x.(\varepsilon + a; x)$$

As the until-loop, the while-loop has only one axiom specifying the unrolling of the recursions on the loop:

(r.2) $\forall a \in Activity \bullet \mu x.(\varepsilon + a; x) \Leftrightarrow \varepsilon + a; \mu x.(\varepsilon + a; x)$ -- unrolling one cycle of while-loop repetition

Applying this rule as many times as necessary results in an infinite repetition of the option to finish the loop or doing the activity and repeat:

$\mu x.(\varepsilon + a; x) \Rightarrow \varepsilon + a; \mu x.(\varepsilon + a; x) \Rightarrow \varepsilon + a; (\varepsilon + a; \mu x.(\varepsilon + a; x)) \Rightarrow ...$ -- by (r.2)

Additionally, there are three special cases where the expression may be reduced, those ones when any of the instantaneous events is involved. The while-loop containing just the empty element ε can be reduced just to ε:

$\mu x.(\varepsilon + \varepsilon; x) \Rightarrow \varepsilon + \varepsilon; \mu x.(\varepsilon + \varepsilon; x) \Rightarrow \varepsilon + \varepsilon; (\varepsilon + \varepsilon; \mu x.(\varepsilon + \varepsilon; x)) \Rightarrow ... \Rightarrow \varepsilon$

-- by (r.2) and (s.3)

The reduction of empty sequences can be made by the axiom (s.3). The recursion keeps going infinitely or finishes when the ε in the selection is chosen. Finally, in the cases where the activity in the while-loop contains only the symbol ϕ or σ, the expression may be reduced and the recursion is eliminated:

$\mu x.(\varepsilon + \phi; x) \Rightarrow \varepsilon + \phi; \mu x.(\varepsilon + \phi; x) \Rightarrow ... \Rightarrow \varepsilon + \phi$ -- by (r.2) and (s.4)

$\mu x.(\varepsilon + \sigma; x) \Rightarrow \varepsilon + \sigma; \mu x.(\varepsilon + \sigma; x) \Rightarrow ... \Rightarrow \varepsilon + \sigma$ -- by (r.2) and (s.5)

5.3.8 Encapsulation

The encapsulation of tasks is used to isolate an *Activity* from the rest of the expression giving it a scope and a name. It is built by using curly brackets "{ }" around the *Activity*. Consequently, *{act}* represents the encapsulation of the *Activity act*. But, the real importance of encapsulation is denoting the scope of a compound task to limit the effect of σ and ϕ, which represent early exit.. A more detailed example could be:

$\{\{a1; \{a2 + a3\}; a4\}; a5\}$

Supposing *a1, a2, ..., a5* are simple tasks, in that case the expression also could be expressed as a set of compound tasks:

let $X = \{a2 + a3\}$

let $Y = \{a1; X; a4\}$

$\{Y; a5\}$

Using encapsulation is a way of abstracting the representation of a complex task flow and treating it as a single task (i.e. a subtask, part of another larger task), in the same

way that a complex diagram can be divided into different sub-diagrams to facilitate comprehension.

As mentioned above, when a *succeed* event occurs in an expression, this corresponds to an early exit from the scope of the enclosing task. The normal flow of control resumes at the task boundary. A different result is obtained when a *fail* event occurs in the expression. In this case, the *fail* event is promoted to the higher level, beyond the immediate task boundary. All the usual axioms apply to activity that is encapsulated within a task. Some additional axioms describe the specific effects of σ at the task boundary:

(e.1) $\{\sigma\} \Leftrightarrow \varepsilon \Leftrightarrow \{\varepsilon\}$ -- vacuous subtask

(e.2) $\forall a \in Activity \bullet \{a;\sigma\} \Leftrightarrow \{a\}$ -- coincident exit

(e.3) $\forall a \in Activity \bullet \{a + \sigma\} \Leftrightarrow \{a\} + \varepsilon$ -- vacuous selection

(e.4) $\{\phi\} \Leftrightarrow \phi$ -- promotion of fail

(e.5) $\forall a \in Activity \bullet \{a;\phi\} \Leftrightarrow \{a\};\phi$ -- promotion of fail in sequence

(e.6) $\forall a \in Activity \bullet \{a + \phi\} \Leftrightarrow \{a\} + \phi$ -- promotion of fail in selection

The vacuous subtask axiom (e.1) denotes that *succeed* alone within curly brackets is equivalent to the empty activity because *succeed* has no influence outside of its scope. Similarly, if *succeed* is next to the right bracket, it has no effect and may be removed even forming part of a sequence (e.2). The axiom (e.3) promotes the selection outside of the encapsulation area changing *succeed* for ε. Basically it establishes that a selection between an activity and *succeed* is equivalent to the choice of that activity within brackets and nothing (ε).If *fail* is alone within the curly brackets, it is promoted to the higher level by the axiom (e.4). The axiom (e.5) denotes the promotion of *fail* when this is next to the left bracket in a sequence. Finally, the axiom (e.6) promotes the selection and *fail* outside the curly brackets.

Additional axioms are not required for parallel composition and repetition, since the transformations can be derived from the existing ones:

$\forall a \in Activity \bullet \{a \parallel \sigma\} \Leftrightarrow \varepsilon$ -- by (p.5) and (e.1)

$\{\mu x.(\sigma;\varepsilon + x)\} \Leftrightarrow \varepsilon$ -- by (r.1), (s.5) and (e.1)

5.4 Summary

This chapter presented the abstract syntax representation for the task flow model in the Discovery method. The abstract task algebra is based on simple and compound tasks structured using operators such as sequence, selection, and parallel composition. Recursion and encapsulation are also considered. The axioms of the algebra were presented as well as a set of examples showing a combination of basic elements in the expressions denoting simple, and more complex, Task Flow diagrams. The definition of the denotational semantics for this algebra, giving the semantics in terms of traces, is depicted in the next chapter.

Chapter 6:
The Semantics of Tasks

The previous chapter described task diagrams formally in terms of an abstract task algebra, modelling all the syntactic constructions that may occur in task diagrams. In this chapter, constructions in the algebra are given a simple denotational semantics in terms of traces. The meaning of a system of tasks is given as a set of traces, in which a single trace is a symbolic string denoting one possible execution path through the system and the set of traces denotes all possible execution paths. The trace semantics may be used to prove the soundness of the axioms in the task algebra. They may also be used to show when syntactically different systems of tasks have the same underlying behaviour. Congruence properties are also demonstrated for the algebra.

6.1 Introduction to Trace Semantics

Trace theories were originally developed as semantic models of executing processes. A trace is a symbol string, where each symbol represents an atomic action executed by a process [134]. Simple trace theories for systems describe all the execution paths through a system [135]. Trace theories are more often used to describe the behaviour of concurrent systems in terms of all possible interleavings of the atomic actions of the concurrent processes [136]. The symbol strings may be drawn from an extended alphabet of symbols that also includes distinguished identifiers representing halting or refusals [134]. The work of Mazurkiewicz is generally acknowledged as the basis for the more sophisticated trace theories [137, 138]. Trace theories often include a description of the underlying labelled transition system (LTS), a finite state automaton whose transition labels are the same symbols collected in the traces. The traces of a system are those symbol strings emitted as all possible transition paths through the automaton are explored. The traces of a system are usually understood to include all partial paths through the automaton. A subset of these, known as the *complete traces* of a system, contains those paths describing the complete execution of the system.

Trace theories form the basis for many process calculi. Hoare's theory of Communicating Sequential Processes (CSP) uses a simple model, whose semantics is based on traces and interleaving [78, 79]. Processes are sequentially executing blocks, which may be composed serially or in parallel. Parallel processes are modelled as a choice between all possible serial interleavings of their respective atomic actions in interleaving semantics. However, in this work we are presenting a non-interleaving semantics because we want just a subset of the traces that all possible interleavings would generate. This subset is the result of the behaviour

required for three particular elements in the traces: commit, succeed and fail symbols. As it will be seen below, these symbols do not behave as normal elements in the traces. The equivalence between two systems is judged in terms of the equivalence of the two sets of traces derived from each system [134]. Milner's theory of Communicating Concurrent Systems (CCS) [73], which also forms the basis for the more sophisticated π calculus [75], assumes a much higher degree of concurrency. Processes are concurrent at a fine-grained level and execute atomic actions on the basis of whether one process can synchronise with another for the exchange of information. Equivalence is judged not just in terms of the observable traces, but also in terms of the structure of the underlying labelled transition systems, which generated the traces. The stronger equivalences, known as *simulation, bisimulation* and *congruence*, are needed because of the way in which processes synchronise in the presence of nondeterministic choice.

When judging equivalence on the basis of traces, it is common to restrict judgements to systems with finite traces. Where systems have looping or recursive behaviour, the traces are potentially infinite. It is sometimes possible to judge the equivalence of infinite trace systems using fixpoints, if the same fixed-point constructions can be derived in both systems. However, there are difficulties combining both the unrolling of recursion and the interleaving semantics of concurrency, since this yields an infinite and non-repeating pattern in the traces. In CSP, there are restrictions on the combination of these operators [78]. In CCS [73] and π [75], recursion is converted into the infinite (concurrent) replication of a process. The semantics are given using the same mechanism that interprets concurrency and synchronisation, which are primitive in Milner's theories.

6.2 Trace Semantics for Tasks

When considering what kind of theory to use for the semantics of tasks in the Discovery Method, it seems appropriate initially to use a trace model similar to that used in CSP, since the task algebra developed in chapter 5 is similar in character to CSP. The main differences are in the meaning of atomicity and the special treatment given to the early exit from a task. The chosen semantic model must be able to satisfy three main concerns. The first is to be able to prove the soundness of the axioms of the abstract task algebra. To achieve this, constructions in the task algebra that were deemed equivalent by assertion must be shown to produce identical traces in the semantic model. The second concern is to be able to prove when two systems of tasks exhibit equivalent behaviour. Here, a *system of tasks* is understood to mean a system described in a hierarchy of task abstractions, where the chosen levels of abstraction are essentially arbitrary. The third concern is to be able to prove strong compositional properties for the task algebra, such as congruence.

A denotational semantics in terms of sets of traces is presented in three parts. Firstly, the semantic domain of traces is described in section 6.3, including the alphabet of atomic symbols and trace constructions. Secondly, a set of semantic functions is presented in section 6.4. These functions are used to manipulate traces and sets of traces in the semantic domain. The kinds of function include trace concatenation, trace interleaving, the concatenated product of trace sets and the distributed interleaving of trace sets. A special function is also given to unpack the traces of an encapsulated task. These functions are used to give the meaning of the operators in

the syntactic domain, which were described in chapter 5. Finally, the main *trace* function is presented as a set of mapping functions in section 6.5, one for each type of construction in the syntactic domain. These functions translate an algebraic structure in the syntactic domain, representing a system of tasks, into a set of traces in the semantic domain, representing all possible complete executions of these tasks.

Small examples of each function are given, to illustrate their intended usage. A fuller treatment of the soundness of the axioms of the algebra from chapter 4 is presented in the following chapter 7. Likewise, proofs of the equivalence of systems of tasks are given. Finally, congruence properties are also demonstrated in chapter 7 and appendix B. The current chapter presents the semantics of tasks. The semantics is capable of describing all possible executions of a system of tasks and whether the system succeeds or fails as a whole.

6.3 The Trace Domain

The chosen trace domain is the infinite set of all traces, constructed according to the rules described below. Traces are strings of all lengths from zero to infinity, consisting of symbols drawn from an alphabet. The trace domain consists of all constructed traces of all possible lengths. This domain is partially ordered under all prefixes (representing all shorter execution paths) and also under all suffixes (representing all path completions). Either of these properties ensures that the trace domain is an ideal [139].

6.3.1 The Trace Alphabet

The trace alphabet includes the (potentially infinite) set of identifiers, representing the names of all the simple tasks to be analysed. (Recall that a *simple task* is the smallest unit of syntactic analysis, and represents a unit of business activity on the same scale as a *use case* in UML. The individual actions, which would correspond to the internal steps of a use case, are not analysable in this model). To this alphabet are added three distinguished symbols, representing special semantic elements.

$$\text{Symbol} ::= \text{Identifier} \cup \{\downarrow, \sigma, \phi\}$$

The special symbols have the following meaning:

- The distinguished symbol $\downarrow$ is the *commit* symbol, meaning commitment to a choice. This symbol is inserted into a trace at a selection point, to indicate that a particular path was chosen and other paths were discarded.

- The distinguished symbol σ is the *succeed* symbol, meaning early return with success. This symbol is inserted into traces where an activity pre-empts all others with immediate success.

- The distinguished symbol ϕ is the *fail* symbol, meaning early return with failure. This symbol is inserted into traces where an activity pre-empts all others with immediate failure.

The *commit* symbol is needed to distinguish pairs of traces in which the commitment to a choice is made at a different point in each trace, notwithstanding all the other

elements of the two traces being pairwise equal. This is useful to restrict the application of distributive laws to state contexts in which the choice condition can be evaluated; and prevents the migration of the choice point outside this context. The *succeed* and *fail* symbols both have the effect of short-circuiting the analysis of the activity in which they appear. Whereas *succeed* returns from the current activity to the next higher level, *fail* returns from the current activity to the top level and halts the execution of the system of tasks. These symbols interact in the rules for concatenating traces and unpacking traces. Both symbols σ and ϕ are used also in the abstract task algebra for the syntax domain and, as with the identifier's names, are overloaded for the semantic domain.

6.3.2 Construction of Traces

Traces in the trace domain are constructed from the empty trace and a *cons* operator, written as an infix dot, which adds a symbol to the head of a trace:

Trace ::= <> | Symbol . Trace

It is assumed that *cons* is primitive and so cannot be defined in terms of other operations. Traces may be represented in a deconstructed way using infix *cons*, or displayed more prettily as a sequence of symbols in angle brackets:

$\forall$a, b : Symbol •

<>	the empty trace
a.<> = <a>	the singleton trace *a*
a.b.<> = a.<b> = <a, b>	the length 2 trace *ab*

In the following treatment, both kinds of notation are used interchangeably. Given the above syntax, some example traces and their intuitive meanings are given below:

$\forall$a, b : Identifier •

<>	the empty trace	(1)
<σ>	exit to higher level	(2)
<ϕ>	halt and terminate	(3)
<a, b>	do a, b in order	(4)
<a, b, σ>	do a, b and return	(5)
<a, b, ϕ>	do a, b and halt	(6)
<a, $\downarrow$, b>	do a, then commit to b	(7)
<$\downarrow$, a, b>	commit to a, b	(8)

Note how the meanings of cases (2) and (3) above are pairwise distinct, likewise cases (5) and (6). The effect of σ is to exit from the current level, whereas ϕ exits from the whole system. Cases (4) and (5) locally behave in an identical way, but are in fact globally distinct. If these traces were extended by concatenation with another trace <*a*>, their difference would be observable. Note how the meanings of cases (7) and (8) are distinguished by the different position of the commit $\downarrow$. In (7) the state context for the choice is not available until after *a*, but in (8) this context is available before *a.*

It is important to say that the universe of possible traces is limited by the mapping functions, translating the syntactic expression into traces by using the appropriate

semantic functions. However, in order to prove completeness we need to specify explicit constraints on particular cases of traces that are not allowed:

- Contiguous commits in a trace. The commit symbol ↓ is merged for all contiguous occurrences, therefore traces containing more that one ↓ such as (<↓, ↓>, <↓, ↓, ↓>, <↓, ↓, ↓, ↓>…) are not valid.
- The succeed symbol not occurring at the end of a trace. The succeed symbol represents end with success of a trace and for this reason traces of the form of (*Trace.σ.Trace*) are not valid.
- The fail symbol not occurring at the end of a trace. Just like the succeed symbol, the fail symbol represents end with fail of a trace and therefore traces of the form of (*Trace.ϕ.Trace*) are also not valid.
- One trace with commit for option. As mentioned above, the commit symbol represents the commitment of a choice, therefore there is going to be a trace of the form of (*Trace.↓.Trace*) for each option represented in the task algebra expression.

6.4 Semantic Functions over the Trace Domain

In this section the semantic functions are defined manipulating the traces for the semantic domain. As mentioned before, the semantic functions are not used arbitrarily and they are employed by the mapping functions. There are semantic functions for concatenation of traces, concatenated product of trace sets, interleaving of traces, and distributed interleaving of trace sets. Additionally, semantic functions for unpacking trace sets are presented.

6.4.1 Concatenation of Traces

An infix function # is defined to concatenate two traces. The # function is the basis for many other semantic functions described in later subsections. In general, this appends all of argument 2 onto the end of argument 1. Special treatment is required to handle occurrences of ↓, σ and ϕ at the head of a trace. It is assumed that traces are in canonical form, such that σ and ϕ are always found as the last elements in a trace.

_ # _ : Trace → Trace → Trace

<> # trace = trace		(tc1)
<σ> # trace = <σ>		(tc2)
<ϕ> # trace = <ϕ>		(tc3)
<↓> # ↓.rest = ↓.rest		(tc4)
<↓> # a.rest = ↓.a.rest,	a ≠ ↓	(tc5)
a.rest # trace = a.(rest # trace),	a ≠ σ, a ≠ ϕ	(tc6)

The cases tc1 and tc6 define the empty trace <> as a left and right identity under concatenation. This is an expected property of concatenation. Cases tc2 and tc3 define σ and ϕ as a left zero, eliminating any trace on the right. This causes the semantic translation of any sequence to short-circuit as desired, resulting in early exit. Cases tc4 and tc5 merge all contiguous occurrences of ↓, such that all contiguous commits are treated as a single commit. This property is needed when resolving distribution over parallel composition.

Examples using the trace concatenation function are shown as follows:

Example 1:

$\forall x, y, z : Identifier \bullet < x, y > \# < z >$

$\Rightarrow x.(< y > \# < z >)$ -- by tc6

$\Rightarrow x.y.(<> \# < z >)$ -- by tc6

$\Rightarrow x.y. < z >$ -- by tc1

$\Rightarrow < x, y, z >$ -- cons operator

Example 2:

$\forall x : Identifier \bullet <> \# < x >$

$\Rightarrow < x >$ -- by tc1

Example 3:

$\forall x : Identifier \bullet < x > \# <>$

$\Rightarrow x.(<> \# <>)$ -- by tc6

$\Rightarrow x. <>$ -- by tc1

$\Rightarrow < x >$ -- cons operator

Example 4:

$<> \# <>$

$\Rightarrow <>$ -- by tc1

Example 5:

$\forall x, y : Identifier \bullet < \phi > \# < x, y >$

$\Rightarrow < \phi >$ -- by tc3

Example 6:

$\forall x : Identifier \bullet < x > \# < \phi >$

$\Rightarrow x.(<> \# < \phi >)$ -- by tc6

$\Rightarrow x. < \phi >$ -- by tc1

$\Rightarrow < x, \phi >$ -- cons operator

Example 7:

$<\phi>\#<>$

$\Rightarrow<\phi>$ -- by tc3

Example 8:

$<>\#<\phi>$

$\Rightarrow<\phi>$ -- by tc1

Example 9:

$<\phi>\#<\phi>$

$\Rightarrow<\phi>$ -- by tc3

Example 10:

$\forall x : Identifier \bullet <\downarrow>\#<x>$

$\Rightarrow\downarrow . <x>$ -- by tc5

$\Rightarrow<\downarrow, x>$ -- cons operator

Example 11:

$\forall x : Identifier \bullet <x>\#<\downarrow>$

$\Rightarrow x.(<>\#<\downarrow>)$ -- by tc6

$\Rightarrow x.<\downarrow>$ -- by tc1

$\Rightarrow<x,\downarrow>$ -- cons operator

Example 12:

$<\downarrow>\#<>$

$\Rightarrow\downarrow . <>$ -- by tc5

$\Rightarrow<\downarrow>$ -- cons operator

Example 13:

$<>\#<\downarrow>$

$\Rightarrow <\downarrow>$ -- by tc1

Example 14:

$<\downarrow>\# <\phi>$

$\Rightarrow \downarrow . <\phi>$ -- by tc5

$\Rightarrow <\downarrow, \phi>$ -- cons operator

Example 15:

$<\phi>\# <\downarrow>$

$\Rightarrow <\phi>$ -- by tc3

Example 16:

$\forall x : Identifier \bullet <\downarrow>\# <\downarrow, x>$

$\Rightarrow \downarrow . <x>$ -- by tc4

$\Rightarrow <\downarrow, x>$ -- cons operator

Example 17:

$\forall x, y : Identifier \bullet <\sigma>\# <x, y>$

$\Rightarrow <\sigma>$ -- by tc2

Example 18:

$\forall x : Identifier \bullet <x>\# <\sigma>$

$\Rightarrow x.(<>\# <\sigma>)$ -- by tc6

$\Rightarrow x. <\sigma>$ -- by tc1

$\Rightarrow <x, \sigma>$ -- cons operator

Example 19:

$<\sigma>\# <>$

$\Rightarrow <\sigma>$ -- by tc2

Example 20:

$<>\# <\sigma>$

$\Rightarrow <\sigma>$ -- by tc1

Example 21:

$<\sigma>\#<\sigma>$

$\Rightarrow <\sigma>$ -- by tc2

Example 22:

$<\downarrow>\#<\sigma>$

$\Rightarrow \downarrow . <\sigma>$ -- by tc5

$\Rightarrow <\downarrow, \sigma>$ -- cons operator

Example 23:

$<\sigma>\#<\downarrow>$

$\Rightarrow <\sigma>$ -- by tc2

Example 24:

$<\sigma>\#<\phi>$

$\Rightarrow <\sigma>$ -- by tc2

Example 25:

$<\phi>\#<\sigma>$

$\Rightarrow <\phi>$ -- by tc3

6.4.2 Concatenated Product of Trace Sets

In general, the semantics deal in sets of traces, representing multiple execution paths, rather than single traces. A version of concatenation is provided for sets of traces. This is the concatenated product, written as the infix function ⊗, which appends every trace in argument 2 onto the end of every trace in argument 1, using the simple concatenation function # defined in section 5.4.1 above to concatenate each distinct pair of traces.

$\_ \otimes \_ : \{Trace\} \rightarrow \{Trace\} \rightarrow \{Trace\}$

$seta \otimes setb = \{a \# b \mid a, b \in Trace, a \in seta, b \in setb\}$ (cp1)

The definition cp1 is given by comprehension on all pairs of traces *a, b* from the two argument sets. If either *seta* or *setb* is ∅, the comprehension yields ∅, showing that ∅ is a left and right zero. The singleton set containing the empty trace {<>} is a left and right identity, resulting in no change to the other argument.

Examples showing the trace concatenation function for sets:

Example 1:

$\forall x, y, z : Identifier \bullet \{< x >\} \otimes \{< y >, < z >\}$

$\Rightarrow \{< x > \# < y >, < x > \# < z >\}$ -- by cp1

$\Rightarrow \{< x, y >, < x, z >\}$ -- by tc

Example 2:

$\forall x, y : Identifier \bullet \{<>\} \otimes \{< x >, < y >\}$

$\Rightarrow \{<> \# < x >, <> \# < y >\}$ -- by cp1

$\Rightarrow \{< x >, < y >\}$ -- by tc

Example 3:

$\forall x, y : Identifier \bullet \{< x >, < y >\} \otimes \{<>\}$

$\Rightarrow \{< x > \# <>, < y > \# <>\}$ -- by cp1

$\Rightarrow \{< x >, < y >\}$ -- by tc

Example 4:

$\{<>, <>\} \otimes \{<>\}$

$\Rightarrow \{<> \# <>, <> \# <>\}$ -- by cp1

$\Rightarrow \{<>\}$ -- by tc

Example 5:

$\forall x, y : Identifier \bullet \{< \phi >\} \otimes \{< x >, < y >\}$

$\Rightarrow \{< \phi > \# < x >, < \phi > \# < y >\}$ -- by cp1

$\Rightarrow \{< \phi >\}$ -- by tc

Example 6:

$\forall x, y : Identifier \bullet \{< x >, < y >\} \otimes \{< \phi >\}$

$\Rightarrow \{< x > \# < \phi >, < y > \# < \phi >\}$ -- by cp1

$\Rightarrow \{< x, \phi >, < y, \phi >\}$ -- by tc

Example 7:

$$\forall x, y : Identifier \bullet \{<\downarrow>\} \otimes \{< x >, < y >\}$$

$$\Rightarrow \{<\downarrow># < x >, <\downarrow># < y >\} \quad \text{-- by cp1}$$

$$\Rightarrow \{<\downarrow, x >, <\downarrow, y >\} \quad \text{-- by tc}$$

Example 8:

$$\forall x, y : Identifier \bullet \{< x >, < y >\} \otimes \{<\downarrow>\}$$

$$\Rightarrow \{< x >#<\downarrow>, < y >#<\downarrow>\} \quad \text{-- by cp1}$$

$$\Rightarrow \{< x, \downarrow>, < y, \downarrow>\} \quad \text{-- by tc}$$

Example 9:

$$\forall x, y : Identifier \bullet \{< \sigma >\} \otimes \{< x >, < y >\}$$

$$\Rightarrow \{< \sigma >#< x >, < \sigma >#< y >\} \quad \text{-- by cp1}$$

$$\Rightarrow \{< \sigma >\} \quad \text{-- by tc}$$

Example 10:

$$\forall x, y : Identifier \bullet \{< x >, < y >\} \otimes \{< \sigma >\}$$

$$\Rightarrow \{< x >#< \sigma >, < y >#< \sigma >\} \quad \text{-- by cp1}$$

$$\Rightarrow \{< x, \sigma >, < y, \sigma >\} \quad \text{-- by tc}$$

6.4.3 Interleaving of Traces

The semantics of concurrency is given by interleaving the traces of the composed tasks. However, this is not understood in the usual way as the interleaving of atomic actions (as in CSP [78]). Even the smallest analysable tasks are not atomic in this sense, but execute over an interval of time. Concurrent tasks literally overlap in the semantics. They are initiated at a single instant, but may terminate at different instants. This justifies the interleaving treatment, on the grounds that all tasks assert state properties on completion, which may enable other tasks. So, the moment of task termination is what governs further control flow decisions, and it is these moments that are interleaved.

An infix function ~ is defined to compute all possible interleavings of two traces, returning the set of all interleaved combinations. This function is biased to resolve any competition between events and intervals in favour of the events. So, empty traces and any artefactual activity, such as *commit*, *succeed* and *fail*, are instantaneous events, which always pre-empt simple tasks, which are intervals. Competition among

simple tasks is resolved by computing all possible termination orders. Competition among events is resolved by a priority rule.

$$\_ \sim \_ : \text{Trace} \rightarrow \text{Trace} \rightarrow \{\text{Trace}\}$$

$$\langle\rangle \sim \text{trace} = \{\text{trace}\} \quad \text{(ti1)}$$
$$\text{trace} \sim \langle\rangle = \{\text{trace}\} \quad \text{(ti2)}$$
$$\langle\sigma\rangle \sim \text{trace} = \{\langle\sigma\rangle\} \quad \text{(ti3)}$$
$$\text{trace} \sim \langle\sigma\rangle = \{\langle\sigma\rangle\} \quad \text{(ti4)}$$
$$\langle\phi\rangle \sim \text{trace} = \{\langle\phi\rangle\}, \quad \text{trace} \neq \langle\sigma\rangle \quad \text{(ti5)}$$
$$\text{trace} \sim \langle\phi\rangle = \{\langle\phi\rangle\}, \quad \text{trace} \neq \langle\sigma\rangle \quad \text{(ti6)}$$
$$\downarrow.\text{rest} \sim \text{trace} = \{\langle\downarrow\rangle\} \otimes (\text{rest} \sim \text{trace}), \quad \text{trace} \neq \langle\sigma\rangle, \text{trace} \neq \langle\phi\rangle \quad \text{(ti7)}$$
$$\text{trace} \sim \downarrow.\text{rest} = \{\langle\downarrow\rangle\} \otimes (\text{rest} \sim \text{trace}), \quad \text{trace} \neq \langle\sigma\rangle, \text{trace} \neq \langle\phi\rangle \quad \text{(ti8)}$$
$$a.as \sim b.bs = (\{\langle a\rangle\} \otimes (as \sim b.bs)) \cup (\{\langle b\rangle\} \otimes (bs \sim a.as)),$$
$$a \neq \sigma, a \neq \phi, a \neq \downarrow, b \neq \sigma, b \neq \phi, b \neq \downarrow \quad \text{(ti9)}$$

Cases ti1 and ti2 describe the instant synchronisation of an empty trace on the left and right, yielding a singleton trace set containing the other trace. Cases ti3, ti4, ti5 and ti6 describe pre-emption on the left and right, exiting the composition with instant success or failure. Cases ti4, ti5 assert that success always takes priority over failure. These four cases are confluent with cases ti1 and ti2. The cases ti7 and ti8 deal with *commit* on the left and right, resolving in favour of the *commit* event, unless the other trace is a pre-emptive exit. These cases are confluent with cases ti1 and ti2. All posible basic cases where these rules are used confluently are shown as follows:

a) $\langle\rangle \sim \langle\rangle$
$\Rightarrow \{\langle\rangle\}$ -- by ti1

or:

$\langle\rangle \sim \langle\rangle$
$\Rightarrow \{\langle\rangle\}$ -- by ti2

b) $\langle\sigma\rangle \sim \langle\rangle$
$\Rightarrow \{\langle\sigma\rangle\}$ -- by ti2

or:

$\langle\sigma\rangle \sim \langle\rangle$
$\Rightarrow \{\langle\sigma\rangle\}$ -- by ti3

b) $\langle\rangle \sim \langle\sigma\rangle$
$\Rightarrow \{\langle\sigma\rangle\}$ -- by ti1

or:

$\langle\rangle \sim \langle\sigma\rangle$
$\Rightarrow \{\langle\sigma\rangle\}$ -- by ti4

c) $\langle\phi\rangle \sim \langle\rangle$
$\Rightarrow \{\langle\phi\rangle\}$ -- by ti2

or:

$\langle\phi\rangle \sim \langle\rangle$
$\Rightarrow \{\langle\phi\rangle\}$ -- by ti5

d) $\langle\rangle \sim \langle\phi\rangle$
$\Rightarrow \{\langle\phi\rangle\}$ -- by ti1

or:

$\langle\rangle \sim \langle\phi\rangle$
$\Rightarrow \{\langle\phi\rangle\}$ -- by ti6

e) $<>\sim<\downarrow>$

$\Rightarrow \{<\downarrow>\}$ -- by ti1

or:

$\Rightarrow \{<\downarrow>\} \otimes (<>\sim<>)$ -- ti8

$\Rightarrow \{<\downarrow>\} \otimes (<>)$ -- ti1 or ti2

$\Rightarrow \{<\downarrow># <>\}$ -- by cp1

$\Rightarrow \{<\downarrow>\}$ -- by tc5

f) $<\downarrow>\sim<>$

$\Rightarrow \{<\downarrow>\}$ -- by ti2

or:

$\Rightarrow \{<\downarrow>\} \otimes (<>\sim<>)$ -- ti7

$\Rightarrow \{<\downarrow>\} \otimes (<>)$ -- ti1 or ti2

$\Rightarrow \{<\downarrow># <>\}$ -- by cp1

$\Rightarrow \{<\downarrow>\}$ -- by tc5

The general case ti9 deals with traces whose first element is not one of the events handled in earlier cases, and computes all possible interleavings. The priority rule establishes a ranking among events: $\sigma > \phi > \downarrow$, which was chosen to avoid having to compute further interleavings.

Examples showing combination of basic elements are presented below:

Example 1:

$\forall x : Identifier \bullet <>\sim< x >$

$\Rightarrow \{< x >\}$ -- by ti1

Example 2:

$\forall x : Identifier \bullet < x >\sim<>$

$\Rightarrow \{< x >\}$ -- by ti2

Example 3:

$<>\sim<> \Rightarrow \{<>\}$ -- by ti1 or ti2

Example 4:

$\forall x : Identifier \bullet < \phi >\sim< x >$

$\Rightarrow \{< \phi >\}$ -- by ti5

Example 5:

$\forall x : Identifier \bullet < x >\sim< \phi >$

$\Rightarrow \{<\phi>\}$ -- by ti6

Example 6:

$<\phi>\sim<\phi>$

$\Rightarrow \{<\phi>\}$ -- by ti5 or ti6

Example 7:

$\forall x, y : Identifier \bullet <x>\sim<y>$

$\Rightarrow (\{<x>\} \otimes (<>\sim<y>)) \cup (\{<y>\} \otimes (<>\sim<x>))$ -- by ti9

$\Rightarrow (\{<x>\} \otimes \{<y>\}) \cup (\{<y>\} \otimes \{<x>\})$ -- by ti1

$\Rightarrow \{<x, y>\} \cup \{<y, x>\}$ -- by cp

$\Rightarrow \{<x, y>, <y, x>\}$ -- union

Example 8:

$\forall x, y, z : Identifier \bullet <x>\sim<y, z>$

$\Rightarrow (<\{x>\} \otimes (<>\sim<y, z>)) \cup (\{<y>\} \otimes (<z>\sim<x>))$ -- by ti9

$\Rightarrow (\{<x>\} \otimes (<>\sim<y, z>)) \cup (\{<y>\} \otimes ((\{<z>\}$
$\otimes (<>\sim<x>)) \cup (\{<x>\} \otimes (<>\sim<z>))))$ -- by ti9

$\Rightarrow (\{<x>\} \otimes \{<y, z>\}) \cup (\{<y>\} \otimes ((\{<z>\} \otimes \{<x>\}) \cup (\{<x>\} \otimes \{<z>\})))$
-- by ti1

$\Rightarrow \{x, y, z\} \cup (\{y\} \otimes (\{z, x\} \cup \{x, z\}))$ -- by cp

$\Rightarrow \{<x, y, z>\} \cup (\{<y>\} \otimes \{<z, x>, <x, z>\})$ -- union

$\Rightarrow \{<x, y, z>\} \cup \{<y, z, x>, <y, x, z>\}$ -- by cp

$\Rightarrow \{<x, y, z>, <y, z, x>, <y, x, z>\}$ -- union

Example 9:

$\forall x, y : Identifier \bullet <x>\sim<\downarrow, y>$

$\Rightarrow \{<\downarrow>\} \otimes (<y>\sim<x>)$ -- by ti8

$\Rightarrow \{<\downarrow>\} \otimes ((\{<y>\} \otimes (<>\sim<x>)) \cup (\{<x>\} \otimes (<>\sim<y>)))$

-- by ti9

$\Rightarrow \{<\downarrow>\} \otimes ((\{< y >\} \otimes \{< x >\}) \cup (\{< x >\} \otimes \{< y >\}))$ -- by ti1

$\Rightarrow \{<\downarrow>\} \otimes (\{< y, x >\} \cup \{< x, y >\})$ -- by cp

$\Rightarrow \{<\downarrow>\} \otimes \{< y, x >, < x, y >\}$ -- union

$\Rightarrow \{<\downarrow, y, x >, <\downarrow, x, y >\}$ -- by cp

Example 10:

$\forall x, y : Identifier \bullet <\downarrow, x >\sim< y >$

$\Rightarrow \{<\downarrow>\} \otimes (< x >\sim< y >)$ -- by ti7

$\Rightarrow \{<\downarrow>\} \otimes ((\{< x >\} \otimes (<>\sim< y >)) \cup (\{< y >\} \otimes (<>\sim< x >)))$

-- by ti9

$\Rightarrow \{<\downarrow>\} \otimes ((\{< x >\} \otimes \{< y >\}) \cup (\{< y >\} \otimes \{< x >\}))$ -- by ti1

$\Rightarrow \{<\downarrow>\} \otimes (\{< x, y >\} \cup < y, x >\})$ -- by cp

$\Rightarrow \{<\downarrow>\} \otimes \{< x, y >, < y, x >\}$ -- union

$\Rightarrow \{<\downarrow, x, y >, <\downarrow, y, x >\}$ -- by cp

Example 11:

$<>\sim<\downarrow>$

$\Rightarrow \{<\downarrow>\}$ -- by ti1

Or:

$\Rightarrow \{<\downarrow>\} \otimes (<>\sim<>)$ -- ti8

$\Rightarrow \{<\downarrow>\} \otimes (<>)$ -- ti1 or ti2

$\Rightarrow \{<\downarrow>\}$ -- by cp

Example 12:

$<\downarrow>\sim<>$

$\Rightarrow \{<\downarrow>\}$ -- by ti2

Or:

$\Rightarrow \{<\downarrow>\} \otimes (<>\sim<>)$ -- ti7

$\Rightarrow \{<\downarrow>\} \otimes (<>)$ -- ti1 or ti2

$\Rightarrow \{<\downarrow>\}$ -- by cp

Example 13:

$\forall x : Identifier \bullet <>\sim<\downarrow, x>$

$\Rightarrow \{<\downarrow, x>\}$ -- by ti1

Example 14:

$\forall x : Identifier \bullet <\downarrow, x>\sim<>$

$\Rightarrow \{<\downarrow, x>\}$ -- by ti2

Example 15:

$\forall x : Identifier \bullet <\phi>\sim<\downarrow, x>$

$\Rightarrow \{<\phi>\}$ -- by ti5

Example 16:

$\forall x : Identifier \bullet <\downarrow, x>\sim<\phi>$

$\Rightarrow \{<\phi>\}$ -- by ti6

Example 17:

$\forall x, y : Identifier \bullet <\downarrow, x>\sim<\downarrow, y>$

$\Rightarrow \{<\downarrow>\} \otimes (<x>\sim<\downarrow, y>)$ -- by ti7

$\Rightarrow \{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes (<y>\sim<x>))$ -- by ti8

$\Rightarrow \{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes ((\{<y>\} \otimes (<>\sim<x>)) \cup (\{<x>\} \otimes (<>\sim<y>))))$
-- by ti9

$\Rightarrow \{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes ((\{<y>\} \otimes \{<x>\}) \cup (\{<x>\} \otimes \{<y>\})))$

-- by ti1

$\Rightarrow \{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes (\{<y, x>\} \cup \{<x, y>\}))$ -- by cp

$\Rightarrow \{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes \{<y, x>, <x, y>\})$ -- union

$\Rightarrow \{<\downarrow>\} \otimes \{<\downarrow, y, x>, <\downarrow, x, y>\}$ -- by cp

$\Rightarrow \{<\downarrow, y, x>, <\downarrow, x, y>\}$ -- by cp

Example 18:

$\forall x : Identifier \bullet < \sigma > \sim < x >$

$\Rightarrow \{< \sigma >\}$ -- by ti3

Example 19:

$\forall x : Identifier \bullet < x > \sim < \sigma >$

$\Rightarrow \{< \sigma >\}$ -- by ti4

Example 20:

$< \sigma > \sim < \sigma >$

$\Rightarrow \{< \sigma >\}$ -- by ti3 or ti4

6.4.4 Distributed Interleaving of Trace Sets

In general, the semantics deal in sets of traces rather than single traces. A version of interleaving is provided for trace sets, describing this as the interleaving of all possible pairs of traces from each set. This is the infix function // for distributed interleaving. Similar in construction to the concatenated product in section 5.4.2, the difference is that this function's result must be flattened by taking the distributed union of the resulting trace sets.

_ // _ : {Trace} → {Trace} → {Trace}

seta // setb = ∪{ a ~ b | a, b ∈ Trace, a ∈ seta, b ∈ setb } (di1)

The case di1 defines the distributed interleaving of trace sets by set comprehension. For each distinct pair of traces *a, b* in the argument sets, it computes the set of traces resulting from the simple interleaving of those traces. The comprehension collects a set of sets. The result is flattened using ∪. If either *seta* or *setb* is ∅, the comprehension collects {∅} and distributed union yields ∅, showing that ∅ is a left and right zero. The singleton set containing the empty trace {<>} is a left and right identity, resulting in no change to the other argument.

Examples with the trace interleaving function for sets:

Example 1:

$\forall x, y, z : Identifier \bullet \{< x >\} // \{< y >, < z >\}$

$\Rightarrow \bigcup \{< x > \sim < y >, < x > \sim < z >\}$ -- by di1

$\Rightarrow \bigcup\{\{< x, y >, < y, x >\}, \{< x, z >, < z, x >\}\}$ -- by ti

$\Rightarrow \{< x, y >, < y, x >, < x, z >, < z, x >\}$

Example 2:

$\forall x, y : Identifier \bullet \{< x >, < y >\} // \{<>\}$

$\Rightarrow \bigcup\{< x >\sim<>, < y >\sim<>\}$ -- by di1

$\Rightarrow \bigcup\{\{< x >\}, \{< y >\}\}$ -- by ti

$\Rightarrow \{< x >, < y >\}$

Example 3:

$\forall x, y : Identifier \bullet \{<>\} // \{< x >, < y >\}$

$\Rightarrow \bigcup\{<>\sim< x >, <>\sim< y >\}$ -- by di1

$\Rightarrow \bigcup\{\{< x >\}, \{< y >\}\}$ -- by ti

$\Rightarrow \{< x >, < y >\}$

Example 4:

$\forall x, y : Identifier \bullet \{< x >, < y >\} // \{< \phi >\}$

$\Rightarrow \bigcup\{< x >\sim< \phi >, < y >\sim< \phi >\}$ -- by di1

$\Rightarrow \bigcup\{\{< \phi >\}, \{< \phi >\}\}$ -- by ti

$\Rightarrow \{< \phi >\}$

Example 5:

$\forall x, y : Identifier \bullet \{< \phi >\} // \{< x >, < y >\}$

$\Rightarrow \bigcup\{< \phi >\sim< x >, < \phi >\sim< y >\}$ -- by di1

$\Rightarrow \bigcup\{\{< \phi >\}, \{< \phi >\}\}$ -- by ti

$\Rightarrow \{< \phi >\}$

Example 6:

$\forall x, y, z : Identifier \bullet \{<\downarrow, x >\} // \{< y >, < z >\}$

$\Rightarrow \bigcup\{<\downarrow, x >\sim< y >, <\downarrow, x >\sim< z >\}$ -- by di1

$\Rightarrow \bigcup\{\{<\downarrow, x, y>, <\downarrow, y, z>\}, \{<\downarrow, x, z>, <\downarrow, z, x>\}\}$ -- by ti

$\Rightarrow \{<\downarrow, x, y>, <\downarrow, y, z>, <\downarrow, x, z>, <\downarrow, z, x>\}$

Example 7:

$\forall x, y : Identifier \bullet \{<x>, <y>\} // \{<\sigma>\}$

$\Rightarrow \bigcup\{<x> \sim <\sigma>, <y> \sim <\sigma>\}$ -- by di1

$\Rightarrow \bigcup\{\{<\sigma>\}, \{<\sigma>\}\}$ -- by ti

$\Rightarrow \{<\sigma>\}$

Example 8:

$\forall x, y : Identifier \bullet \{<\sigma>\} // \{<x>, <y>\}$

$\Rightarrow \bigcup\{<\sigma> \sim <x>, <\sigma> \sim <y>\}$ -- by di1

$\Rightarrow \bigcup\{\{<\sigma>\}, \{<\sigma>\}\}$ -- by ti

$\Rightarrow \{<\sigma>\}$

Example 9:

$\forall x, y : Identifier \bullet \{<\sigma>\} // \{<\phi>\}$

$\Rightarrow \bigcup\{<\sigma> \sim <\phi>, <\sigma> \sim <\phi>\}$ -- by di1

$\Rightarrow \bigcup\{\{<\sigma>\}, \{<\sigma>\}\}$ -- by ti

$\Rightarrow \{<\sigma>\}$

6.4.5 Unpacking of Trace Sets

In theories dealing with type abstraction, the rules governing information hiding and the dual notion of information revealing are sometimes called *packing* and *unpacking* rules, respectively [140]. The task semantics requires an unpacking rule to remove the abstraction boundary from around an encapsulated task, while preserving the intended boundary semantics of the pre-empting events σ and ϕ. The main goal of the unpacking function is to ensure that the effects of σ are only felt up to the task boundary, but the effect of ϕ should be propagated up to the top level. An auxiliary function *lift* is defined to lift a trace from inside a boundary to outside the boundary:

lift : Trace → Trace

lift <> = <> (li1)
lift <σ> = <> (li2)
lift a.as = a.(lift as), a ≠ σ (li3)

This has the effect of stripping σ from the ends of a trace, but allows all other traces to proceed unaffected. In particular, a trace of the form: <a, b, c, σ> will be reduced to: <a, b, c> at the higher level and so will be able to combine with other traces. However, a trace of the form: <a, b, c, ϕ> will propagate upwards unchanged, such that any attempt to combine this trace with others will eliminate the other traces.

The general unpacking function for a trace set, *unpack*, is defined as the set comprehension:

unpack : {Trace} → {Trace}

unpack seta = { lift a | a ∈ Trace, a ∈ seta } (up1)

This basically lifts every trace in the argument set. Later, *unpack* is used on the set of traces computed from a constructed task, before the abstraction boundary of this task is removed. Its normal and pre-empting traces are lifted to the higher level. Whereas normal traces and traces that pre-empt with success will generate normally terminating traces at the higher level, all traces that pre-empt with failure will be transmitted unchanged to the higher level, such that failure will occur at this level also. Eventually, the traces containing failures will rise to the top level, indicating all those execution paths that cause the system of tasks to fail.

Example 1:

$\forall x, y, z : Identifier \bullet unpack \{\langle x, y\rangle, \langle z\rangle\}$

$\Rightarrow \{\langle x, y\rangle, \langle z\rangle\}$ -- by up1

Example 2:

$\forall x : Identifier \bullet unpack \{\langle x, \phi\rangle, \langle \phi\rangle\}$

$\Rightarrow \{\langle x, \phi\rangle, \langle \phi\rangle\}$ -- by up1

Example 3:

$\forall x : Identifier \bullet unpack \{\langle x, \sigma\rangle, \langle \sigma\rangle\}$

$\Rightarrow \{\langle x\rangle, \langle\rangle\}$ -- by up1

In the last examples can be seen how identifiers and ϕ are passed directly to the higher level, while the example 3 shows that where unpacking a set of traces σ is eliminated by the function *lift* (li2).

6.5 Interpreting Task Algebra in the Trace Domain

The last part of the simple semantics of tasks is described as a translation function, or mapping function, that maps a syntactic expression in the algebra into a set of traces in the trace domain. The meaning of a syntactic expression is denoted by:

[[_]] : Activity → {Trace}

where ⟦ ⟧ indicates application of the *trace* function to the syntactic expression, to yield a set of traces, which is the denotation of the expression's meaning. This tracing function is defined piece-wise over every construction case in the syntactic domain. In the following, the tracing function for each case is given separately.

6.5.1 Tracing Basic Elements

The basic elements are the minimal elements that can be represented in the traces. Each of these functions generates a singleton:

$$⟦\varepsilon⟧ = \{<>\} \qquad \text{(tb1)}$$

$$\forall x : Simple \cup \{\sigma, \phi\} \bullet ⟦x⟧ = \{<x>\} \qquad \text{(tb2)}$$

Here, the elements σ and ϕ are presumed to exist in both the syntax and semantic domains. Similarly, the identifiers for simple tasks exist in both the syntax and semantic domains.

6.5.2 Tracing a Sequence of Activity

Trace of sequences defines the mapping function for a sequence by applying the concatenated product of trace sets. Tracing a sequence is solved by the concatenation of the partial traces of the sequence:

$$\forall a, b : Activity \bullet ⟦a\ ;\ b⟧ = ⟦a⟧ \otimes ⟦b⟧ \qquad \text{(ts1)}$$

Where both ⟦*a*⟧ and ⟦*b*⟧ are also trace functions mapping activities to sets of traces. These functions have to be resolved before calculating their product. Below, the possible combinations of atomic traces in traces for sequences are exemplified.

Sequence of Simple Task elements is made reducing each element to its equivalent identifier. Subsequently the trace concatenation semantic function is applied.

$\forall x, y \in Simple \bullet ⟦x;\ y⟧$

$\Rightarrow ⟦x⟧ \otimes ⟦y⟧$ -- by ts1

$\Rightarrow \{< x >\} \otimes \{< y >\}$ -- by tb2

$\Rightarrow \{< x > \# < y >\}$ -- by cp1

$\Rightarrow \{x.(<> \# < y >)\}$ -- by tc6

$\Rightarrow \{x. < y >\}$ -- by tc1

$\Rightarrow \{x,y\}$ -- cons operator

A sequence of a Simple Task with an empty trace results in just the Simple Task, with ε working as the identity element, it is either on the left or on the right of the Simple Task. Empty trace on the right:

$\forall x \in Simple \bullet [\![x;\ \varepsilon]\!]$

$\Rightarrow [\![x]\!] \otimes [\![\varepsilon]\!]$ -- by ts1

$\Rightarrow \{< x >\} \otimes [\![\varepsilon]\!]$ -- by tb2

$\Rightarrow \{< x >\} \otimes \{<>\}$ -- by tb1

$\Rightarrow \{< x > \# <>\}$ -- by cp1

$\Rightarrow \{x.(<> \# <>)\}$ -- by tc6

$\Rightarrow \{x.<>\}$ -- by tc1

$\Rightarrow \{< x >\}$ -- cons operator

Empty trace on the left:

$\forall x \in Simple \bullet [\![\varepsilon;\ x]\!]$

$\Rightarrow [\![\varepsilon]\!] \otimes [\![x]\!]$ -- by ts1

$\Rightarrow [\![\varepsilon]\!] \otimes \{< x >\}$ -- by tb2

$\Rightarrow \{<>\} \otimes \{< x >\}$ -- by tb1

$\Rightarrow \{<> \# < x >\}$ -- by cp1

$\Rightarrow \{< x >\}$ -- by tc1

A sequence of a Simple Task element with *succeed* results in the two elements concatenated if *succeed* is on the right side:

$\forall x \in Simple \bullet [\![x;\ \sigma]\!]$

$\Rightarrow [\![x]\!] \otimes [\![\sigma]\!]$ -- by ts1

$\Rightarrow \{< x >\} \otimes [\![\sigma]\!]$ -- by tb2

$\Rightarrow \{< x >\} \otimes \{< \sigma >\}$ -- by tb2

$\Rightarrow \{< x > \# < \sigma >\}$ -- by cp1

$\Rightarrow \{x.(<> \# < \sigma >)\}$ -- by tc6

$\Rightarrow \{x.< \sigma >\}$ -- by tc1

$\Rightarrow \{< x, \sigma >\}$ -- cons operator

But, if *succeed* is on the left side, the mapping function should return the singleton σ:

$\forall x \in Simple \bullet [\![\sigma; x]\!]$

$\Rightarrow [\![\sigma]\!] \otimes [\![x]\!]$ -- by ts1

$\Rightarrow [\![\sigma]\!] \otimes \{< x >\}$ -- by tb2

$\Rightarrow \{< \sigma >\} \otimes \{< x >\}$ -- by tb2

$\Rightarrow \{< \sigma > \# < x >\}$ -- by cp1

$\Rightarrow \{< \sigma >\}$ -- by tc2

In the sequence of the empty activity and *succeed*, ε also works as the identity element:

$[\![\varepsilon; \sigma]\!]$

$\Rightarrow [\![\varepsilon]\!] \otimes [\![\sigma]\!]$ -- by ts1

$\Rightarrow \{<>\} \otimes [\![\sigma]\!]$ -- by tb1

$\Rightarrow \{<>\} \otimes \{< \sigma >\}$ -- by tb2

$\Rightarrow \{<> \# < \sigma >\}$ -- by cp1

$\Rightarrow \{< \sigma >\}$ -- by tc1

And *succeed* on the left:

$[\![\sigma; \varepsilon]\!]$

$\Rightarrow [\![\sigma]\!] \otimes [\![\varepsilon]\!]$ -- by ts1

$\Rightarrow \{< \sigma >\} \otimes [\![\varepsilon]\!]$ -- by tb2

$\Rightarrow \{< \sigma >\} \otimes \{<>\}$ -- by tb1

$\Rightarrow \{< \sigma > \# <>\}$ -- by cp1

$\Rightarrow \{< \sigma >\}$ -- by tc2

A sequence of empty activities is equivalent to the empty trace:

$[\![\varepsilon; \varepsilon]\!]$

$\Rightarrow [\![\varepsilon]\!] \otimes [\![\varepsilon]\!]$ -- by ts1

$\Rightarrow \{<>\} \otimes \{<>\}$ -- by tb1

$\Rightarrow \{<>\#<>\}$ -- by cp1

$\Rightarrow \{<>\}$ -- by tc1

A sequence of *succeeds* results in the singleton σ after applying the trace concatenation semantic function:

$[\![\sigma; \sigma]\!]$

$\Rightarrow [\![\sigma]\!] \otimes [\![\sigma]\!]$ -- by ts1

$\Rightarrow \{<\sigma>\} \otimes \{<\sigma>\}$ -- by tb2

$\Rightarrow \{<\sigma>\#<\sigma>\}$ -- by cp1

$\Rightarrow \{<\sigma>\}$ -- by tc2

Fail has the same behaviour as *succeed* in a sequence. For instance, a sequence of a simple task and *fail*:

$\forall x \in Simple \bullet [\![x; \phi]\!]$

$\Rightarrow [\![x]\!] \otimes [\![\phi]\!]$ -- by ts1

$\Rightarrow \{<x>\} \otimes [\![\phi]\!]$ -- by tb2

$\Rightarrow \{<x>\} \otimes \{<\phi>\}$ -- by tb2

$\Rightarrow \{<x>\#<\phi>\}$ -- by cp1

$\Rightarrow \{x.(<>\#<\phi>)\}$ -- by tc6

$\Rightarrow \{x.<\phi>\}$ -- by tc1

$\Rightarrow \{<x, \phi>\}$ -- cons operator

Some significant examples are the ones when *succeed* and *fail* are together in the expression:

$[\![\sigma; \phi]\!]$

$\Rightarrow [\![\sigma]\!] \otimes [\![\phi]\!]$ -- by ts1

$\Rightarrow \{<\sigma>\} \otimes \{<\phi>\}$ -- by tb2

$\Rightarrow \{<\sigma>\#<\phi>\}$ -- by cp1

$\Rightarrow \{< \sigma >\}$ -- by tc2

And the other way around:

$[\![\phi; \sigma]\!]$

$\Rightarrow [\![\phi]\!] \otimes [\![\sigma]\!]$ -- by ts1

$\Rightarrow \{< \phi >\} \otimes \{< \sigma >\}$ -- by tb2

$\Rightarrow \{< \phi > \# < \sigma >\}$ -- by cp1

$\Rightarrow \{< \phi >\}$ -- by tc3

6.5.3 Tracing a Selection of Activity

Trace for selection defines the mapping function for the choice between two set of traces:

$\forall a, b : Activity \bullet [\![a + b]\!] =$ **if** $([\![a]\!] = [\![b]\!])$
then $[\![a]\!]$ (ta1)
else $\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])$ (ta2)

Tracing a selection is defined as the concatenated product of the singleton containing the commit symbol with the union of the traces of each operand. There exists a special case considering the idempotent axiom (sel.3) defined in the last chapter. The mapping function defines only the trace of one activity if *a* is equal to *b*. This allows the idempotent behaviour in the next expressions:

$\forall x \in Simple \bullet [\![x + x]\!]$

$\Rightarrow [\![x]\!]$ -- by ta1

$\Rightarrow \{< x >\}$ -- by tb2

The condition for idempotence is defined using a syntactic equivalence where a selection is considered idempotent if two activities, such as *a* and *b*, are the same; as it was defined in the chapter 4.

The idempotence of the empty sequence is shown in the next example:

$[\![\varepsilon + \varepsilon]\!]$

$\Rightarrow [\![\varepsilon]\!]$ -- by ta1

$\Rightarrow \{<>\}$ -- by tb1

With *succeed* the difference is that it is mapped to σ*:*

$[\![\sigma + \sigma]\!]$

$\Rightarrow [\![\sigma]\!]$ -- by ta1

$\Rightarrow \{< \sigma >\}$ -- by tb2

The choice between two different simple tasks is presented in the next example:

$\forall x, y \in Simple \bullet [\![x + y]\!]$

$\Rightarrow \{<\downarrow>\} \otimes [\![x]\!] \cup [\![y]\!]$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes \{< x >\} \cup \{< y >\}$ -- by iii

$\Rightarrow \{<\downarrow>\} \otimes \{< x >, < y >\}$ -- union of traces

$\Rightarrow \{<\downarrow> \# < x >, <\downarrow> \# < y >\}$ -- by cp1

$\Rightarrow \{<\downarrow, x >, <\downarrow, y >\}$ -- by tc5

The choice between a simple task and an empty activity:

$\forall x \in Simple \bullet [\![x + \varepsilon]\!]$

$\Rightarrow \{<\downarrow>\} \otimes [\![x]\!] \cup [\![\varepsilon]\!]$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes \{< x >\} \cup \{<>\}$ -- by tb1 and tb2

$\Rightarrow \{<\downarrow>\} \otimes \{< x >, <>\}$ -- union of traces

$\Rightarrow \{<\downarrow> \# < x >, <\downarrow> \# <>\}$ -- by cp1

$\Rightarrow \{<\downarrow, x >, <\downarrow>\}$ -- by tc5

In a similar way, the choice between a simple task and *succeed* results in a trace with x and a trace with σ, both preceded by a commit:

$\forall x \in Simple \bullet [\![x + \sigma]\!]$

$\Rightarrow \{<\downarrow>\} \otimes [\![x]\!] \cup [\![\sigma]\!]$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes \{< x >\} \cup \{< \sigma >\}$ -- by tb2

$\Rightarrow \{<\downarrow>\} \otimes \{< x >, < \sigma >\}$ -- union of traces

$\Rightarrow \{<\downarrow> \# < x >, <\downarrow> \# < \sigma >\}$ -- by cp1

$\Rightarrow \{<\downarrow, x >, <\downarrow, \sigma >\}$ -- by tc5

The selection between a simple task and *fail* is solved in the same way:

$\forall x \in Simple \bullet [\![x + \phi]\!]$

$\Rightarrow \{<\downarrow>\} \otimes [\![x]\!] \cup [\![\phi]\!]$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes \{< x >\} \cup \{< \phi >\}$ -- by tb2

$\Rightarrow \{<\downarrow>\} \otimes \{< x >, < \phi >\}$ -- union of traces

$\Rightarrow \{<\downarrow> \# < x >, <\downarrow> \# < \phi >\}$ -- by cp1

$\Rightarrow \{<\downarrow, x >, <\downarrow, \phi >\}$ -- by tc5

A selection between *succeed* and *fail* results, as can be expected, results in a set with two traces with σ and ϕ, both preceded by $\downarrow$:

$[\![\sigma + \phi]\!]$

$\Rightarrow \{<\downarrow>\} \otimes [\![\sigma]\!] \cup [\![\phi]\!]$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes \{< \sigma >\} \cup \{< \phi >\}$ -- by tb2

$\Rightarrow \{<\downarrow>\} \otimes \{< \sigma >, < \phi >\}$ -- union of traces

$\Rightarrow \{<\downarrow> \# < \sigma >, <\downarrow> \# < \phi >\}$ -- by cp1

$\Rightarrow \{<\downarrow, \sigma >, <\downarrow, \phi >\}$ -- by tc5

6.5.4 Tracing a Parallel Composition of Activity

Parallelism was defined before simply as the simultaneous execution of tasks. The precise meaning expressed here is that parallel composition is solved as the interleaving of all the possible terminations of the simple tasks within the expression:

$\forall a, b : Activity \bullet [\![a \,||\, b]\!] = [\![a]\!] \,//\, [\![b]\!]$ (tp1)

Specifically, traces for parallelism of $a||b$ is defined as the interleaving of the set of traces of a with the set of traces of b. The operator // which uses the union distribution is applied.

Below, an example showing the transformation of an expression with two simple tasks in parallel composition:

$\forall x, y \in Simple \bullet [\![x \,||\, y]\!]$

$\Rightarrow [\![x]\!] \,//\, [\![y]\!]$ -- by tp1

$\Rightarrow \{< x >\} \,//\, \{< y >\}$ -- by tb2

$\Rightarrow \cup\{< x >\sim< y >\}$ -- by di1

$\Rightarrow \cup\{\{< x,y >,< y,x >\}\}$ -- by ti9

$\Rightarrow \{< x, y >,< y, x >\}$

A simple task in parallel composition with an empty activity results only in the simple task:

$\forall x \in Simple \bullet [\![x \parallel \varepsilon]\!]$

$\Rightarrow [\![x]\!] // [\![\varepsilon]\!]$ -- by tp1

$\Rightarrow \{< x >\} // \{<>\}$ -- by tb2 and tb1

$\Rightarrow \cup\{< x >\sim<>\}$ -- by di1

$\Rightarrow \cup\{\{< x >\}\}$ -- by ti2

$\Rightarrow \{< x >\}$

A finished activity predominates over any expression in parallel composition. In this case, the example depicts a simple task with *succeed* as the other operand:

$\forall x \in Simple \bullet [\![x \parallel \sigma]\!]$

$\Rightarrow [\![x]\!] // [\![\sigma]\!]$ -- by tp1

$\Rightarrow \{< x >\} // \{< \sigma >\}$ -- by tb2

$\Rightarrow \cup\{< x >\sim< \sigma >\}$ -- by di1

$\Rightarrow \cup\{\{< \sigma >\}\}$ -- by ti4

$\Rightarrow \{< \sigma >\}$

The case of a pair of empty activities in parallel composition is shown below:

$[\![\varepsilon \parallel \varepsilon]\!]$

$\Rightarrow [\![\varepsilon]\!] // [\![\varepsilon]\!]$ -- by tp1

$\Rightarrow \{<>\} // \{<>\}$ -- by tb1

$\Rightarrow \cup\{<>\sim<>\}$ -- by di1

$\Rightarrow \cup\{\{<>\}\}$ -- by ti1

$\Rightarrow \{<>\}$

When *succeed* is in both sides of the operator, only one σ prevails:

$[\![\sigma || \sigma]\!]$

$\Rightarrow [\![\sigma]\!] // [\![\sigma]\!]$ -- by tp1

$\Rightarrow \{<\sigma>\} // \{<\sigma>\}$ -- by tb2

$\Rightarrow \cup\{<\sigma>\sim<\sigma>\}$ -- by di1

$\Rightarrow \cup\{\{<\sigma>\}\}$ -- by ti3

$\Rightarrow \{<\sigma>\}$

The next example shows the empty activity in parallel composition with *succeed*:

$[\![\varepsilon || \sigma]\!]$

$\Rightarrow [\![\varepsilon]\!] // [\![\sigma]\!]$ -- by tp1

$\Rightarrow \{<>\} // \{<\sigma>\}$ -- by tb1 and tb2

$\Rightarrow \cup\{<>\sim<\sigma>\}$ -- by di1

$\Rightarrow \cup\{\{<\sigma>\}\}$ -- by ti1

$\Rightarrow \{<\sigma>\}$

In the case of *succeed* in parallel composition with *fail*, *succeed* will prevail over *fail*:

$[\![\sigma || \phi]\!]$

$\Rightarrow [\![\sigma]\!] // [\![\phi]\!]$ -- by tp1

$\Rightarrow \{<\sigma>\} // \{<\phi>\}$ -- by tb2

$\Rightarrow \cup\{<\sigma>\sim<\phi>\}$ -- by di1

$\Rightarrow \cup\{\{<\sigma>\}\}$ -- by ti3

$\Rightarrow \{<\sigma>\}$

The final case presented in this section presents an expression where *fail* is in both sides of the parallel composition operator:

$[\![\phi || \phi]\!]$

$\Rightarrow [\![\phi]\!] // [\![\phi]\!]$ -- by tp1

$\Rightarrow \{<\phi>\} // \{<\phi>\}$ -- by tb2

$\Rightarrow \cup\{<\phi>\sim<\phi>\}$ -- by di1

$\Rightarrow \cup\{\{<\phi>\}\}$ -- by ti5

$\Rightarrow \{<\phi>\}$

6.5.5 Tracing a Repetition of Activity

The interpretation for repetition is more difficult to express. While the constructions already explained produce finite sets of traces, the repetition involve computing infinite sets of traces. Traces are usually records of finite executions [135]. Even when it is possible to express formally infinite traces, in practice it is hard to combine these in other rules.

Just as the syntactic model of repetition binds a repetition over some activity *x* in: $\mu x.f(x)$, it is also desirable to express the repetition with infinite traces, binding a fixpoint over a set of traces *t* in: $\mu t.g(t)$. The problem is that to determine the correct form of the expression *g(t)*, the mapping function ⟦ ⟧ should be applied recursively within the scope of the fixpoint; but the mapping function is defined to be applied to syntactic expressions. We need to make this supposition, which is related to the property of completeness of the semantics proposed for our algebra, because a way is needed of referring to the recursion variable x in a scope where *t* is bound. Completeness is a property not considered fully within the scope of this work because it is complex to prove and will be specified as possible future work arising from this research.

Supposing that the ⟦ ⟧ function has an abstract inverse, $⟦ ⟧^{-1}$, such that:

$$\forall x : \text{Activity} \bullet \forall y : \text{Trace} \bullet ⟦x⟧ = y \Leftrightarrow ⟦y⟧^{-1} = x \wedge ⟦ ⟦y⟧^{-1} ⟧ = y$$

After this, *x* can be denoted in terms of *t* by referring to *x* as $⟦t⟧^{-1}$, and later expect $⟦x⟧= ⟦ ⟦t⟧^{-1} ⟧ = t$, in accordance with the identity law that a function applied to its inverse yield the identity function. With this supposition, the form of the trace expression *g(t)* can be derived by construction:

$\forall a : \text{Activity} \bullet ⟦ \mu x.(a ; \varepsilon + x) ⟧$

$= \mu t.(⟦ a; \varepsilon + ⟦t⟧^{-1} ⟧)$ -- mapping the fixpoint

$= \mu t.(⟦a⟧ \otimes ⟦\varepsilon + ⟦t⟧^{-1}⟧)$ -- by (ts1)

$= \mu t.(⟦a⟧ \otimes \{<\downarrow>\} \otimes (⟦\varepsilon ⟧ \cup ⟦⟦t⟧^{-1}⟧))$ -- by (ta2)

$= \mu t.(⟦a⟧ \otimes \{<\downarrow>\} \otimes (⟦\varepsilon ⟧ \cup t))$ -- by the identity law

$= \mu t.(⟦a⟧ \otimes \{<\downarrow>\} \otimes (\{<>\}\cup t))$ -- by (tb1)

$= \mu t.(⟦a⟧ \otimes (\{<\downarrow>\} \otimes \{<>\}) \cup (\{<\downarrow>\} \otimes t))$

$= \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- left-distributive over $\cup$

Intuitively, this means *t* is bound over a set of traces, consisting of the traces of *a* concatenated with the choice to stop the repetition or the choice to repeat the whole cycle. Formally, this is the complete representation of infinite traces for the until-loop. For any given set of traces, an unrolling rule may be constructed for the semantic expression, to unwrap one repetition of the cycle. The whole fixpoint expression would be recursively substituted for *t*.

In the same way, the form of the trace expression *g(t)* can be derived by construction for the while-loop:

$\forall a : \text{Activity} \bullet [\![\mu x.(\varepsilon + a ; x)]\!]$

$= \mu t.([\![\varepsilon + a; [\![t]\!]^{-1}]\!])$ -- mapping the fixpoint

$= \mu t.(\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![a; [\![t]\!]^{-1}]\!]))$ -- by (ta2)

$= \mu t.(\{<\downarrow>\} \otimes (\{<>\} \cup [\![a; [\![t]\!]^{-1}]\!]))$ -- by (tb1)

$= \mu t.(\{<\downarrow>\} \otimes (\{<>\} \cup ([\![a]\!] \otimes [\![[\![t]\!]^{-1}]\!])))$ -- by (ts1)

$= \mu t.(\{<\downarrow>\} \otimes (\{<>\} \cup ([\![a]\!] \otimes t)))$ -- by the identity law

$= \mu t.((\{<\downarrow>\} \otimes \{<>\}) \cup$

$(\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$ -- left-distributive over $\cup$

$= \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$ -- by (cp1)

Intuitively, this means *t* is bound over a set of traces, consisting of the choice to finish the while-loop or the choice of the traces of *a* followed with the repetition of the whole cycle. By derivation, this is the formal representation of infinite traces for the while-loop. As for the until-loop, for any given set of traces an unrolling rule may be constructed for the semantic expression, to unwrap one repetition of the cycle. The whole fixpoint expression would be recursively substituted for *t*.

The forms of the trace expressions explained before can be used as the general cases for infinite repetitions. In addition, when the activity is empty is considered a special case, because the empty activity is transformed in the empty trace:

$\forall a : \text{Activity} \bullet [\![\mu x.(a ; \varepsilon + x)]\!] =$ **if** $(a = \varepsilon)$

then $\{<>\}$ (tr1)

else $\mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ (tr2)

$\forall a : \text{Activity} \bullet [\![\mu x.(\varepsilon + a ; x)]\!] =$ **if** $(a = \varepsilon)$

then $\{<>\}$ (tr3)

else μt.({<↓>} ∪ ({<↓>} ⊗ ([[a]] ⊗ t))) (tr4)

The until-loop is defined in (tr1) and (tr2).Whilst (tr1) treats the special case when the repetition is empty, (tr2) manages the general case where the fixpoint μx is converted into a fixpoint μt in the semantics. In the same way, the while-loop is defined by functions (tr3) and (tr4), where the function (tr3) specifies the special case for the empty activity and the function (tr4) converts the fixpoint μx into a fixpoint μt in the semantics. Consequently, an abstract syntax repetition is translated to its resultant expression in the semantic domain. For instance, the function (tr1) maps to fixpoint μt:

$\forall a \in Simple \bullet [\![\mu x.(a;\varepsilon+x)]\!]$

$\Rightarrow$ μt.([[a]] ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by tr2

$\Rightarrow$ μt.({<a>} ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by tb2

$\Rightarrow$ μt.(({<a>} ⊗ {<↓>})

∪ ({<a>} ⊗ ({<↓>} ⊗ t))) -- by distributive law

$\Rightarrow$ μt.({<a, ↓>} ∪ ({<a>} ⊗ ({<↓>} ⊗ t))) -- by cp1

$\Rightarrow$ μt.({<a, ↓>} ∪ (({<a>} ⊗ {<↓>}) ⊗ t)) -- by associative law

$\Rightarrow$ μt.({<a, ↓>} ∪ ({<a, ↓>} ⊗ t)) -- by cp1

The last example depicted the repetition of a simple task. Repetition of empty activity is solved as a special case:

$[\![\mu x.(\varepsilon;\varepsilon+x)]\!]$

$\Rightarrow$ {<>} -- by tr1

The reason is because the expression as the one in the last example may be reduced by axioms (r.1) and (s.3). In a similar situation, a *fail* alone within the repetition may be reduced by axioms (r.1) and (s.4), but can be interpreted in the semantics:

$[\![\mu x.(\phi;\ \varepsilon+x)]\!]$

$\Rightarrow$ μt.([[ϕ]] ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by tr2

$\Rightarrow$ μt.({<ϕ>} ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by tb2

$\Rightarrow$ μt.(({<ϕ>} ⊗ {<↓>})

∪ ({<ϕ>} ⊗ ({<↓>} ⊗ t))) -- by distributive law

$\Rightarrow$ μt.({<ϕ>} ∪ ({<ϕ>} ⊗ ({<↓>} ⊗ t))) -- by cp1

$\Rightarrow \mu t.(\ \{<\phi>\} \cup (\ (\{<\phi>\} \otimes \{<\downarrow>\}) \otimes t\ ))$ -- by associative law

$\Rightarrow \mu t.(\ \{<\phi>\} \cup (\ \{<\phi>\} \otimes t\ ))$ -- by cp1

$\Rightarrow \bigcup_{i=1}^{\infty} \{<\phi>\}$ -- infinite union

$\Rightarrow \{<\phi>\}$ -- set union

An expression with *succeed* instead of *fail* can follow the same derivations:

$[\![\mu x.(\sigma;\ \varepsilon + x)]\!]$

$\Rightarrow \mu t.(\ [\![\sigma]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.(\ \{<\sigma>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(\ (\ \{<\sigma>\} \otimes \{<\downarrow>\}\ )$

$\cup (\ \{<\sigma>\} \otimes (\{<\downarrow>\} \otimes t\ )))$ -- by distributive law

$\Rightarrow \mu t.(\ \{<\sigma>\} \cup (\ \{<\sigma>\} \otimes (\{<\downarrow>\} \otimes t\ )))$ -- by cp1

$\Rightarrow \mu t.(\ \{<\sigma>\} \cup (\ (\{<\sigma>\} \otimes \{<\downarrow>\}) \otimes t\ ))$ -- by associative law

$\Rightarrow \mu t.(\ \{<\sigma>\} \cup (\ \{<\sigma>\} \otimes t\ ))$ -- by cp1

$\Rightarrow \bigcup_{i=1}^{\infty} \{<\sigma>\}$ -- infinite union

$\Rightarrow \{<\sigma>\}$ -- set union

Similar examples are depicted for the while-loop structure to exemplify the behaviour defined by (tr3) and (tr4). The next example shows how an abstract syntax while-loop is translated to its resultant expression in the semantic domain:

$\forall a \in Simple \bullet [\![\mu x.(\varepsilon + a;\ x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ \{<a>\} \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\ (\{<\downarrow>\} \otimes \{<a>\}) \otimes t))$ -- by associative law

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\ \{<\downarrow, a>\} \otimes t))$ -- by cp1

Repetition of empty activity in a while-loop is treated also as a special case:

$[\![\mu x.(\varepsilon + \varepsilon;\ x)]\!]$

$\Rightarrow$ {<>} -- by tr3

A *fail* alone within the repetition may be reduced by axioms (r.1) and (s.4). This can be interpreted in the semantics:

$[\![\mu x.(\varepsilon + \phi; x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\phi]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\phi>\} \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup ((\{<\downarrow>\} \otimes \{<\phi>\}) \otimes t))$ -- by associative law

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow, \phi>\} \otimes t))$ -- by cp1

$\Rightarrow \bigcup_{i=1}^{\infty} \{<\downarrow>, <\downarrow \phi>\}$ -- infinite union

$\Rightarrow \{<\downarrow>, <\downarrow, \phi>\}$

An expression with *succeed* instead of *fail* can follow the same derivations:

$[\![\mu x.(\varepsilon + \sigma; x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\sigma]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\sigma>\} \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup ((\{<\downarrow>\} \otimes \{<\sigma>\}) \otimes t))$ -- by associative law

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow, \sigma>\} \otimes t))$ -- by cp1

$\Rightarrow \bigcup_{i=1}^{\infty} \{<\downarrow>, <\downarrow \sigma>\}$ -- infinite union

$\Rightarrow \{<\downarrow>, <\downarrow, \sigma>\}$

Additionally, just as the unrolling axioms in the abstract syntax level are necessary to expand the expressions, there are rules for unrolling in the semantic level. Rules (tr5) and (tr6) define the unrolling for the until- and while-loops:

$\forall a \in Activity \bullet \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- unrolling until-loop (tr5)

$\Leftrightarrow [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$

if $(a \neq \varepsilon)$

$\forall a \in Activity \bullet \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$ -- unrolling while-loop (tr6)

$\Leftrightarrow \{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))))$

if $(a \neq \varepsilon)$

Examples unrolling until-loop expressions by applying the rule (tr5). The first of these examples shows the unrolling of an until-loop with a simple element:

$\forall a \in Simple \bullet [\![\mu x.(a;\varepsilon+x)]\!]$

$\Rightarrow \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$ -- by tr5

$\Rightarrow [\![a]\!] \otimes (\{<\downarrow>\} \otimes (\{<>\} \cup \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$ -- by tb1

$\Rightarrow \{<a>\} \otimes (\{<\downarrow>\} \otimes (\{<>\} \cup \mu t.(\{<a>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$

-- by tb2

$\Rightarrow \{<a>\} \otimes (\{<\downarrow>\} \otimes (\{<>\} \cup$

$\mu t.((\{<a>\} \otimes \{<\downarrow>\}) \cup (\{<a>\} \otimes (\{<\downarrow>\} \otimes t)))))$

-- by distributive law of $\otimes$ over $\cup$

$\Rightarrow \{<a>\} \otimes ((\{<\downarrow>\} \otimes \{<>\}) \cup (\{<\downarrow>\} \otimes$

$\mu t.((\{<a>\} \otimes \{<\downarrow>\}) \cup (\{<a>\} \otimes (\{<\downarrow>\} \otimes t)))))$

-- by distributive law of $\otimes$ over $\cup$

$\Rightarrow \{<a>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes \mu t.(\{<a, \downarrow>\} \cup (\{<a>\} \otimes (\{<\downarrow>\} \otimes t)))))$

-- by cp1

$\Rightarrow (\{<a>\} \otimes \{<\downarrow>\}) \cup$

$(\{<a>\} \otimes (\{<\downarrow>\} \otimes \mu t.(\{<a, \downarrow>\} \cup (\{<a>\} \otimes (\{<\downarrow>\} \otimes t)))))$

-- by distributive law of $\otimes$ over $\cup$

$\Rightarrow (\{<a>\} \otimes \{<\downarrow>\}) \cup$

$(\{<a>\} \otimes \{<\downarrow>\} \otimes \mu t.(\{<a, \downarrow>\} \cup (\{<a>\} \otimes \{<\downarrow>\} \otimes t)))$

-- by associativity of $\otimes$

$\Rightarrow \{<a, \downarrow>\} \cup (\{<a, \downarrow>\} \otimes \mu t.(\{<a, \downarrow>\} \cup (\{<a, \downarrow>\} \otimes t))))$

-- by cp1

Unrolling of *fail*:

$[\![\mu x.(\phi;\ \varepsilon+x)\ ]\!]$

$\Rightarrow\ \mu t.(\ [\![\phi]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow\ [\![\phi]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![\phi]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$ -- by tr5

$\Rightarrow\ [\![\phi]\!] \otimes (\{<\downarrow>\} \otimes (\{<>\} \cup \mu t.(\ [\![\phi]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$ -- by tb1

$\Rightarrow\ \{<\phi>\} \otimes (\{<\downarrow>\} \otimes (\{<>\} \cup \mu t.(\ \{<\phi>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$

-- by tb2

$\Rightarrow\ \{<\phi>\} \otimes (\{<\downarrow>\} \otimes (\{<>\} \cup$

$\mu t.(\ (\{<\phi>\} \otimes \{<\downarrow>\}) \cup (\{<\phi>\} \otimes (\{<\downarrow>\} \otimes t)))))$

-- by distributive law of $\otimes$ over $\cup$

$\Rightarrow\ \{<\phi>\} \otimes ((\{<\downarrow>\} \otimes \{<>\}) \cup\ (\{<\downarrow>\} \otimes$

$\mu t.(\ (\{<\phi>\} \otimes \{<\downarrow>\}) \cup (\{<\phi>\} \otimes (\{<\downarrow>\} \otimes t)))))$

-- by distributive law of $\otimes$ over $\cup$

$\Rightarrow\ \{<\phi>\} \otimes (\{<\downarrow>\} \cup\ (\{<\downarrow>\} \otimes \mu t.(\ \{<\phi>\} \cup (\{<\phi>\} \otimes (\{<\downarrow>\} \otimes t)))))$

-- by cp1

$\Rightarrow\ (\{<\phi>\} \otimes \{<\downarrow>\}) \cup (\{<\phi>\} \otimes (\{<\downarrow>\} \otimes$

$\mu t.(\ \{<\phi>\} \cup (\{<\phi>\} \otimes (\{<\downarrow>\} \otimes t)))))$

-- by distributive law of $\otimes$ over $\cup$

$\Rightarrow\ (\{<\phi>\} \otimes \{<\downarrow>\}) \cup (\{<\phi>\} \otimes \{<\downarrow>\} \otimes$

$\mu t.(\ \{<\phi>\} \cup (\{<\phi>\} \otimes (\{<\downarrow>\} \otimes t))))$ -- by associativity of $\otimes$

$\Rightarrow\ \{<\phi>\} \cup \{<\phi>\}$ -- by cp1

$\Rightarrow\ \{<\phi>\}$ -- by set union

Unrolling of *succeed*:

$[\![\mu x.(\sigma;\ \varepsilon+x)\ ]\!]$

$\Rightarrow\ \mu t.(\ [\![\sigma]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow\ [\![\sigma]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![\sigma]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$ -- by tr5

$\Rightarrow$ $[\![\sigma]\!] \otimes (\{<\downarrow>\} \otimes (\{<>\} \cup \mu t.([\![\sigma]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$ -- by tb1

$\Rightarrow$ $\{<\sigma>\} \otimes (\{<\downarrow>\} \otimes (\{<>\} \cup \mu t.(\{<\sigma>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$

-- by tb2

$\Rightarrow$ $\{<\sigma>\} \otimes (\{<\downarrow>\} \otimes (\{<>\} \cup$

$\mu t.((\{<\sigma>\} \otimes \{<\downarrow>\}) \cup (\{<\sigma>\} \otimes (\{<\downarrow>\} \otimes t))))$

-- by distributive law of $\otimes$ over $\cup$

$\Rightarrow$ $\{<\sigma>\} \otimes ((\{<\downarrow>\} \otimes \{<>\}) \cup (\{<\downarrow>\} \otimes$

$\mu t.((\{<\sigma>\} \otimes \{<\downarrow>\}) \cup (\{<\sigma>\} \otimes (\{<\downarrow>\} \otimes t))))$

-- by distributive law of $\otimes$ over $\cup$

$\Rightarrow$ $\{<\sigma>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes \mu t.(\{<\sigma>\} \cup (\{<\sigma>\} \otimes (\{<\downarrow>\} \otimes t)))))$

-- by cp1

$\Rightarrow$ $(\{<\sigma>\} \otimes \{<\downarrow>\}) \cup (\{<\sigma>\} \otimes (\{<\downarrow>\} \otimes$

$\mu t.(\{<\sigma>\} \cup (\{<\sigma>\} \otimes (\{<\downarrow>\} \otimes t))))$

-- by distributive law of $\otimes$ over $\cup$

$\Rightarrow$ $(\{<\sigma>\} \otimes \{<\downarrow>\}) \cup (\{<\sigma>\} \otimes \{<\downarrow>\} \otimes$

$\mu t.(\{<\sigma>\} \cup (\{<\sigma>\} \otimes (\{<\downarrow>\} \otimes t))))$ -- by associativity of $\otimes$

$\Rightarrow$ $\{<\sigma>\} \cup \{<\sigma>\}$ -- by cp1

$\Rightarrow$ $\{<\sigma>\}$ -- by set union

Finally, the examples unrolling while-loop expressions by applying the rule (tr6) are presented below. Unrolling an expression with a simple element:

$\forall a \in Simple \bullet [\![\mu x.(\varepsilon + a; x)]\!]$

$\Rightarrow$ $\mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$ -- by tr4

$\Rightarrow$ $\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))))$ -- by tr6

$\Rightarrow$ $\{<\downarrow>\} \otimes (\{<>\} \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))))$ -- by tb1

$\Rightarrow$ $\{<\downarrow>\} \otimes (\{<>\} \cup (\{<a>\} \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<a>\} \otimes t)))))$

-- by tb2

$\Rightarrow$ ({<↓>} $\otimes$ {<>}) $\cup$ ({<↓>}$\otimes$ ({<a>} $\otimes$

μt.({<↓>} $\cup$ ({<↓>} $\otimes$ ({<a>} $\otimes$ t)))))

-- by distributive law of $\otimes$ over $\cup$

$\Rightarrow$ ({<↓>} $\otimes$ {<>}) $\cup$ ({<↓>}$\otimes$ {<a>} $\otimes$

μt.({<↓>} $\cup$ ({<↓>} $\otimes$ {<a>} $\otimes$ t))) -- by associativity of $\otimes$

$\Rightarrow$ {<↓>} $\cup$ ({<↓, a>} $\otimes$ μt.({<↓>} $\cup$ ({<↓, a>} $\otimes$ t))) -- by cp1

Unrolling of *fail* in the while-loop:

$[\![\mu x.(\varepsilon+\phi; x)]\!]$

$\Rightarrow$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ ($[\![\phi]\!]$ $\otimes$ t))) -- by tr4

$\Rightarrow$ {<↓>} $\otimes$ ($[\![\varepsilon]\!]$ $\cup$ ($[\![\phi]\!]$ $\otimes$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ ($[\![\phi]\!]$ $\otimes$ t))))) -- by tr6

$\Rightarrow$ {<↓>} $\otimes$ ({<>} $\cup$ ($[\![\phi]\!]$ $\otimes$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ ($[\![\phi]\!]$ $\otimes$ t))))) -- by tb1

$\Rightarrow$ {<↓>} $\otimes$ ({<>} $\cup$ ({<ϕ>} $\otimes$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ ({<ϕ>} $\otimes$ t)))))

-- by tb2

$\Rightarrow$ ({<↓>} $\otimes$ {<>}) $\cup$ ({<↓>} $\otimes$ ({<ϕ>} $\otimes$

μt.({<↓>} $\cup$ ({<↓>} $\otimes$ ({<ϕ>} $\otimes$ t)))))

-- by distributive law of $\otimes$ over $\cup$

$\Rightarrow$ ({<↓>} $\otimes$ {<>}) $\cup$ ({<↓>} $\otimes$ {<ϕ>} $\otimes$

μt.({<↓>} $\cup$ ({<↓>} $\otimes$ {<ϕ>} $\otimes$ t))) -- by associative law

$\Rightarrow$ {<↓>} $\cup$ ({<↓, ϕ>} $\otimes$ μt.({<↓>} $\cup$ ({<↓, ϕ>} $\otimes$ t))) -- by cp1

Unrolling of *succeed* in the while-loop:

$[\![\mu x.(\varepsilon+\sigma; x)]\!]$

$\Rightarrow$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ ($[\![\sigma]\!]$ $\otimes$ t))) -- by tr4

$\Rightarrow$ {<↓>} $\otimes$ ($[\![\varepsilon]\!]$ $\cup$ ($[\![\sigma]\!]$ $\otimes$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ ($[\![\sigma]\!]$ $\otimes$ t))))) -- by tr6

$\Rightarrow$ {<↓>} $\otimes$ ({<>} $\cup$ ($[\![\sigma]\!]$ $\otimes$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ ($[\![\sigma]\!]$ $\otimes$ t))))) -- by tb1

$\Rightarrow$ {<↓>} $\otimes$ ({<>} $\cup$ ({<σ>} $\otimes$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ ({<σ>} $\otimes$ t)))))

-- by tb2

$\Rightarrow$ ({<↓>} ⊗ {<>}) ∪ ({<↓>} ⊗ ({<σ>} ⊗

μt.({<↓>} ∪ ({<↓>} ⊗ ({<σ>} ⊗ t)))))

-- by distributive law of ⊗ over ∪

$\Rightarrow$ ({<↓>} ⊗ {<>}) ∪ ({<↓>} ⊗ {<σ>} ⊗

μt.({<↓>} ∪ ({<↓>} ⊗ {<σ>} ⊗ t))) -- by associative law

$\Rightarrow$ {<↓>} ∪ ({<↓,σ>} ⊗ μt.({<↓>} ∪ ({<↓, σ>} ⊗ t))) -- by cp1

It is important to remember that each unrolled expression is semantically equivalent to the expression in the first examples of this section (i.e., the corresponding examples without the unrolling). The general proof of soundness will be provided in Chapter 7.

6.5.6 Tracing the Unpacking of Activity

Encapsulation provides a scope within the task boundary, allowing to establish a limit for the effect of *Exit.* A subscript $_T$ is allowed to be added in the task boundary syntax, to facilitate the identification of the compound tasks:

$$\forall a : Activity \bullet [\![\{a\}_T]\!] = unpack([\![a]\!]) \qquad \text{(tu1)}$$

Unpacking an activity eliminates the task boundary promoting the activity to the higher level. σ is understood as early success within the boundary and is eliminated for the higher level. On the other hand, ϕ remains in the higher level propagating the failure.

The encapsulation of basic elements is depicted in the next examples. Encapsulation of a simple task:

$\forall x \in Simple \bullet [\![\{x\}_T]\!]$

$\Rightarrow unpack([\![x]\!])$ -- by tu1

$\Rightarrow unpack(\{< x >\})$ -- by tb2

$\Rightarrow \{lift(x)\}$ -- by up1

$\Rightarrow \{x.(lift <>)\}$ -- by li3

$\Rightarrow \{x.<>\}$ -- by li1

$\Rightarrow \{< x >\}$ -- cons operator

The encapsulation of the empty activity results in the empty activity:

$[\![\{\varepsilon\}_T]\!]$

$\Rightarrow unpack([\![\varepsilon]\!])$ -- by tu1

$\Rightarrow unpack(\{<>\})$ -- by tb1

$\Rightarrow \{lift <>\}$ -- by up1

$\Rightarrow \{<>\}$ -- by li1

The use of *succeed* and *fail* is the main reason of the encapsulation in the abstract syntax representation. *Succeed* in an encapsulation is transformed in an empty activity:

$[\![\{\sigma\}_T]\!]$

$\Rightarrow unpack([\![\sigma]\!])$ -- by tu1

$\Rightarrow unpack(\{<\sigma>\})$ -- by tb2

$\Rightarrow \{lift <\sigma>\}$ -- by up1

$\Rightarrow \{<>\}$ -- by li2

More interesting results can be seen using *succeed* in combination with a binary operator such as sequential composition, being σ the right operand:

$\forall x \in Simple \bullet [\![\{x; \sigma\}_T]\!]$

$\Rightarrow unpack([\![x; \sigma]\!])$ -- by tu1

$\Rightarrow unpack([\![x]\!] \otimes [\![\sigma]\!])$ -- by ts1

$\Rightarrow unpack(\{<x>\} \otimes [\![\sigma]\!])$ -- by tb2

$\Rightarrow unpack(\{<x>\} \otimes \{<\sigma>\})$ -- by tb2

$\Rightarrow unpack(\{<x>\#<\sigma>\})$ -- by cp1

$\Rightarrow unpack(\{<x,\sigma>\})$ -- by tc6

$\Rightarrow \{lift <x,\sigma>\}$ -- by up1

$\Rightarrow \{x.(lift <\sigma>)\}$ -- by li3

$\Rightarrow \{x.<>\}$ -- by li2

$\Rightarrow \{<x>\}$ -- cons operator

Placing *succeed* over the left produces first the elimination of the simple task in the right and, subsequently, the transformation of σ into the empty activity, when the lower level is eliminated:

$\forall x \in Simple \bullet [\![\{\sigma, x\}_T]\!]$

$\Rightarrow unpack([\![\sigma; x]\!])$ -- by tu1

$\Rightarrow unpack([\![\sigma]\!] \otimes [\![x]\!])$ -- by ts1

$\Rightarrow unpack([\![\sigma]\!] \otimes \{< x >\})$ -- by tb2

$\Rightarrow unpack(\{< \sigma >\} \otimes \{< x >\})$ -- by tb2

$\Rightarrow unpack(\{< \sigma > \# < x >\})$ -- by cp1

$\Rightarrow unpack(\{< \sigma >\})$ -- by tc2

$\Rightarrow \{lift < \sigma >\}$ -- by up1

$\Rightarrow \{<>\}$ -- by li2

For selection being *succeed* in the right side:

$\forall x \in Simple \bullet [\![\{x + \sigma\}_T]\!]$

$\Rightarrow unpack([\![x + \sigma]\!])$ -- by tu1

$\Rightarrow unpack(\{<\downarrow>\} \otimes [\![x]\!] \cup [\![\sigma]\!])$ -- by ta2

$\Rightarrow unpack(\{<\downarrow>\} \otimes \{< x >\} \cup \{< \sigma >\})$ -- by tb2

$\Rightarrow unpack(\{<\downarrow>\} \otimes \{< x >, < \sigma >\})$ -- union of traces

$\Rightarrow unpack(\{<\downarrow> \# < x >, <\downarrow> \# < \sigma >\})$ -- by cp1

$\Rightarrow unpack(\{<\downarrow, x >, <\downarrow, \sigma >\})$ -- by tc5

$\Rightarrow \{lift <\downarrow, x >, lift <\downarrow \sigma >\}$ -- by up1

$\Rightarrow \{\downarrow.(lift < x >), \downarrow.(lift < \sigma >)\}$ -- by li3

$\Rightarrow \{\downarrow.(lift < x >), \downarrow.(lift <>)\}$ -- by li2

$\Rightarrow \{\downarrow.x.(lift <>), \downarrow.(lift <>)\}$ -- by li3

$\Rightarrow \{\downarrow.x.<>, \downarrow.<>\}$ -- by li1

$\Rightarrow \{<\downarrow, x>, <\downarrow>\}$ -- cons operator

For parallel composition with *succeed*:

$\forall x \in Simple \bullet [\![\{x \parallel \sigma\}_T]\!]$

$\Rightarrow unpack([\![x \parallel \sigma]\!])$ -- by tu1

$\Rightarrow unpack([\![x]\!] // [\![\sigma]\!])$ -- by tp1

$\Rightarrow unpack(\{< x >\} // \{< \sigma >\})$ -- by tb2

$\Rightarrow unpack(\cup\{< x > \sim < \sigma >\})$ -- by di1

$\Rightarrow unpack(\cup\{\{< \sigma >\}\})$ -- by ti4

$\Rightarrow unpack(\{< \sigma >\})$

$\Rightarrow \{lift <>\}$ -- by up1

$\Rightarrow \{<>\}$ -- by li1

Both selection and parallel composition are defined in the abstract syntax as commutative, therefore inverting the order of the operands for the expressions in the last two examples will generate the same set of traces for each expression. Whilst σ is eliminated by the application of the unpacking function, ϕ is promoted to the higher level, propagating the fail and eliminating the traces that are combined with it:

$[\![\{\phi\}_T]\!]$

$\Rightarrow unpack([\![\phi]\!])$ -- by tu1

$\Rightarrow unpack(\{< \phi >\})$ -- by tb2

$\Rightarrow \{lift < \phi >\}$ -- by up1

$\Rightarrow \{\phi.(lift <>)\}$ -- by li3

$\Rightarrow \{\phi. <>\}$ -- by li1

$\Rightarrow \{< \phi >\}$ -- cons operator

The next example depicts the use of *fail* in combination with a binary operator such as sequential composition, being *fail* the operand on the right:

$\forall x \in Simple \bullet [\![\{x; \phi\}_T]\!]$

$\Rightarrow unpack([\![x; \phi]\!])$ -- by tu1

$\Rightarrow unpack([\![x]\!] \otimes [\![\phi]\!])$ -- by ts1

$\Rightarrow unpack(\{< x >\} \otimes [\![\phi]\!])$ -- by tb2

$\Rightarrow unpack(\{< x >\} \otimes \{< \phi >\})$ -- by tb2

$\Rightarrow unpack(\{< x > \# < \phi >\})$ -- by cp1

$\Rightarrow unpack(\{< x, \phi >\})$ -- by tc6

$\Rightarrow \{lift < x, \phi >\}$ -- by up1

$\Rightarrow \{x.(lift < \phi >)\}$ -- by li3

$\Rightarrow \{x.\phi.(lift <>)\}$ -- by li3

$\Rightarrow \{x.\phi. <>\}$ -- by li1

$\Rightarrow \{< x, \phi >\}$ -- cons operator

Placing *fail* over the left produces the elimination of the simple task in the right:

$\forall x \in Simple \bullet [\![\{\phi; x\}_T]\!]$

$\Rightarrow unpack([\![\phi; x]\!])$ -- by tu1

$\Rightarrow unpack([\![\phi]\!] \otimes [\![x]\!])$ -- by ts1

$\Rightarrow unpack([\![\phi]\!] \otimes \{< x >\})$ -- by tb2

$\Rightarrow unpack(\{< \phi >\} \otimes \{< x >\})$ -- by tb2

$\Rightarrow unpack(\{< \phi > \# < x >\})$ -- by cp1

$\Rightarrow unpack(\{< \phi >\})$ -- by tc3

$\Rightarrow \{lift < \phi >\}$ -- by up1

$\Rightarrow \{\phi.(lift <>)\}$ -- by li3

$\Rightarrow \{\phi. <>\}$ -- by li1

$\Rightarrow \{< \phi >\}$ -- cons operator

For selection with *fail*:

$\forall x \in Simple \bullet [\![\{x + \phi\}_T]\!]$

$\Rightarrow unpack([\![x + \phi]\!])$ -- by tu1

$\Rightarrow unpack(\{<\downarrow>\} \otimes [\![x]\!] \cup [\![\phi]\!])$ -- by ta2

$\Rightarrow unpack(\{<\downarrow>\} \otimes \{< x >\} \cup \{< \phi >\})$ -- by tb2

$\Rightarrow unpack(\{<\downarrow>\} \otimes \{< x >, < \phi >\})$ -- union of traces

$\Rightarrow unpack(\{<\downarrow># < x >, <\downarrow># < \phi >\})$ -- by cp1

$\Rightarrow unpack(\{<\downarrow, x >, <\downarrow, \phi >\})$ -- by tc5

$\Rightarrow \{lift <\downarrow, x >, lift <\downarrow, \phi >\}$ -- by up1

$\Rightarrow \{\downarrow.(lift < x >), \downarrow.(lift < \phi >)\}$ -- by li3

$\Rightarrow \{\downarrow.x.(lift <>), \downarrow.\phi.(lift <>)\}$ -- by li3

$\Rightarrow \{\downarrow.x.<>, \downarrow.\phi.<>\}$ -- by li1

$\Rightarrow \{<\downarrow, x >, <\downarrow, \phi >\}$ -- cons operator

For parallel composition with *fail*:

$\forall x \in Simple \bullet [\![\{x \parallel \phi\}_T]\!]$

$\Rightarrow unpack([\![x \parallel \phi]\!])$ -- by tu1

$\Rightarrow unpack([\![x]\!] // [\![\phi]\!])$ -- by tp1

$\Rightarrow unpack(\{< x >\} // \{< \phi >\})$ -- by tb2

$\Rightarrow unpack(\cup\{< x > \sim < \phi >\})$ -- by di1

$\Rightarrow unpack(\cup\{\{< \phi >\}\})$ -- by ti6

$\Rightarrow unpack(\{< \phi >\})$

$\Rightarrow \{lift < \phi >\}$ -- by up1

$\Rightarrow \{\phi.(lift <>)\}$ -- by li3

$\Rightarrow \{\phi.<>\}$ -- by li1

$\Rightarrow \{< \phi >\}$ -- cons operator

6.6 Summary

This chapter described the simple denotational semantics for the abstract task algebra presented in the previous chapter. The semantics were presented in terms of trace sets representing all possible complete execution paths for a system. The trace semantics for the algebra was explained using examples showing combinations of the basic

elements. The soundness of the axioms from chapter 4 and congruence properties are presented in the next chapter.

Chapter 7:
Soundness for the Semantics of Tasks

The previous chapter described the denotational semantics in terms of traces of the constructions in the abstract task algebra. In this chapter, the trace semantics are used to prove the soundness of the axioms for the task algebra illustrated in the chapter 4. Some examples of congruence properties are demonstrated for the algebra. A full listing of congruence properties is defined in Appendix B.

7.1 Introduction

Soundness is a basic requisite in an algebra to prove that the axioms are really trace equivalent; i.e. that the axioms are true for all the elements of the algebra. The next section demonstrates the soundness of the task algebra. Each of the axioms of the task algebra is proved to hold, based on the given trace semantics, and on the fundamental properties of the semantic functions.

7.2 Soundness

In this section, the soundness of the abstract task algebra is proved. The soundness of an algebra means to prove that syntactic constructions which are equivalent, according to the axioms of the algebra, are also equivalent in the semantics. In other words, every theorem that is provable in the axioms is also provable in all semantic interpretations. Informally it can be said that it is impossible to derive contradictory propositions [141]. The soundness of the task algebra is proved for all their axioms using the semantic definitions and a set of basic properties for the semantic functions A.1 to A.8. The set of basic properties are as follows:

• A.1 Associativity of ⊗

• A.2 Distribution of ⊗ over union

• A.3 Identity for ⊗

• A.4 Associativity of //

• A.5 Commutativity of //

• A.6 Distribution of // over union

• A.7 Identity for //

• A.8 Distribution of unpack over union

These properties are assumed to hold here, and are derived in Appendix A. In property A.1, lemma 1 was proved by mathematical induction defining as base cases arbitrary traces of length 0 and 1. There are 5 base cases: $<>$, $<\phi>$, $<\sigma>$, $<\downarrow>$ and identifier. The base case of commit needed to be proved with a more profound analysis. It was necessary to analyse the two possible cases for *trace2*: $\downarrow$*.rest* and *a.rest*. The $\downarrow$*.rest* case for *trace2* was proved again by mathematical induction over the length of *rest* with base case of length 0 and 1. In the case of *a.rest* the proof follows directly from the properties.

7.2.1 Sequential composition

The task algebra defines for the following axioms for sequential composition: associative sequence (s.1), right distributivity of sequence over selection (s.2), empty sequence (s.3), early end with fail (s.4), and early end with succeed (s.5). In this section, each of these axioms is proved sound by deriving the semantics of each equivalence expression in the axioms.

7.2.1.1 Soundness for the associative sequence axiom

$\forall a, b, c \in Activity \bullet$ ⟦a; (b; c)⟧ = ⟦(a; b); c⟧

$\Rightarrow$ ⟦a⟧ $\otimes$ ⟦(b; c)⟧ = ⟦(a; b)⟧ $\otimes$ ⟦c⟧ -- by ts1

$\Rightarrow$ ⟦a⟧ $\otimes$ (⟦b⟧ $\otimes$ ⟦c⟧) = (⟦a⟧ $\otimes$ ⟦b⟧) $\otimes$ ⟦c⟧ -- by ts1

$\Rightarrow$ ⟦a⟧ $\otimes$ (⟦b⟧ $\otimes$ ⟦c⟧) = ⟦a⟧ $\otimes$ (⟦b⟧ $\otimes$ ⟦c⟧) -- by A.1

As can be seen the soundness for the associative sequence is proved by applying the mapping function for tracing a sequence of *Activity*, followed by the associativity of concatenated product (property A.1).

7.2.1.2 Soundness for the right distributivity of sequence over selection axiom

$\forall a, b, c \in Activity \bullet$ ⟦(a + b); c⟧ = ⟦(a; c) + (b; c)⟧

$\Rightarrow$ ⟦a + b⟧ $\otimes$ ⟦c⟧ = ⟦(a; c) + (b; c)⟧ -- by ts1

$\Rightarrow$ {$<\downarrow>$} $\otimes$ (⟦a⟧ $\cup$ ⟦b⟧) $\otimes$ ⟦c⟧ = {$<\downarrow>$} $\otimes$ (⟦a; c ⟧ $\cup$ ⟦b; c⟧) -- by ta2

$\Rightarrow$ {$<\downarrow>$} $\otimes$ (⟦a⟧ $\cup$ ⟦b⟧) $\otimes$ ⟦c⟧ = {$<\downarrow>$} $\otimes$ ((⟦a⟧ $\otimes$ ⟦c⟧) $\cup$ (⟦b⟧ $\otimes$ ⟦c⟧)) -- by ts1

$\Rightarrow$ {$<\downarrow>$} $\otimes$ ((⟦a⟧ $\otimes$ ⟦c⟧) $\cup$ (⟦b⟧ $\otimes$ ⟦c⟧))

= {$<\downarrow>$} $\otimes$ ((⟦a⟧ $\otimes$ ⟦c⟧) $\cup$ (⟦b⟧ $\otimes$ ⟦c⟧)) -- by A.2

The soundness for this axiom is proved applying the mapping function for tracing a sequence of *Activity* in combination with the application of the general mapping function for tracing a selection of *Activity*. Finally, the transformation by distribution of concatenated product over union is applied to the left expression to obtain the equivalent semantics.

7.2.1.3 Soundness for the empty sequence axiom

$\forall a \in Activity \bullet [\![a; \varepsilon]\!] = [\![\varepsilon; a]\!] = [\![a]\!]$

$\Rightarrow [\![a]\!] \otimes [\![\varepsilon]\!] = [\![\varepsilon]\!] \otimes [\![a]\!] = [\![a]\!]$ -- by ts1

$\Rightarrow [\![a]\!] \otimes \{<>\} = \{<>\} \otimes [\![a]\!] = [\![a]\!]$ -- by tb1

$\Rightarrow [\![a]\!] = [\![a]\!] = [\![a]\!]$ -- by A.3

In this case, the expression are reduced by applying initially the mapping function for tracing a sequence of *Activity* followed by the rule for mapping the empty element, and the set containing the empty sequence is eliminated by identity for the concatenated product.

7.2.1.4 Soundness for the exit with failure axiom

$\forall a \in Activity \bullet [\![\phi; a]\!] = [\![\phi]\!]$

$\Rightarrow [\![\phi]\!] \otimes [\![a]\!] = [\![\phi]\!]$ -- by ts1

$\Rightarrow \{<\phi>\} \otimes [\![a]\!] = \{<\phi>\}$ -- by tb2

$\Rightarrow \{<\phi>\} \otimes \{t_1, t_2, \ldots, t_n\} = \{<\phi>\}$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\phi>\} = \{<\phi>\}$ $\bigcup_{i=1}^{n} \{< \phi > \# t_i\}$

Exit with failure is proved deriving the left expression by mapping the sequence and applying the distributed union for the trace concatenation of $<\phi>$ and $\{t_1, t_2, \ldots, t_n\}$. The right-side expression is mapped by the second rule for tracing basic elements.

7.2.1.5 Soundness for the exit with success axiom

$\forall a \in Activity \bullet [\![\sigma; a]\!] = [\![\sigma]\!]$

$\Rightarrow [\![\sigma]\!] \otimes [\![a]\!] = [\![\sigma]\!]$ -- by ts1

$\Rightarrow \{<\sigma>\} \otimes [\![a]\!] = \{<\sigma>\}$ -- by tb2

$\Rightarrow \{<\sigma>\} \otimes \{t_1, t_2, \ldots, t_n\} = \{<\sigma>\}$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\sigma>\} = \{<\sigma>\}$ $\bigcup_{i=1}^{n} \{< \sigma > \# t_i\}$

As in 6.2.1.4, exit with success is proved deriving the left expression by mapping the sequence and applying the distributed union for the trace concatenation of <σ> and $\{t_1, t_2, \ldots, t_n\}$. The right-side expression is mapped by the second rule for tracing basic elements.

7.2.2 Selection

The task algebra defines the axioms of associative selection (sel.1), commutative selection (sel. 2), and idempotent selection (sel. 3). The soundness of these axioms is proved below.

7.2.2.1 Soundness for the associative selection axiom

$\forall a, b, c \in Activity \bullet [\![(a + b) + c]\!] = [\![a+ (b + c)]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![a + b]\!] \cup [\![c]\!]) = \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b + c]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])) \cup [\![c]\!])$

$= \{<\downarrow>\} \otimes ([\![a]\!] \cup (\{<\downarrow>\} \otimes ([\![b]\!] \cup [\![c]\!])))$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes ((((\{<\downarrow>\} \otimes [\![a]\!]) \cup (\{<\downarrow>\} \otimes [\![b]\!]))) \cup [\![c]\!])$

$= \{<\downarrow>\} \otimes ([\![a]\!] \cup ((\{<\downarrow>\} \otimes [\![b]\!]) \cup (\{<\downarrow>\} \otimes [\![c]\!])))$ -- by A.2

$\Rightarrow (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes [\![a]\!]) \cup (\{<\downarrow>\} \otimes [\![b]\!]))) \cup (\{<\downarrow>\} \otimes [\![c]\!])$

$= (\{<\downarrow>\} \otimes [\![a]\!]) \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes [\![b]\!]) \cup (\{<\downarrow>\} \otimes [\![c]\!])))$

-- by A.2

$\Rightarrow ((\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes [\![a]\!])) \cup (\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes [\![b]\!])))$

$\cup (\{<\downarrow>\} \otimes [\![c]\!])$

$= (\{<\downarrow>\} \otimes [\![a]\!]) \cup ((\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes [\![b]\!]))$

$\cup (\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes [\![c]\!])))$ -- by A.2

$\Rightarrow ((\{<\downarrow>\} \otimes [\![a]\!]) \cup (\{<\downarrow>\} \otimes [\![b]\!])) \cup (\{<\downarrow>\} \otimes [\![c]\!])$

$= (\{<\downarrow>\} \otimes [\![a]\!]) \cup ((\{<\downarrow>\} \otimes [\![b]\!]) \cup (\{<\downarrow>\} \otimes [\![c]\!]))$ -- by cp1

Soundness for the associative selection axiom is proved mapping initially the selection into the semantics, followed by applying distribution of concatenated product over union to successively in order to place each activity in concatenation product with the commit symbol. Finally, the rule for concatenated product of trace sets is applied to the expressions to eliminate the case where a redundant commit exists.

7.2.2.2 Soundness for the commutative selection axiom

$\forall a, b \in Activity \bullet [\![a + b]\!] = [\![b + a]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!]) = \{<\downarrow>\} \otimes ([\![b]\!] \cup [\![a]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!]) = \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])$

-- by commutativity of union

As can be expected, to solve the soundness for the commutative selection axiom initially is applied the general mapping function for tracing a selection of *Activity*. After that, the commutativity for set union is used.

7.2.2.3 Soundness for the idempotent selection axiom

$\forall a \in Activity \bullet [\![a + a]\!] = [\![a]\!]$

$\Rightarrow [\![a]\!] = [\![a]\!]$ -- by ta1

Soundness for the idempotent selection axiom is resolved in one step by applying the special case of the mapping function for tracing a selection of *Activity*.

7.2.3 Parallel composition

Parallel composition defines six axioms in the task algebra: associative parallel composition (p.1), commutative parallel composition (p.2), right distributivity of concurrency over selection (p.3), instant synchronisation (p.4), fail in parallel composition (p.5), and succeed in parallel composition (p.6).

7.2.3.1 Soundness for the associative parallel composition axiom

$\forall a, b, c \in Activity \bullet [\![(a \parallel b) \parallel c]\!] = [\![a \parallel (b \parallel c)]\!]$

$\Rightarrow [\![(a \parallel b)]\!] // [\![c]\!] = [\![a]\!] // [\![(b \parallel c)]\!]$ -- by tp1

$\Rightarrow ([\![a]\!] // [\![b]\!]) // [\![c]\!] = [\![a]\!] // ([\![b]\!] // [\![c]\!])$ -- by tp1

$\Rightarrow [\![a]\!] // ([\![b]\!] // [\![c]\!]) = [\![a]\!] // ([\![b]\!] // [\![c]\!])$ -- by A.4

The present axiom is proved to be sound after tracing a parallel composition of *Activity* and deriving by associativity of the distributed interleaving of trace sets.

7.2.3.2 Soundness for the commutative parallel composition axiom

$\forall a, b \in Activity \bullet [\![a \parallel b]\!] = [\![b \parallel a]\!]$

$\Rightarrow [\![a]\!] // [\![b]\!] = [\![b]\!] // [\![a]\!]$ -- by tp1

$\Rightarrow [\![a]\!] // [\![b]\!] = [\![a]\!] // [\![b]\!]$ -- by A.5

The axiom for commutative parallel composition is derived initially by mapping from parallel composition to the distributed interleaving of trace sets. Subsequently, the elements in the right-side expression are interchanged applying the commutativity for the distributed interleaving of trace sets.

7.2.3.3 Soundness for the right distributivity of concurrency over selection axiom

$\forall a, b, c \in Activity \bullet$ ⟦(a + b) || c⟧ = ⟦(a || c) + (b || c)⟧

⇒ ⟦a + b⟧ // ⟦c⟧ = ⟦(a || c) + (b || c)⟧ -- by tp1

⇒ {<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧) // ⟦c⟧ = {<↓>} ⊗ (⟦a || c ⟧ ∪ ⟦b || c⟧) -- by ta2

⇒ {<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧) // ⟦c⟧ = {<↓>} ⊗ ((⟦a⟧ // ⟦c⟧) ∪ (⟦b⟧ // ⟦c⟧))

-- by tp1

⇒ {<↓>} ⊗ ((⟦a⟧ // ⟦c⟧) ∪ (⟦b⟧ // ⟦c⟧))

= {<↓>} ⊗ ((⟦a⟧ // ⟦c⟧) ∪ (⟦b⟧ // ⟦c⟧)) -- by A.6

In this case, the rule for tracing a selection of *Activity* and tracing the parallel composition to the distributed interleaving of trace sets are applied to both expressions. The final transformation is made by distribution of distributed interleaving over union.

7.2.3.4 Soundness for the instant synchronisation axiom

$\forall a \in Activity \bullet$ ⟦a || ε⟧ = ⟦a⟧

⇒ ⟦a⟧ // ⟦ε⟧ = ⟦a⟧ -- by tp1

⇒ ⟦a⟧ // {<>} = ⟦a⟧ -- by tb1

⇒ ⟦a⟧ = ⟦a⟧ -- by A.7

The instant synchronisation axiom is proved soundness by the identity rule for the distributed interleaving of trace sets, once the parallel composition has been mapped to distributive interleaving and the empty element has been mapped to the empty trace.

7.2.3.5 Soundness for instant failure in parallel composition axiom

$\forall a \in Activity \bullet$ ⟦a || ϕ⟧ = ⟦ϕ⟧

⇒ ⟦a⟧ // ⟦ϕ⟧ = ⟦ϕ⟧ -- by tp1

⇒ ⟦a⟧ // {<ϕ>} = {<ϕ>} -- by tb2

$\Rightarrow \{t_1, t_2, \ldots, t_n\} // \{<\phi>\} = \{<\phi>\}$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\phi>\} = \{<\phi>\}$ $\bigcup_{i=1}^{n} \{t_i \sim < \phi >\}$

Soundness for the fail in parallel composition axiom is proved by mapping from parallel composition to the distributed interleaving of trace sets, followed by mapping the fail element using the second rule for tracing basic elements. Finally, $[\![a]\!]$ is represented as a set of traces $\{t_1, t_2, \ldots, t_n\}$ and each t_i applied, using distributed union, in interleaving with the fail trace.

7.2.3.6 Soundness for instant success in parallel composition axiom

$\forall a \in Activity \bullet [\![a \parallel \sigma]\!] = [\![\sigma]\!]$

$\Rightarrow [\![a]\!] // [\![\sigma]\!] = [\![\sigma]\!]$ -- by tp1

$\Rightarrow [\![a]\!] // \{<\sigma>\} = \{<\sigma>\}$ -- by tb2

$\Rightarrow \{t_1, t_2, \ldots, t_n\} // \{<\sigma>\} = \{<\sigma>\}$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\sigma>\} = \{<\sigma>\}$ $\bigcup_{i=1}^{n} \{t_i \sim < \sigma >\}$

The same process described in 6.2.4.5 is applied here, with the only difference that succeed is used instead of fail.

7.2.4 Repetition

Repetition is represented in the task algebra in the form of an until- and while-loop. It has two axioms which recursively unfold the expression contained: unrolling one cycle of until-loop repetition (r.1) and Unrolling one cycle of while-loop repetition (r.2).

7.2.4.1 Soundness for the unrolling one cycle of until-loop repetition axiom

$\forall a \in Activity \bullet [\![\mu x.(a \,;\, \varepsilon + x)]\!] = [\![a; \varepsilon + \mu x.(a \,;\, \varepsilon + x)]\!]$

Case $a=\varepsilon$:

$\Rightarrow \{<>\} = [\![\varepsilon; \varepsilon + \mu x.(\varepsilon \,;\, \varepsilon + x)]\!]$ -- by tr1

$\Rightarrow \{<>\} = [\![\varepsilon]\!] \otimes [\![\varepsilon + \mu x.(\varepsilon \,;\, \varepsilon + x)]\!]$ -- by ts1

$\Rightarrow \{<>\} = [\![\varepsilon]\!] \otimes [\![\varepsilon + \varepsilon]\!]$ -- by r1 and s3

$\Rightarrow \{<>\} = [\![\varepsilon]\!] \otimes [\![\varepsilon]\!]$ -- by sel3

$\Rightarrow \{<>\} = \{<>\} \otimes \{<>\}$ -- by tb1

$\Rightarrow \{<>\} = \{<>\#<>\}$ -- by cp1

$\Rightarrow \{<>\} = \{<>\}$ -- by tc1

Otherwise:

$\Rightarrow \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t))) = [\![a;\ \varepsilon + \mu x.(a\ ;\ \varepsilon + x)\ ]\!]$ -- by tr2

$\Rightarrow [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$

$= [\![a;\ \varepsilon + \mu x.(a\ ;\ \varepsilon + x)\ ]\!]$ -- by tr5

$\Rightarrow [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$

$= [\![a]\!] \otimes [\![\varepsilon + \mu x.(a\ ;\ \varepsilon + x)\ ]\!]$ -- by ts1

$\Rightarrow [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$

$= [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![\mu x.(a\ ;\ \varepsilon + x)\ ]\!]))$ -- by ta2

$\Rightarrow [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$

$= [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))$

-- by tr2

In order to prove the soundness of this axiom, the expressions on both sides have to be mapped to the trace semantics by tracing the repetitions. The left-side expression has to be unrolled by using the unrolling until-loop rule for the semantics and, subsequently, both expressions are translated, as far as possible, to the same trace semantics.

7.2.4.2 Soundness for the unrolling one cycle of while-loop repetition axiom

$\forall a \in Activity \bullet [\![\ \mu x.(\varepsilon + a\ ;\ x)\ ]\!] = [\![\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x)\ ]\!]$

Case a=ε:

$\Rightarrow \{<>\} = [\![\varepsilon + \varepsilon;\ \mu x.(\varepsilon + \varepsilon\ ;\ x)\ ]\!]$ -- by tr3

$\Rightarrow \{<>\} = [\![\varepsilon + \varepsilon]\!]$ -- by r2 and s3

$\Rightarrow \{<>\} = [\![\varepsilon]\!]$ -- by sel3

$\Rightarrow \{<>\} = \{<>\}$ -- by tb1

Otherwise:

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))) = [\![\varepsilon + a; \mu x.(\varepsilon + a ; x)]\!]$ -- by tr4

$\Rightarrow \{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))))$

$= [\![\varepsilon + a; \mu x.(\varepsilon + a ; x)]\!]$ -- by tr6

$\Rightarrow \{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))))$

$= \{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![a; \mu x.(\varepsilon + a ; x)]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))))$

$= \{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes [\![\mu x.(\varepsilon + a ; x))]\!])$ -- by ts1

$\Rightarrow \{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))))$

$= \{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))$

-- by tr4

In a similar way as in 6.2.5.1, this axiom is proved soundness by mapping both expressions to the trace semantics by tracing the repetitions. The left-side expression has to be unrolled by using the unrolling while-loop rule for the semantics and, subsequently, both expressions are translated to the same trace semantics.

7.2.5 Encapsulation

As is mentioned in chapter 5, encapsulation is used to group a set of tasks and structures in the task algebra. It is supported by three axioms: vacuous subtask (e.1), coincident exit (e.2), and vacuous selection (e.3).

7.2.5.1 Soundness for the vacuous subtask axiom

$[\![\{\sigma\}_T]\!] = [\![\varepsilon]\!] = [\![\{\varepsilon\}_T]\!]$

$\Rightarrow$ unpack($[\![\sigma]\!]$) = $[\![\varepsilon]\!]$ = unpack($[\![\varepsilon]\!]$) -- by tu1

$\Rightarrow$ unpack($\{<\sigma>\}$) = $[\![\varepsilon]\!]$ = unpack($[\![\varepsilon]\!]$) -- by tb2

$\Rightarrow$ unpack($\{<\sigma>\}$) = $[\![\varepsilon]\!]$ = unpack($\{<>\}$) -- by tb1

$\Rightarrow \{<>\} = [\![\varepsilon]\!] = \{<>\}$ -- by up1

$\Rightarrow \{<>\} = \{<>\} = \{<>\}$ -- by tb1

Soundness for the vacuous subtask axiom is proved by tracing the unpacking of *Activity*, which, in the case of a subtask containing *succeed*, is transformed to the empty trace by the unpacking function. The second and third expressions are derived

from the empty sequence to the empty trace, where in the case of the third expression, this is passed to the unpack function and the empty trace prevails at the end.

7.2.5.2 Soundness for the coincident exit axiom

$\forall a \in Activity \bullet [\![\{a;\sigma\}_T]\!] = [\![\{a\}_T]\!]$

$\Rightarrow$ unpack($[\![a; \sigma]\!]$) = unpack($[\![a]\!]$) -- by tu1

$\Rightarrow$ unpack($[\![a]\!] \otimes [\![\sigma]\!]$) = unpack($[\![a]\!]$) -- by ts1

$\Rightarrow$ unpack($[\![a]\!] \otimes \{<\sigma>\}$) = unpack($[\![a]\!]$) -- by tb2

$\Rightarrow$ unpack($\{t_1, t_2, \ldots, t_n\} \otimes \{<\sigma>\}$) = unpack($[\![a]\!]$)

Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$ in unpack($[\![a]\!] \otimes \{<\sigma>\}$)

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow [\![a]\!] = \text{unpack}([\![a]\!])$ $\bigcup_{i=1}^{n} \{lift\ (t_i \# <\sigma>)\}$

$\Rightarrow [\![a]\!] = \text{unpack}([\![\{t_1, t_2, \ldots, t_n\}]\!])$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$ in unpack($[\![a]\!]$})

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow [\![a]\!] = [\![a]\!]$ $\bigcup_{i=1}^{n} \{lift\ t_i\}$

The axiom for the coincident exit considers the case where the succeed symbol is next to the right boundary of a subtask. This, as is proved above, is equivalent to having the same subtask without the succeed symbol. The derivation details formally the operation of the unpacking rule for task sets. For this derivation, *succeed* is disallowed to be in the *Activity a* in order to avoid the examination of every $t_1, t_2, \ldots, t_n$. If *succeed* could be in *a*, the activity should be resolved before the function *lift* eliminates fail, and the result should be a subset of *a*.

7.2.5.3 Soundness for the vacuous selection axiom

$\forall a \in Activity \bullet [\![\{a + \sigma\}_T]\!] = [\![\{a\}_T + \varepsilon]\!]$

$\Rightarrow$ unpack ($[\![a + \sigma]\!]$) = $[\![\{a\}_T + \varepsilon]\!]$ -- by tu1

$\Rightarrow$ unpack ($\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])$) = $[\![\{a\}_T + \varepsilon]\!]$ -- by ta2

$\Rightarrow$ unpack ($\{<\downarrow>\} \otimes ([\![a]\!] \cup \{<\sigma>\})$) = $[\![\{a\}_T + \varepsilon]\!]$ -- by tb2

$\Rightarrow$ unpack ($\{<\downarrow>\} \otimes ([\![a]\!] \cup \{<\sigma>\})$)

= $\{<\downarrow>\} \otimes ([\![\{a\}_T]\!] \cup [\![\varepsilon]\!])$ -- by ta2

$\Rightarrow$ unpack ({<↓>} ⊗ ([[a]] ∪ {<σ>}))

= {<↓>} ⊗ (unpack([[a]]) ∪ [[ε]]) -- by tu1

$\Rightarrow$ unpack ({<↓>} ⊗ ([[a]] ∪ {<σ>}))

= {<↓>} ⊗ (unpack([[*a*]]) ∪ {<>}) -- by tb1

$\Rightarrow$ unpack ({<↓>} ⊗ [[a]] ∪ {<↓>} ⊗ {<σ>})

= {<↓>} ⊗ (unpack([[a]]) ∪ {<>}) -- by A.2

$\Rightarrow$ unpack ({<↓>} ⊗ [[a]] ∪ {<↓, σ>})

= {<↓>} ⊗ (unpack([[a]]) ∪ {<>}) -- cp1

$\Rightarrow$ unpack ({<↓>} ⊗ [[a]]) ∪ unpack({<↓, σ>})

= {<↓>} ⊗ (unpack([[a]]) ∪ {<>}) -- by A.8

$\Rightarrow$ unpack ({<↓>} ⊗ {t_1, t_2, …, t_n}) ∪ unpack({<↓, σ>})

= {<↓>} ⊗ (unpack({t_1, t_2, …, t_n}) ∪ {<>})

Let [[a]] = {t_1, t_2, …, t_n}

where $t_1 \neq <\sigma>$, $t_2 \neq <\sigma>$, …, $t_n \neq <\sigma>$

$\Rightarrow$ unpack ({<↓>} ⊗ {t_1, t_2, …, t_n}) ∪ {<↓>}

= {<↓>} ⊗ ({t_1, t_2, …, t_n} ∪ {<>}) -- by up1

$\Rightarrow$ unpack ({<↓>} ⊗ {t_1, t_2, …, t_n}) ∪ {<↓>}

= {<↓>} ⊗ {t_1, t_2, …, t_n} ∪ {<↓>} ⊗{<>} -- by A.2

$\Rightarrow$ unpack ({<↓t_1>, <↓t_2>, …, <↓t_n>}) ∪ {<↓>}

= {<↓t_1>, <↓t_2>, …, <↓t_n>} ∪ {<↓>} -- by cp1

$\Rightarrow$ {<↓t_1>, <↓t_2>, …, <↓t_n>}) ∪ {<↓>}

= {<↓t_1>, <↓t_2>, …, <↓t_n>} ∪ {<↓>} $\bigcup_{i=1}^{n}$ {*lift* <↓ t_i > }

For this proof, it is necessary also to use some of the basic properties assumed at the start of this chapter, which are proved in Appendix A; namely the distribution of unpack over union and distribution of unpack over the concatenated product. The rest

of the derivation depends on the semantics defined in chapter 5 and the distribution of the concatenated product over union.

7.3 Congruence

Congruence is a property showing an equivalence relation of the algebra. This property can be proved directly with the axioms of the algebra. Nevertheless, congruence in an algebra can also be checked by taking equivalent expressions and adding a subexpression to each of the equivalences. The semantics has to be equal for the equivalent expression if the expression is congruent. However, this approach has the disadvantage that formally, the proof depends of the proof of completeness for the algebra with respect to the semantics. As mentioned in chapter 6, the property of completeness is considered beyond the scope of this work and specified as future work. Even so, in this section some examples of congruence properties are depicted. The complete listing of congruence properties can be seen in Appendix B.

7.3.1 Showing congruence for basic operators in the associative sequence axiom

In this section, the congruence for the associative sequence axiom (s.1) is demonstrated for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

7.3.1.1 Congruence in s.1 with the sequence operator

If $\forall a,b,c \in Activity \bullet$ ⟦a; (b; c)⟧ ≡ ⟦ (a; b); c ⟧, **then**

$\forall a,b,c,d \in Activity \bullet$ ⟦(a; (b; c)); d⟧ ≡ ⟦ ((a; b); c); d⟧

⇒⟦a; (b; c)⟧ ⊗ ⟦d⟧ ≡ ⟦(a; b); c⟧ ⊗ ⟦d⟧ -- by ts1

⇒⟦a⟧ ⊗ ⟦(b; c)⟧ ⊗ ⟦d⟧ ≡ ⟦(a; b)⟧ ⊗ ⟦c⟧ ⊗ ⟦d⟧ -- by ts1

⇒⟦a⟧ ⊗ ⟦b⟧ ⊗ ⟦c⟧ ⊗ ⟦d⟧ ≡ ⟦a⟧ ⊗ ⟦b⟧ ⊗ ⟦c⟧ ⊗ ⟦d⟧ -- by ts1

7.3.1.2 Congruence in s.1 with the selection operator

If $\forall a,b,c \in Activity \bullet$ ⟦a; (b; c)⟧ ≡ ⟦ (a; b); c ⟧, **then**

$\forall a,b,c,d \in Activity \bullet$ ⟦(a; (b; c)) + d⟧ ≡ ⟦ ((a; b); c) + d⟧

⇒ {<↓>} ⊗ (⟦(a; (b; c))⟧ ∪ ⟦d⟧) ≡ {<↓>} ⊗ (⟦((a; b); c)⟧ ∪ ⟦d⟧) -- by ta2

⇒ {<↓>} ⊗ ((⟦a⟧ ⊗ ⟦(b; c)⟧) ∪ ⟦d⟧)

≡ {<↓>} ⊗ ((⟦(a; b)⟧ ⊗ ⟦c⟧) ∪ ⟦d⟧) -- by ts1

⇒ {<↓>} ⊗ ((⟦a⟧ ⊗ ⟦b⟧ ⊗ ⟦c⟧) ∪ ⟦d⟧)

≡ {<↓>} ⊗ (([[a]] ⊗ [[b]] ⊗ [[c]]) ∪ [[d]]) -- by ts1

7.3.1.3 Congruence in s.1 with the parallel composition operator

If $\forall a,b,c \in Activity \bullet$ [[a; (b; c)]] ≡ [[(a; b); c]], **then**

$\forall a,b,c,d \in Activity \bullet$ [[(a; (b; c)) || d]] ≡ [[((a; b); c) || d]]

⇒ ([[(a; (b; c))]] // [[d]]) ≡ ([[((a; b); c)]] // [[d]]) -- by tp1

⇒ (([[a]] ⊗ [[(b; c)]]) // [[d]]) ≡ (([[(a; b)]] ⊗ [[c]]) // [[d]]) -- by ts1

⇒ (([[a]] ⊗ [[b]] ⊗ [[c]]) // [[d]]) ≡ (([[a]] ⊗ [[b]] ⊗ [[c]]) // [[d]]) -- by ts1

7.3.1.4 Congruence in s.1 with the until-loop

If $\forall a,b,c \in Activity \bullet$ [[a; (b; c)]] ≡ [[(a; b); c]], **then**

$\forall a,b,c \in Activity \bullet$ [[μx.((a; (b; c)); ε + x)]] ≡ [[μx.(((a; b); c); ε + x)]]

⇒ μt.([[a; (b; c)]] ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.([[(a; b); c]] ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by tr2

⇒ μt.([[a]] ⊗ [[(b; c)]] ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.([[(a; b)]] ⊗ [[c]] ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by ts1

⇒ μt.([[a]] ⊗ [[b]] ⊗ [[c]] ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.([[a]] ⊗ [[b]] ⊗ [[c]] ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by ts1

7.3.1.5 Congruence in s.1 with the while-loop

If $\forall a,b,c \in Activity \bullet$ [[a; (b; c)]] ≡ [[(a; b); c]], **then**

$\forall a,b,c \in Activity \bullet$ [[μx.(ε + (a; (b; c)); x)]] ≡ [[μx.(ε + ((a; b); c); x)]]

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ ([[a; (b; c)]] ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ([[(a; b); c]] ⊗ t))) -- by tr4

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ ([[a]] ⊗ [[(b; c)]] ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ([[(a; b)]] ⊗ [[c]] ⊗ t))) -- by ts1

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ ([[a]] ⊗ [[b]] ⊗ [[c]] ⊗ t)))

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (⟦a⟧ \otimes ⟦b⟧ \otimes ⟦c⟧ \otimes t)))$ -- by ts1

7.3.1.6 Congruence in s.1 with the encapsulation

If $\forall a, b, c \in Activity \bullet ⟦a; (b; c)⟧ \equiv ⟦(a; b); c⟧$, **then**

$\forall a, b, c \in Activity \bullet ⟦\{a; (b; c)\}_T⟧ \equiv ⟦\{(a; b); c\}_T⟧$

$\Rightarrow$unpack($⟦a; (b; c)⟧$) $\equiv$ unpack($⟦(a; b); c⟧$) -- by tu1

$\Rightarrow$ unpack($⟦a⟧ \otimes ⟦(b; c)⟧$) $\equiv$ unpack($⟦(a; b)⟧ \otimes ⟦c⟧$) -- by ts1

$\Rightarrow$ unpack($⟦a⟧ \otimes ⟦b⟧ \otimes ⟦c⟧$) $\equiv$ unpack($⟦a⟧ \otimes ⟦b⟧ \otimes ⟦c⟧$) -- by ts1

7.4 Summary

In this chapter, the trace semantics defined in the previous chapter were used to prove the soundness of the axioms for the task algebra illustrated in chapter 4. Some examples of congruence properties were demonstrated for the algebra. A full listing of congruence properties is depicted in Appendix B.

Chapter 8:
The Task Algebra Implementation

The previous chapters demonstrated the soundness of the task algebra semantics presented in chapter 6. In this chapter, the implementation of the algebra is presented using a case study translating a Task Flow Diagram into the task algebra. The traces generated by the program are then interrogated by LTL and CTL queries to demonstrate how it is possible to model-check temporal logic properties of the described system.

8.1 Introduction

The syntax for the task algebra was presented in chapter 5 followed by its semantics in chapter 6. This chapter presents the implementation of the algebra and some results from applying the algebra to represent task flow diagrams and model-checking temporal logic properties in the trace outputs.

So far, the objectives of this work have been developed in previous chapters. The last objective proposed for this work was to provide an implementation of the algebra. This implementation is necessary to test the feasibility of the formal representation. In addition, the task algebra implementation will be complemented with model-checking extensions (allowing LTL and CTL expressions) in an attempt to show practical uses of this work.

The next section describes the implementation showing the main algorithms implemented. The full code of the programs is shown in Appendix C. Section 8.3 presents an example where a task flow diagram is translated to its corresponding task algebra representation and the trace semantics generated by the program. From the traces, it is possible to obtain useful semantic information about task flows, such as whether two alternative flow diagrams are equivalent, or whether certain properties hold always or eventually. To determine the first requires no more than simple set operations upon trace sets; whereas the latter requires temporal logic expressions, as shown in section 8.4.

8.2 Task algebra implementation

The implementation for the task algebra was developed in the Haskell language, which is a lazy functional language based on lambda calculus [142]. The application in Haskell is a compiler that transforms a task algebra expression and, if the expression is correct, generates the corresponding traces for the expression. The

process will be similar to a one-pass compiler [143]. Figure 8.1 shows the process for a task algebra expression in the implementation to generate the set of traces.

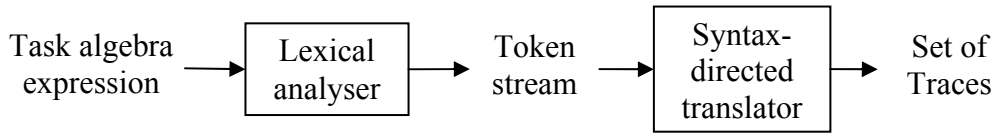

Figure 8.1. Structure of the Task Algebra implementation

From the BNF definition for the task algebra described in chapter 4, there are just a couple of changes that have been made with the aim of facilitating the analysis of the input string representing an expression in the algebra:

```
Activity  ::=  Epsilon                      -- empty activity
          | Sigma                           -- σ succeed
          | Phi                             -- ϕ fail
          | Task                            -- a single task
          | Activity ; Activity             -- a sequence of activity
          | Activity + Activity             -- a selection of activity
          | Activity || Activity            -- parallel activity
          | Mu.x(Activity ; Epsilon + x)    -- until-loop activity
          | Mu.x(Epsilon + Activity ; x)    -- while-loop activity

Task::= Simple                              -- a simple task
          | { Activity }                    -- encapsulated activity
```

Evidently, the Greek symbols used in the algebra had to be converted into machine-readable tokens in the Latin character set. Also, the *Mu* symbol was separated from the variable *x* using a dot to simplify their identification in the lexical analyser (the bound expression is then contained in parentheses). Table 8.1 shows the correspondence between the expression written in the original algebra syntax and the machine-readable syntax for the Haskell application.

Task Algebra	**Task Algebra implementation**
a; ϕ; c	a; Phi; c
a + ε + b	a + Epsilon + b
a || b || σ	a || b || Sigma
μx.(a ; ε + x)	Mu.x(a ; Epsilon + x)
μx.(ε + a ; x)	Mu.x(Epsilon + a ; x)

Table 8.1 Comparison between original Task Algebra syntax and the Haskell implementation

Additionally, the traces for the expression are generated executing the function *tr*. For instance, the execution of *tr "a; Phi; c"* creates the traces for the expression *a;*

Phi; c. Consequently, the expressions depicted above have the following set of traces:

tr “a; Phi; c”	{[a,Phi]}
tr “a + Epsilon + b”	{[!],[!,a],[!,b]}
tr “a || b || Sigma”	{[Sigma]}
tr “Mu.x(a ; Epsilon + x)”	{[a,!],[a,!,a]}
tr “Mu.x(Epsilon + a ; x)”	{[!],[!,a,!],[!,a,!,a]}

As can be seen, traces are produced following the semantics defined in chapter 5 with the exception of the repetition structures. Traces for the until- and while-loops are generated for a finite number of cycles, setting an arbitrary maximum limit of two repetitions for each loop. While- and until-loop show, as expected, different trace sets due to the position of the condition (e.g., the trace [!] is produced in the while-loop as a result of the possibility of doing nothing). Minor differences in the trace notation are the syntax for commit ‘!’ instead of ‘↓’, and the use of square brackets to delimit traces as a substitute for the angle brackets used originally. In addition, simple task names should begin with a lowercase; uppercases are reserved for compound tasks and the algebra keywords.

The implementation takes a string as an input for the expression in the algebra, which is translated to the corresponding functions to generate the resulting trace semantics. The parser was built using the Happy parser generator for Haskell. In addition, a simple hand-written lexical analyser was built. Together, the parser and the lexical analyser are responsible of linking to the appropriate constructor for the *Activity* data type.

```
Model : Activity            { $1 }
      | CompoundTask Model  { Model $1 $2 }

CompoundTask :
      'let' taskName '=' Encapsulation { CompoundTask $2 $4 }

Encapsulation:
      '{' Activity '}'      { Task (Encapsulation $2) }

Activity :
      Activity ';' Activity { Sequence $1 $3 }
      | Activity '+' Activity    { Selection $1 $3 }
      | Activity '||' Activity   { Parallel $1 $3 }
      --  Until-loop
      | 'Mu' '.' simple '(' Activity ';'
      'Epsilon' '+' simple ')'   { UntilLoop $5 (Simple $3)
(Simple $9) }
      -- While-loop
      |'Mu' '.' simple '(' 'Epsilon' '+' Activity';' simple ')'
                 { WhileLoop $7 (Simple $3) (Simple $9) }
      | '(' Activity ')'    { Task (Brackets $2) }
      | Encapsulation       { $1 }
```

```
      | 'Epsilon'           { Epsilon }
      | 'Phi'               { Fail }
      | 'Sigma'             { Succeed }
      | simple              { Task (Simple $1) }
      | taskName            { Task (Compound $1) }
```

The definition of the *Activity* data type is as follows:

```
-- Activity
data Activity
      = Epsilon
      | Fail
      | Succeed
      | Task Task
      | Sequence Activity Activity
      | Selection Activity Activity
      | Parallel Activity Activity
      | UntilLoop Activity Task Task
      | WhileLoop Activity Task Task
      | CompoundTask String Activity
      | Model Activity Activity
   deriving (Eq, Ord)
```

Then, an instance declaration of *Show Activity* is defined for each constructor, where the trace operation is called for most of the constructors together with the consequent data type, allowing by pattern matching to do the aproppiate calls to generate the set of traces. The definition of the function *trace* is as follows:

```
trace :: Activity -> DataDictionary -> SetOfTraces
```

where *SetOfTraces* is declared as a set if the *Trace* type. *Trace* is declared as a list of Event:

```
type Trace       = [Event]

type SetOfTraces = Set Trace
```

Event is a data type defining the trace elements:

```
data Event = Ident String | Phi | Sigma | Commit

  deriving (Eq, Ord)
```

From here, the use of the function *trace*, by pattern matching, calls the appropriate functions implementing the semantics from Chapter 6. For example, for sequence composition the function *trace* is called as follows:

```
trace (Sequence a b) dict
```

which is equal to:

```
trace a dict #* trace b dict
```

meaning that the trace of a sequence of *a* followed by *b* is equal to the trace of *a* concatenated with the trace of *b*, using the concatenated product operation (#*). As defined in Chapter 6, the concatenated product works over set of traces:

```
(#*) :: SetOfTraces -> SetOfTraces -> SetOfTraces
setA #* setB
      | setA == empty  = empty
      | setB == empty = empty
      | otherwise
            = union (insert (findMin setA # findMin setB)
             (singleton (findMin setA) #* (difference setB
(singleton (findMin setB)))))
      ((difference setA (singleton (findMin setA))) #* setB )
```

which uses the concatenation function to append the traces. The semantic function for concatenation of traces implemented in Haskell:

```
(#) :: Trace -> Trace -> Trace
[Sigma] # (item:rest)            = [Sigma] # rest
[Phi] # (item:rest)              = [Phi] # rest
[Commit] # trace@(item:rest)
      | item == Commit           = trace
      | otherwise                = Commit : trace
(item:rest) # trace              = item : (rest # trace)
epsilon#trace                    = trace
```

As mentioned above, the implementation for the rest of the semantic functions can be seen in Appendix C. The next section introduces a case of study to show how this implementation can be used.

8.3 An electronic journal

An interesting case study was developed by Adams [144] working with the Discovery Method for modelling a web based electronic journal. The study models an electronic journal, which is offered free to all subscribers, where the authors submit their articles and pay towards the costs of their online publication by conducting peer reviews of articles submitted by other authors.

There are four actor roles identified in the system. *Reader* is the role denoting someone who wants to browse the journal, read articles or search for information in the journal. The role of *Author* defines someone who wants to publish his/her articles. The *Reviewer* is the role of an author who is required to review other unpublished papers with the aim of paying towards the cost of publishing his/her own paper. The last role is that of the *Editor*, which is the role of the administrator of the system. The editor role is subdivided into a master editor and sub-editors, which can be assigned their role by any master editor. In the study, a Task Structure diagram is developed for each of the four main roles, describing the tasks they individually perform. The diagrams can be seen in chapter 3 in [144].

8.3.1 Task Flow analysis

This section focuses on the Task Flow analysis, which is the part of the Discovery Method where Task Flow Diagrams are constructed in order to determine the

workflows linking the identified tasks. For every *Task Structure Diagram* in the case study, there is a corresponding *Task Flow Diagram*, illustrating the order in which the tasks are carried out for each role. In general, Task Flow diagrams are constructed from the viewpoint of the principal users of a system.

Figure 8.2 shows the *Task Flow Diagram* for the reader role. The diagram describes the choice the reader has initially to decide between reading information about the journal, searching for an article, or reading about content alerting before subscribing to the content alerting service.

The diagram is formed by six tasks: *Read Info on Journal*, *Search for Article*, *Read Abstract*, *Download Article*, *Read about Content Alerting*, and *Register for Content Alerting*. The first task is clearly defined as a compound task, which is formed by the subtasks *Read Journal Aims*, and *Read Submission Instructions*.

The task algebra expression for the diagram from Figure 8.2 should be as follows:

Mu.x(ReadInfoOnJournal; Epsilon + x)
+Mu.x((searchForArticle; Phi + readAbstract; downloadArticle + Epsilon); Epsilon + x)
+(readAboutContentAlerting; Epsilon + registerForContentAlerting)

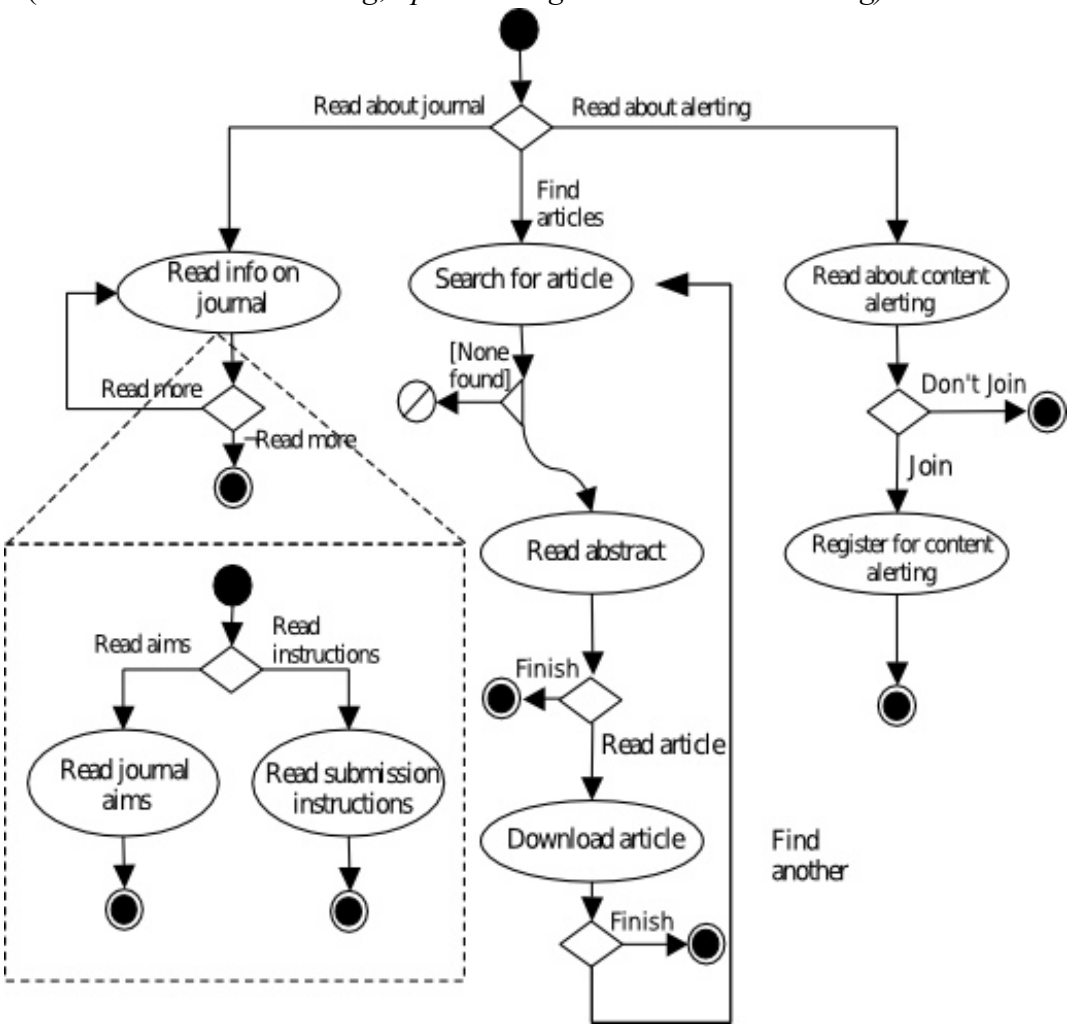

Figure 8.2. Reader Task Flow Diagram

Additionally, the compound task *ReadInfoOnJournal* can be defined like this:

let ReadInfoOnJournal = {readJournalAims + readSubmissionInstructions}

In the trace semantics only simple tasks are represented. The compound task *ReadInfoOnJournal* is unpacked and its subtasks promoted to the higher level as defined by the semantics in chapter 5. After the task algebra expression is defined, it may be processed by the *tr* function to generate the set of traces. For this case, the set of traces is:

```
{ [!,readAboutContentAlerting,!],
[!,readAboutContentAlerting,!, registerForContentAlerting],
[!,readJournalAims,!],
[!,readJournalAims,!,readJournalAims],
[!,readJournalAims,!,readSubmisiionInstructions],
[!,readSubmisiionInstructions,!],
[!,readSubmisiionInstructions,!,readJournalAims],
[!,readSubmisiionInstructions,!,readSubmisiionInstructions],
[!,searchForArticle,!,readAbstract,!],
[!,searchForArticle,!,readAbstract,!,downloadArticle,!],
[!,searchForArticle,!,readAbstract,!,downloadArticle,!,
searchForArticle,!,readAbstract,!],
[!,searchForArticle,!,readAbstract,!,downloadArticle,!,
searchForArticle,!,readAbstract,!,downloadArticle],
[!,searchForArticle,!,readAbstract,!,downloadArticle,!,
searchForArticle,!,Phi],
[!,searchForArticle,!,readAbstract,!,searchForArticle,!,
readAbstract,!],
[!,searchForArticle,!,readAbstract,!,searchForArticle,!,
readAbstract,!,downloadArticle],
[!,searchForArticle,!,readAbstract,!,searchForArticle,!,Phi],
[!,searchForArticle,!,Phi] }
```

8.3.1.1 Author Task Flow Diagram

The role of author is used for someone who wants to publish his/her articles. It involves the options of *Read Instructions*, *Obtain Style*, *Complete Restricted Task* (such as *Read Reviews* or *Check Article Status*), and *Submit Article*. Figure 8.3 shows the *Task Flow Diagram* for the author role. All tasks in the diagram are simple tasks with the exception of *Login,* which is defined later.

The Task Algebra expression for the Author diagram is represented as follows:

(readAuthorGuidelines;readReviewerGuidelines) + viewStyleGuide
+(Mu.x(Login; Epsilon + x);(readReviews;obtainEditorsDecision;
submitReworkedArticle + Epsilon) + checkArticleStatus)
+(completeSubmissionEform;obtainReviewerID)

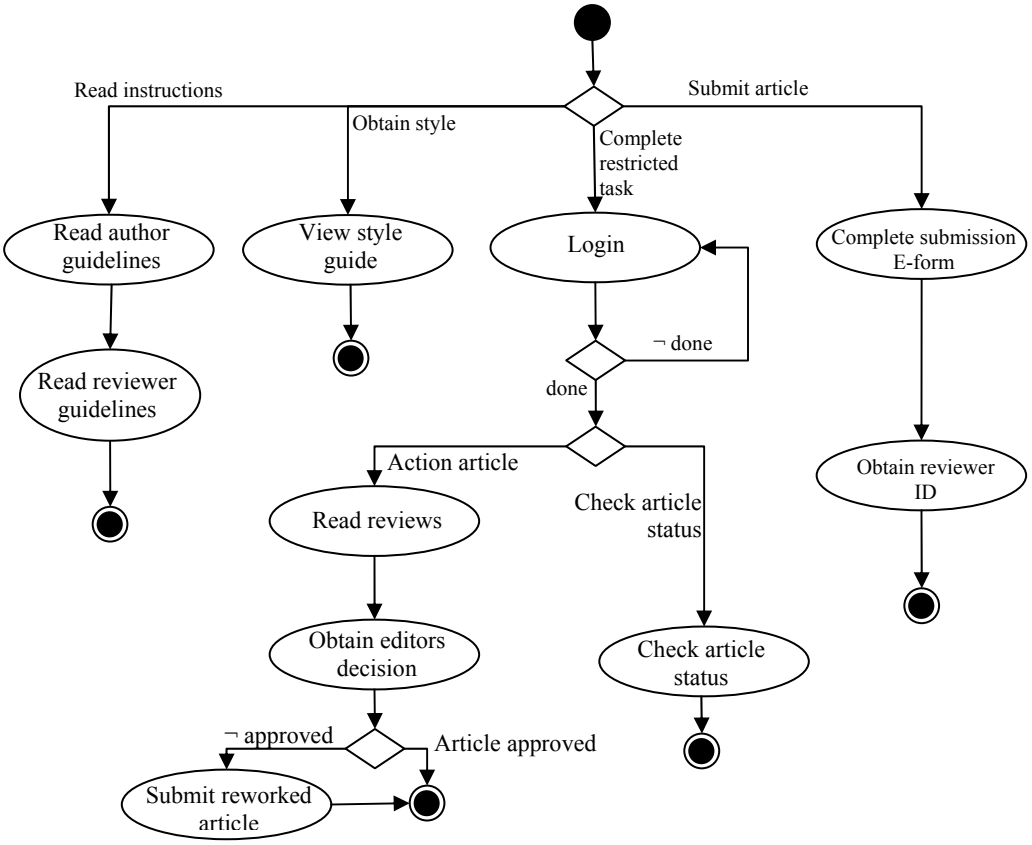

Figure 8.3. Author Task Flow Diagram

The compound task *Login* contemplates the complete process for login into the system, including the case when the user fails to introduce correctly the password, with the possibility to activate a password reminder. Figure 8.4 presents the Task Flow diagram for this task. The resultant expression in the task algebra is:

let Login = { (*Phi* + *Epsilon* +(*requestPassword*; *Epsilon* + *Phi*));
enterPassword }

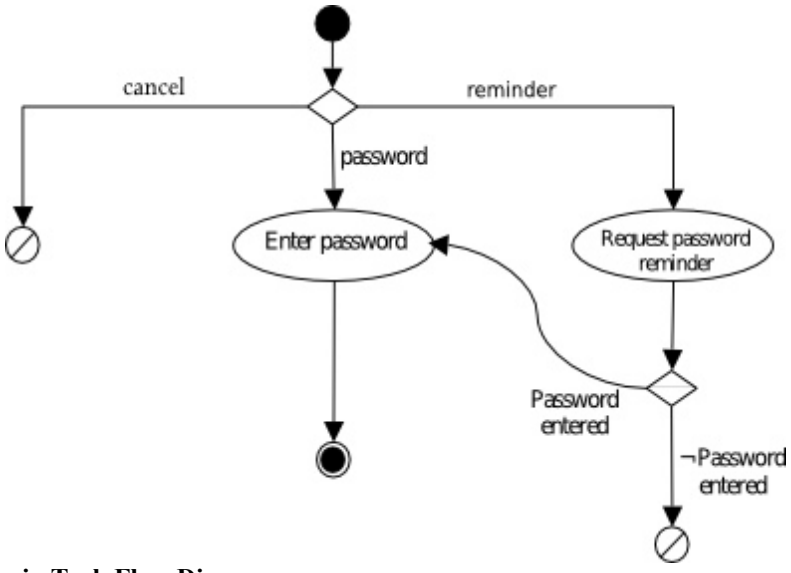

Figure 8.4. Login Task Flow Diagram

The set of traces resulting from the task algebra expression includes the task *Login* which, as was mentioned above, manages the success and failure cases of logging into the system by entering the password. Because *Login* is in a cycle to allow multiple opportunities to gain entry into the system, an until-loop structure *Mu.x(Login; Epsilon + x)* is needed. The set of traces from *Login* is unpacked within the set of traces in the general expression to generate the complete set of traces:

```
{[!,completeSubmissionEform,obtainReviewerID],
[!,enterPassword,!,checkArticleStatus],
[!,enterPassword,!,enterPassword,!,checkArticleStatus],
[!,enterPassword,!,enterPassword,!,readReviews,obtainEditorsDecision,
!],[!,enterPassword,!,enterPassword,!,
readReviews,obtainEditorsDecision,!,submitReworkedArticle],
[!,enterPassword,!,readReviews,obtainEditorsDecision,!],
[!,enterPassword,!,readReviews,obtainEditorsDecision,!,
submitReworkedArticle],
[!,enterPassword,!,requestPassword,!,enterPassword,!,
checkArticleStatus],
[!,enterPassword,!,requestPassword,!,enterPassword,!,
readReviews,obtainEditorsDecision,!],
[!,enterPassword,!,requestPassword,!,enterPassword,!,
readReviews,obtainEditorsDecision,!,submitReworkedArticle],
[!,enterPassword,!,requestPassword,!,Phi], [!,enterPassword,!,Phi],
[!,readAuthorGuidelines,readReviewerGuidelines],
[!,requestPassword,!,enterPassword,!,checkArticleStatus],
[!,requestPassword,!,enterPassword,!,enterPassword,!,
checkArticleStatus],
[!,requestPassword,!,enterPassword,!,enterPassword,!,
readReviews,obtainEditorsDecision,!],
[!,requestPassword,!,enterPassword,!,enterPassword,!,
readReviews,obtainEditorsDecision,!,submitReworkedArticle],
[!,requestPassword,!,enterPassword,!,
readReviews,obtainEditorsDecision,!],
[!,requestPassword,!,enterPassword,!,
readReviews,obtainEditorsDecision,!,submitReworkedArticle],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,checkArticleStatus],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,readReviews,obtainEditorsDecision,!],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,readReviews,obtainEditorsDecision,!,submitReworkedArticle],
[!,requestPassword,!,enterPassword,!,requestPassword,!,Phi],
[!,requestPassword,!,enterPassword,!,Phi], [!,requestPassword,!,Phi],
[!,viewStyleGuide], [!,Phi]}
```

8.3.1.2 Reviewer Task Flow Diagram

The *reviewer* role defines the behaviour in the system for a user who wants to write a review of an article or perform some related activity, such as read an abstract in order to choose a paper, check his/her payment status (authors "pay" by doing reviews), or simply checking the guidelines for the reviewers. Figure 8.5 presents the *Task Flow Diagram* for this role where, as for the previous role, *Login* is the only compound task in this diagram. The flow for *Login* is the same defined earlier in Figure 8.4.

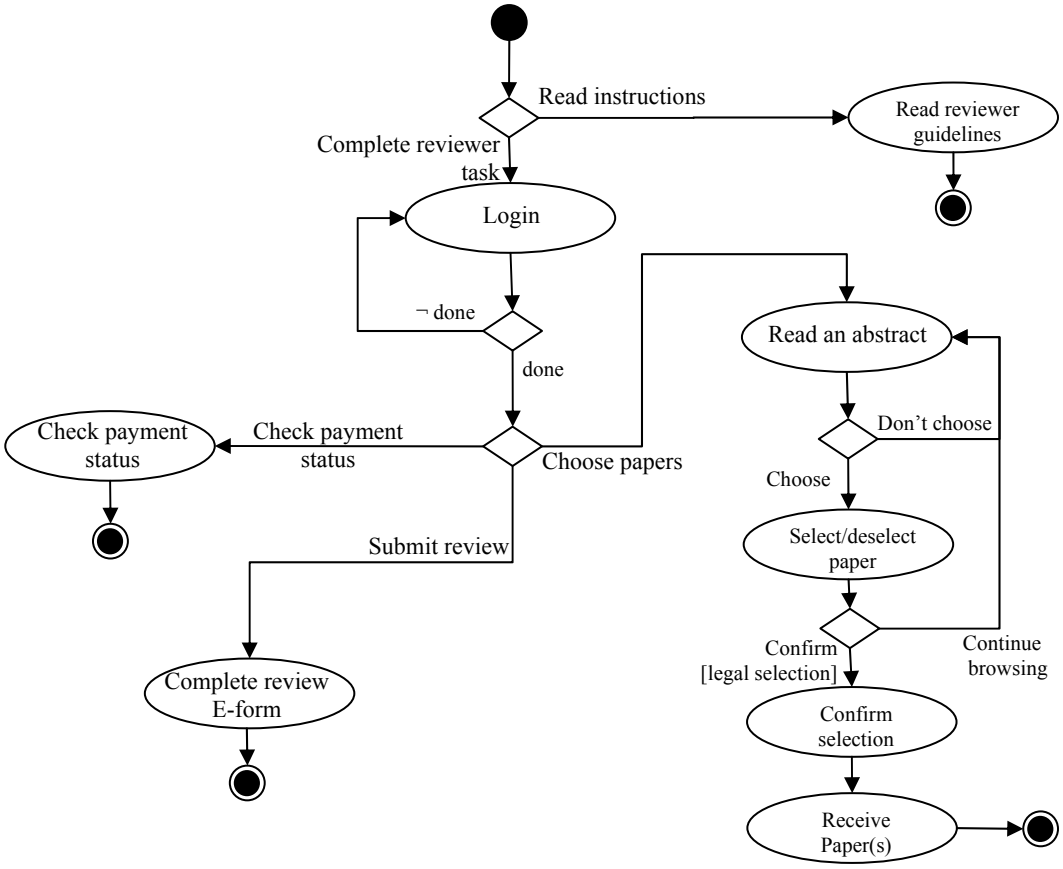

Figure 8.5. Reviewer Task Flow Diagram

In a similar manner to the section above, the content of the *Reviewer Task Flow Diagram* may be expressed directly in the syntax of the task algebra, incorporating as unitary wholes any tasks that encapsulate further flows, such as the *Login* task:

readReviewerGuidelines + (Mu.x(Login; Epsilon + x); checkPaymentStatus
+completeReviewForm + (Mu.x((Mu.y(readAnAbstract; Epsilon + y);
selectPaper); Epsilon + x); confirmSelection; receivePapers))

From applying the trace function to the task algebra expression above, the following set of traces is obtained, in which once again the behaviour of the *Login* task is unpacked:

```
{[!,enterPassword,!,checkPaymentStatus],
[!,enterPassword,!,completeReviewEform],
[!,enterPassword,!,enterPassword,!,checkPaymentStatus],
[!,enterPassword,!,enterPassword,!,completeReviewEform],
[!,enterPassword,!,enterPassword,!,readAnAbstract,!,readAnAbstract,
selectPaper,!,confirmSelection,receivePapers],
[!,enterPassword,!,enterPassword,!,readAnAbstract,!,readAnAbstract,
selectPaper,!,readAnAbstract,!,readAnAbstract,selectPaper,
confirmSelection,receivePapers],
[!,enterPassword,!,enterPassword,!,readAnAbstract,!,readAnAbstract,
selectPaper,!,readAnAbstract,!,selectPaper,confirmSelection,
receivePapers],[!,enterPassword,!,enterPassword,!,readAnAbstract,!,
selectPaper,!,confirmSelection,receivePapers],
```

```
[!,enterPassword,!,enterPassword,!,readAnAbstract,!,selectPaper,!,
readAnAbstract,!,readAnAbstract,selectPaper,confirmSelection,
receivePapers],[!,enterPassword,!,enterPassword,!,readAnAbstract,!,
selectPaper,!,readAnAbstract,!,selectPaper,confirmSelection,
receivePapers],
[!,enterPassword,!,readAnAbstract,!,readAnAbstract,selectPaper,!,
confirmSelection,receivePapers],
[!,enterPassword,!,readAnAbstract,!,readAnAbstract,selectPaper,!,
readAnAbstract,!,readAnAbstract,selectPaper,confirmSelection,
receivePapers],
[!,enterPassword,!,readAnAbstract,!,readAnAbstract,selectPaper,!,
readAnAbstract,!,selectPaper,confirmSelection,receivePapers],
[!,enterPassword,!,readAnAbstract,!,selectPaper,!,confirmSelection,
receivePapers],
[!,enterPassword,!,readAnAbstract,!,selectPaper,!,readAnAbstract,!,
readAnAbstract,selectPaper,confirmSelection,receivePapers],
[!,enterPassword,!,readAnAbstract,!,selectPaper,!,readAnAbstract,!,
selectPaper,confirmSelection,receivePapers],
[!,enterPassword,!,requestPassword,!,enterPassword,!,
checkPaymentStatus],
[!,enterPassword,!,requestPassword,!,enterPassword,!,
completeReviewEform],
[!,enterPassword,!,requestPassword,!,enterPassword,!,readAnAbstract,!
,readAnAbstract,selectPaper,!,confirmSelection,receivePapers],
[!,enterPassword,!,requestPassword,!,enterPassword,!,readAnAbstract,!
,readAnAbstract,selectPaper,!,readAnAbstract,!,readAnAbstract,
selectPaper,confirmSelection,receivePapers],
[!,enterPassword,!,requestPassword,!,enterPassword,!,readAnAbstract,!
,readAnAbstract,selectPaper,!,readAnAbstract,!,selectPaper,
confirmSelection,receivePapers],
[!,enterPassword,!,requestPassword,!,enterPassword,!,readAnAbstract,!
,selectPaper,!,confirmSelection,receivePapers],
[!,enterPassword,!,requestPassword,!,enterPassword,!,readAnAbstract,!
,selectPaper,!,readAnAbstract,!,readAnAbstract,selectPaper,
confirmSelection,receivePapers],
[!,enterPassword,!,requestPassword,!,enterPassword,!,readAnAbstract,!
,selectPaper,!,readAnAbstract,!,selectPaper,confirmSelection,
receivePapers], [!,enterPassword,!,requestPassword,!,Phi],
[!,enterPassword,!,Phi], [!,readReviewerGuidelines],
[!,requestPassword,!,enterPassword,!,checkPaymentStatus],
[!,requestPassword,!,enterPassword,!,completeReviewEform],
[!,requestPassword,!,enterPassword,!,enterPassword,!,
checkPaymentStatus],
[!,requestPassword,!,enterPassword,!,enterPassword,!,
completeReviewEform],
[!,requestPassword,!,enterPassword,!,enterPassword,!,readAnAbstract,!
,readAnAbstract,selectPaper,!,confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,enterPassword,!,readAnAbstract,!
,readAnAbstract,selectPaper,!,readAnAbstract,!,readAnAbstract,
selectPaper,confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,enterPassword,!,readAnAbstract,!
,readAnAbstract,selectPaper,!,readAnAbstract,!,selectPaper,
confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,enterPassword,!,readAnAbstract,!
,selectPaper,!,confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,enterPassword,!,readAnAbstract,!
,selectPaper,!,readAnAbstract,!,readAnAbstract,selectPaper,
confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,enterPassword,!,readAnAbstract,!
,selectPaper,!,readAnAbstract,!,selectPaper,confirmSelection,
receivePapers],
```

```
[!,requestPassword,!,enterPassword,!,readAnAbstract,!,readAnAbstract,
selectPaper,!,confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,readAnAbstract,!,readAnAbstract,
selectPaper,!,readAnAbstract,!,readAnAbstract,selectPaper,
confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,readAnAbstract,!,readAnAbstract,
selectPaper,!,readAnAbstract,!,selectPaper,confirmSelection,
receivePapers],
[!,requestPassword,!,enterPassword,!,readAnAbstract,!,selectPaper,!,
confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,readAnAbstract,!,selectPaper,!,
readAnAbstract,!,readAnAbstract,selectPaper,confirmSelection,
receivePapers],
[!,requestPassword,!,enterPassword,!,readAnAbstract,!,selectPaper,!,
readAnAbstract,!,selectPaper,confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,checkPaymentStatus],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,completeReviewEform],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,readAnAbstract,!,readAnAbstract,selectPaper,!,confirmSelection,
receivePapers],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,readAnAbstract,!,readAnAbstract,selectPaper,!,readAnAbstract,!,
readAnAbstract,selectPaper,confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,readAnAbstract,!,readAnAbstract,selectPaper,!,readAnAbstract,!,
selectPaper,confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,readAnAbstract,!,selectPaper,!,confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,readAnAbstract,!,selectPaper,!,readAnAbstract,!,readAnAbstract,
selectPaper,confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,readAnAbstract,!,selectPaper,!,readAnAbstract,!,selectPaper,
confirmSelection,receivePapers],
[!,requestPassword,!,enterPassword,!,requestPassword,!,Phi],
[!,requestPassword,!,enterPassword,!,Phi], [!,requestPassword,!,Phi],
[!,Phi]}
```

8.3.1.3 Editor Task Flow Diagram

The *Editor* role behaviour is specified in Figure 8.6. As can be seen, an editor is able to evaluate articles and reviews, publish a new edition of the journal, and even to assign sub-editor privileges. The *Task Flow Diagram* shows the different tasks involved for the execution of this role and, like the other roles, but with the exception of the reader role, the compound task of *Login* is required. The rest of the tasks used in this diagram are considered simple tasks.

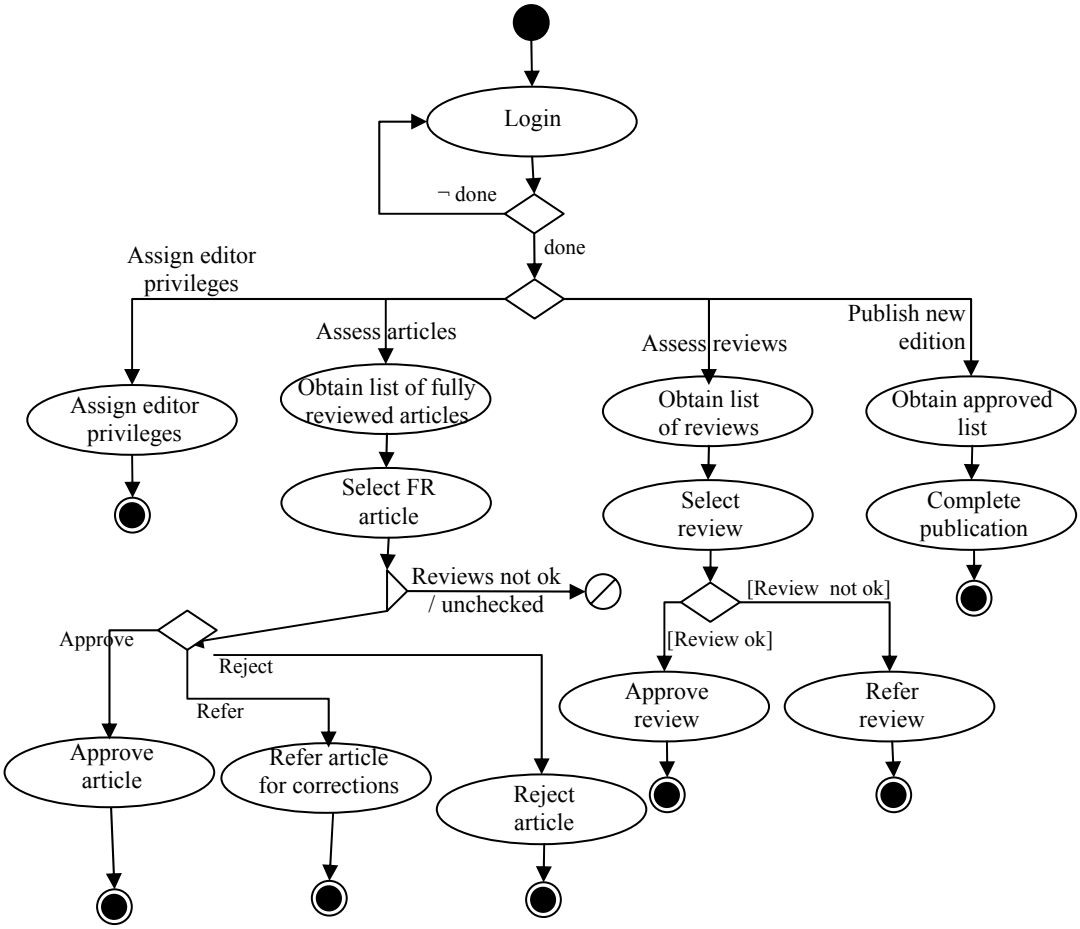

Figure 8.6. Editor Task Flow Diagram

The expression in the Task Algebra includes the until-loop for the verification of the login before carrying out the remaining tasks. After the editor has logged in, s/he has to choose which of the activities want to perform. The task algebra expression is presented here:

Mu.x(Login; Epsilon + x);(assignEditorPrivileges
+(obtainListFRArticles;selectFRArticle;
Phi + (approveArticle + referArtForCorrections + rejectArticle))
+(obtainListReviews;selectReview;approveReview + referReview)
+(obtainApprovedList;completePublication)

The many different executions of this Task Algebra expression may be obtained by applying the *tr* tracing function, which obtains the following traces:

```
{[!,enterPassword,!,assignEditorPrivileges],
[!,enterPassword,!,enterPassword,!,assignEditorPrivileges],
[!,enterPassword,!,enterPassword,!,obtainApprovedList,
completePublication],
[!,enterPassword,!,enterPassword,!,obtainListFRArticles,
selectFRArticle,!,approveArticle],
[!,enterPassword,!,enterPassword,!,obtainListFRArticles,
selectFRArticle,!,referArtForCorrections],
```

```
[!,enterPassword,!,enterPassword,!,obtainListFRArticles,
selectFRArticle,!,rejectArticle],
[!,enterPassword,!,enterPassword,!,obtainListFRArticles,
selectFRArticle,!,Phi],
[!,enterPassword,!,enterPassword,!,obtainListReviews,selectReview,!,
approveReview],
[!,enterPassword,!,enterPassword,!,obtainListReviews,selectReview,!,
referReview],
[!,enterPassword,!,obtainApprovedList,completePublication],
[!,enterPassword,!,obtainListFRArticles,selectFRArticle,!,
approveArticle],
[!,enterPassword,!,obtainListFRArticles,selectFRArticle,!,
referArtForCorrections],
[!,enterPassword,!,obtainListFRArticles,selectFRArticle,!,
rejectArticle],
[!,enterPassword,!,obtainListFRArticles,selectFRArticle,!,Phi],
[!,enterPassword,!,obtainListReviews,selectReview,!,approveReview],
[!,enterPassword,!,obtainListReviews,selectReview,!,referReview],
[!,enterPassword,!,requestPassword,!,enterPassword,!,
assignEditorPrivileges],
[!,enterPassword,!,requestPassword,!,enterPassword,!,
obtainApprovedList,completePublication],
[!,enterPassword,!,requestPassword,!,enterPassword,!,
obtainListFRArticles,selectFRArticle,!,approveArticle],
[!,enterPassword,!,requestPassword,!,enterPassword,!,
obtainListFRArticles,selectFRArticle,!,referArtForCorrections],
[!,enterPassword,!,requestPassword,!,enterPassword,!,
obtainListFRArticles,selectFRArticle,!,rejectArticle],
[!,enterPassword,!,requestPassword,!,enterPassword,!,
obtainListFRArticles,selectFRArticle,!,Phi],
[!,enterPassword,!,requestPassword,!,enterPassword,!,
obtainListReviews,selectReview,!,approveReview],
[!,enterPassword,!,requestPassword,!,enterPassword,!,
obtainListReviews,selectReview,!,referReview],
[!,enterPassword,!,requestPassword,!,Phi], [!,enterPassword,!,Phi],
[!,requestPassword,!,enterPassword,!,assignEditorPrivileges],
[!,requestPassword,!,enterPassword,!,enterPassword,!,
assignEditorPrivileges],
[!,requestPassword,!,enterPassword,!,enterPassword,!,
obtainApprovedList,completePublication],
[!,requestPassword,!,enterPassword,!,enterPassword,!,
obtainListFRArticles,selectFRArticle,!,approveArticle],
[!,requestPassword,!,enterPassword,!,enterPassword,!,
obtainListFRArticles,selectFRArticle,!,referArtForCorrections],
[!,requestPassword,!,enterPassword,!,enterPassword,!,
obtainListFRArticles,selectFRArticle,!,rejectArticle],
[!,requestPassword,!,enterPassword,!,enterPassword,!,
obtainListFRArticles,selectFRArticle,!,Phi],
[!,requestPassword,!,enterPassword,!,enterPassword,!,
obtainListReviews,selectReview,!,approveReview],
[!,requestPassword,!,enterPassword,!,enterPassword,!,
obtainListReviews,selectReview,!,referReview],
[!,requestPassword,!,enterPassword,!,obtainApprovedList,
completePublication],
[!,requestPassword,!,enterPassword,!,obtainListFRArticles,
selectFRArticle,!,approveArticle],
[!,requestPassword,!,enterPassword,!,obtainListFRArticles,
selectFRArticle,!,referArtForCorrections],
[!,requestPassword,!,enterPassword,!,obtainListFRArticles,
selectFRArticle,!,rejectArticle],
[!,requestPassword,!,enterPassword,!,obtainListFRArticles,
```

```
selectFRArticle,!,Phi],
[!,requestPassword,!,enterPassword,!,obtainListReviews,selectReview,!
,approveReview],
[!,requestPassword,!,enterPassword,!,obtainListReviews,selectReview,!
,referReview],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,assignEditorPrivileges],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,obtainApprovedList,completePublication],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,obtainListFRArticles,selectFRArticle,!,approveArticle],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,obtainListFRArticles,selectFRArticle,!,referArtForCorrections],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,obtainListFRArticles,selectFRArticle,!,rejectArticle],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,obtainListFRArticles,selectFRArticle,!,Phi],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,obtainListReviews,selectReview,!,approveReview],
[!,requestPassword,!,enterPassword,!,requestPassword,!,enterPassword,
!,obtainListReviews,selectReview,!,referReview],
[!,requestPassword,!,enterPassword,!,requestPassword,!,Phi],
[!,requestPassword,!,enterPassword,!,Phi], [!,requestPassword,!,Phi],
[!,Phi]}
```

These examples show how it is possible to express realistic Task Flow diagrams in the Task Algebra and convert them to traces, illustrating the possible executions of the diagrams.

8.4 Operations on traces

A set of traces is the trace semantic representation for a Task Flow Diagram. The verification of the diagram may be made in different ways. The simplest operations could be performed by set operators but more operations may be applied over the traces using temporal logic. In this work, we offer three different approaches for checking the models represented with the algebra:

- Set operations on traces
- Model-checking with LTL
- Model-checking with CTL

8.4.1 Set operations on traces

Operations over a set of traces can be easily applied. In some cases, for instance, two or more trace sets may be compared to demonstrate equality (or inequality). For example, comparing two expressions in the task algebra such as *a;b+c* and *(a;b)+(a;c)* can be done using the equal operator in Haskell:

```
Main> tr "{a;b+c}" == tr "{(a;b)+(a;c)}"

False
```

Which is the expected result because the trace semantics for the expression *a;b+c* is *{[a,!,b],[a,!,c]}*, while the trace semantics for *(a;b)+(a;c)* is *{[!,a,b],[!,a,c]}*. It is

easy to see that while in the first case the commit symbol occurs after *a*, in the second case the commit symbol is placed in the first place for every trace in the semantics.

In the same way, it is possible to use common set operations (e.g., set membership, subset, union, difference, intersection) to obtain results over two or more sets of traces. For instance, we could ask whether the trace describing a normal login is one of the identified traces for the *Login* task (Figure 8.4):

```
Main> member ([Commit, Ident "enterPassword"]) (tr "let
Login={(Phi+Epsilon+(requestPassword; Epsilon + Phi)); enterPassword}
{Login}")

True
```

The last example uses the standard *member* function from the Haskell Library to search through the results produced by the *tr* function. The next sections will focus in the use of LTL and CTL operators in order to obtain results reasoning temporal aspects on the Task Flow diagrams.

8.4.2 Model-checking with LTL

Temporal logic has being extensively applied with specification and verification of software. A set of traces, obtained from a task algebra expression, may be used to verify some temporal and logical properties within the specification expressed by the diagrams. For this reason, a simple implementation of LTL was built. This LTL implementation works over the trace semantics generated from a Task Algebra expression. Because the trace semantics represent every possible path of the Task Flow diagram expressed in the Task Algebra, it is straightforward to use LTL formulas to quantify universally over all those paths. In this section, some examples using Linear Temporal Logic (LTL) are presented, to illustrate the reasoning capabilities of the LTL module.

LTL is a temporal logic, formed adding temporal operators to the predicate calculus. These operators that can be used to refer to future states with no quantification over paths. Logical operators inherited from prepositional calculus are the usual ones: ($\neg$, $\wedge$, $\vee$, $\rightarrow$) and the syntax of the LTL expression is as follows[6]:

- *Not p*
- *And p q*
- *Or p q*
- *Impl p q*

The modal operators are divided into the unary (next, always or globally, and finally):

- *X p.* *p* holds on the next state.

[6] As explained here, this is just the syntax of the LTL expression, which is applied to a set of traces generated from a Task Algebra expression. An example showing the whole functions is depicted later.

- *G p.* *p* holds globally.
- *F p.* *p* holds in some future state.

And the binary modal operators (until, weak-until, release):

- *U p q.* *q* holds on the current state or *p* is true and then *q*.
- *W p q.* *q* holds on the current state or *p* is true and then *q*, or *p* is true for all the states.
- *R p q.* *q* holds in all the states or until *p* is true.

Where for this project, the operands *p* and *q* are LTL expressions and the basic expression is the propositional denoted by:

- *Pr <task>*

<task> is any valid name for an simple task or special task symbol. *Pr* is a constructor used to identify such tasks.

With temporal logic, system properties such as safety ("the system never reaches a bad state"), liveness ("there is progress in the system"), fairness ("once X occurred, Y will ocurr in n steps") and self-stabilisation ("the system recovers from a failure in a finite number of steps") can be proven [145]. Property specification of functional requirements written, for instance, in LTL can help to find errors in the design of systems.

8.4.3 Model-checking with CTL

While LTL formulas express temporal properties over all undifferentiated paths, Computational Tree Logic (CTL) also considers quantification over sets of paths. CTL is a branching-time logic [146] and theorems in this logic may also be tested against a set of traces obtained from a task algebra expression, in the same way that LTL theorems were tested above. A CTL application was built to test CTL theorems against expressions in the task algebra. In this case, the application has to transform the traces in a tree representation before applying the expression.

CTL is formed by a combination of logical operators, *path* operators and temporal modal operators. Logical operators are the usual ($\neg$, $\wedge$, $\vee$, $\rightarrow$) and are used as follows:

- *Not p*
- *And p q*
- *Or p q*
- *Impl p q*

Path operators are quantifiers *A* expressing "all paths" and *E* expressing "exists at least one path". *Path* operators are used in combination with temporal modal operators. The modal operators are (next, always or globally, finally and until):

- *X p.* *p* holds on the next state.
- *G p.* *p* holds globally.
- *F p.* *p* holds in some future state.
- *U p q.* *q* holds on the current state or *p* is true and then *q*.

Where for this project, the operands *p* and *q* are CTL expressions and the basic expression is the propositional denoted by:

- *Pr <task>*

<task> is any valid name for an simple task or special task symbol. *Pr* is a constructor used to identify such tasks.

In CTL *path* operators have to be used together with modal operators. *A* and *E* cannot be used without modal operators *X*, *G*, *F* and *U* just like path operators cannot be used without a modal operator. Accordingly, LTL-like expressions are not allowed in CTL.

8.4.4 An implementation of model-checking with LTL

The implementation of LTL model-checking uses *Phi*, which is a data type defined to specify the LTL expressions:

```
data Phi
    = Bool Bool
    | Pr String
    | Not Phi
    | And Phi Phi
    | Or Phi Phi
    | Impl Phi Phi
    | X Phi                 -- Next phi
    | G Phi                 -- All future states
    | F Phi                 -- Eventually
    | U Phi Phi             -- Until
    | W Phi Phi             -- Weak-until
    | R Phi Phi             -- Release
     deriving (Eq, Ord, Show)
```

Additionally, the function *trace* presented in section 8.2 is also used. To check a LTL expression against a model in the algebra, the function *check* is defined. The function *check* is declared as follow:

```
check :: String -> Phi -> (Bool, Trace)

check expr phi   = evalAllTraces (toList (tr expr)) phi
```

where *evalAllTraces* evaluates every possible path against the LTL expression. In order to do this, the function *eval* is called for each trace, while *eval* returns *true*:

```
eval :: Trace -> Phi -> Bool
```

where *eval* initiates the evaluation of each trace by calling the appropiate functions by matching the corresponding constructor. If the evaluation is true, *evalAllTraces* goes to the next trace; on the contrary, if the evaluation is false, *evalAlltraces* stops evaluating and returns the Boolean value together with counterexample.

The syntax for using the LTL application is as follows:

```
check <task-algebra-expression> <LTL-expression>
```

where *task-algebra-expression* is a string expressing a well-formed expression in the algebra, and *LTL-expression* is a valid expression in LTL. As explained above, the result is presented as a tuple showing the result (true or false) and, in the case the LTL expression is false, a counterexample. Evidently, a true result would show no counterexample (an empty list is presented).

In the implementation of LTL for the task algebra, it is possible to construct a theorem to test against the task algebra for the diagram in Figure 8.4, asserting that eventually every path leads to a fail. It is easy to see that this expression is false and, when it is executed, the result is accompanied by a counterexample:

```
MyLTL> check "(Phi+Epsilon+(requestPassword; Epsilon + Phi));
enterPassword" (F (Pr "Phi"))

(False,[!,enterPassword])
```

The counterexample shows that there is a case when, after a choice, the password is entered and then the trace finishes without *fail*. It is important to note that, while fail is represented on the semantics as *fail*, there is no corresponding symbol to represent a successfully terminating path.

The same kind of LTL theorem could be tested against the larger diagram from Figure 8.5. This diagram shows no fail at the top level, but the possibility of failure inside the compound task *Login* (see Figure 8.4) is considered, as depicted in the latter example. The execution command is as follows:

```
MyLTL> check "let Login={(Phi+Epsilon+(requestPassword; Epsilon +
Phi)); enterPassword} {readReviewerGuidelines + (Mu.x(Login; Epsilon
+ x); checkPaymentsStatus + completeReviewEForm +
(Mu.x((Mu.y(readAnAbstract; Epsilon + y); selectPaper); Epsilon + x);
confirmSelection; receivePapers))}" (F (Pr "Phi"))

(False,[!,enterPassword,!,checkPaymentsStatus])
```

As expected, the search to prove the LTL theorem finds a counterexample, where no fail is found in at least one path: namely, after a choice, the password is introduced and then the task of checking the payments status is chosen. In the next example, just an extract of the diagram in Figure 8.5 is analysed in order to verify if the task *receivePapers* is eventually specified after *selectPaper*, which it is true because no counterexample could be found:

```
MyLTL> check "{Mu.x((Mu.y(readAnAbstract; Epsilon + y); selectPaper);
Epsilon + x); confirmSelection; receivePapers}"
(F(U (Pr "selectPaper") (F(Pr "receivePapers"))) )
```

```
(True,[])
```

As could be seen, the LTL queries were interpreted as logical statements about the existence of events, or relationships between events found in the traces generated using the semantics proposed here for the task flow diagrams. These and similar kinds of property can be verified globally for the workflows expressed by the diagrams. The limitation with LTL is that the theorem has to hold for every path (or be falsifiable for at least one path). It may be desirable instead to quantify over different sets of paths, allowing a finer-grained kind of model checking.

8.4.5 An implementation of model-checking with CTL

As with the LTL code, the implementation of CTL model-checking defines the data type *Phi* in order to specify the CTL expressions:

```
data Phi    -- Path and State Operators
     -- operands and logical operators
     = Pr String
     | Bool Bool
     | Not Phi
     | And Phi Phi
     | Or Phi Phi
     | Impl Phi Phi
-- A ? - All: ? has to hold on all paths starting from the
current state.
     | AX Phi                    -- Next phi
     | AG Phi                    -- All future states
     | AF Phi                    -- Eventually
     | AU Phi Phi                -- Until
-- E ? - Exists: there exists at least one path starting from
the current state where ? holds.
     | EX Phi                    -- Next phi
     | EG Phi                    -- All future states
     | EF Phi                    -- Eventually
     | EU Phi Phi                -- Until
   deriving (Eq, Ord, Show)
```

As in the LTL implementation, the function *trace* presented in section 8.2 is also used. In addition, to check a CTL expression against a model in the algebra, the function *check* is defined. The function *check* is declared as follow:

```
check :: String -> Phi ->  ([ [Integer] ], Node)

check expr phi = (sort (sat (tree (Set.toList(tr expr))) phi),
tree (Set.toList(tr expr)) )
```

In this implementations, the expression *expr* is passed to *tr* to generate the traces and the result is used by *tree* to build a tree representation based in the data type *Node*:

```
data Node =     Empty

     |  Node ([Integer], Event) (SubTree)
```

```
      deriving (Eq, Ord, Show)

type Empty =[]

type SubTree = [Node]
```

where as can be seen, a node may be an empty node, or a node containing a value (to identify the node, and *Event* which, as was mentioned above, represents the elements of the traces. In addition, a node may have a *Subtree* which, as can be seen, is defined as a lists of nodes.

The result of *check* is a tuple containing first, a list with the numbers of the nodes for which the expression in CTL is true (or an empty list if its not the case), and the second element of the tuple is the whole tree representation. The list of nodes is obtained by the function *sat*, which takes the CTL expression and, calling specific functions (e.g., *satAX*, *satEF*) when necessary, returns the list of nodes satisfying the expression:

```
sat :: Node -> Phi -> [ [Integer] ]
```

The syntax for using the CTL applications is as follows:

```
check <task-algebra-expression> <CTL-expression>
```

where *task-algebra-expression* is a string expressing a well-formed expression in the algebra, and *CTL-expression* is a valid expression in CTL. In this case, the result is expressed as a pair of values containing first, the set of nodes for which the CTL expression is true (if there is any), and second, the structure of the tree built from the traces.

In CTL for example, we could construct a theorem asserting that the task *enterPassword* was eventually reachable at least once (viz. in at least one trace), and test this against the algebra expression denoting the diagram in Figure 8.4:

```
MyCTL> check "(Phi+Epsilon+(requestPassword; Epsilon + Phi));
enterPassword" (EF (Pr "enterPassword"))

([[0],[0,1],[0,2],[0,2,1]],Node ([0],null) [Node ([0,3],Phi)
[Empty],Node ([0,2],requestPassword) [Node ([0,2,2],Phi) [Empty],Node
[0,2,1],enterPassword) [Empty]],Node ([0,1],enterPassword) [Empty]])
```

As mentioned above, the result of *check* is a tuple containing first, a list with the numbers of the nodes for which the expression in CTL and the second element of the tuple is the whole tree representation. The node [0] is null and it is used to integrate all the paths. The enumeration of the nodes is consecutive and also depicts the level of the node in the tree. Consequently, subnodes of node [0] are nodes [0, 1], [0, 2] and [0, 3]. Node [0, 2] have as subnodes to [0, 2, 1] and [0, 2, 2]. As can be seen, the CTL application returns the nodes that are considered valid under the CTL expression ([0],[0,1],[0,2],[0,2,1]); where nodes [0,1] and [0,2,1] are the nodes representing the states where the task *enterPassword* happens. Figure 8.7 shows a visual tree representation where the valid states can be observed.

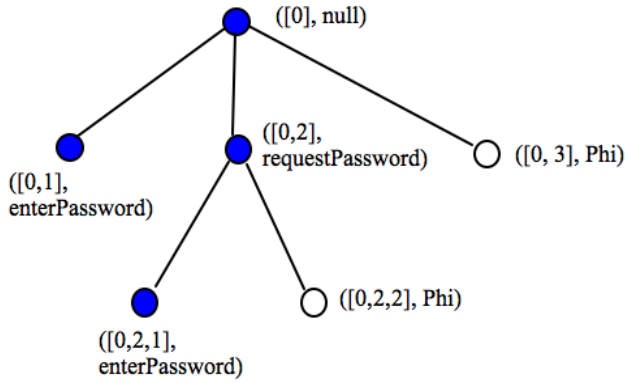

Figure 8.7. Tree representation of traces from diagram in Figure 8.4

Again, we may test the same CTL formula against the larger diagram from Figure 8.5 where the compound task described in Figure 8.4 is embedded. The Haskell command and the corresponding result are presented as follows (see Figure 8.8 for the visual tree representation):

```
MyCTL> check "let Login={(Phi+Epsilon+(requestPassword; Epsilon +
Phi)); enterPassword} {readReviewerGuidelines + (Mu.x(Login; Epsilon
+ x); checkPaymentsStatus + completeReviewEForm +
(Mu.x((Mu.y(readAnAbstract; Epsilon + y); selectPaper); Epsilon + x);
confirmSelection; receivePapers))}" (EF (Pr "enterPassword"))

([[0],[0,1],[0,3],[0,3,1]],Node ([0],null) [Node ([0,4],Phi)
[Empty],Node ([0,3],requestPassword) [Node ([0,3,2],Phi) [Empty],Node
([0,3,1],enterPassword) [Node ([0,3,1,6],Phi) [Empty],Node
([0,3,1,5],requestPassword) [Node ([0,3,1,5,2],Phi) [Empty],Node
([0,3,1,5,1],enterPassword) [Node ([0,3,1,5,1,3],readAnAbstract)
[Node ([0,3,1,5,1,3,2],selectPaper) [Node
([0,3,1,5,1,3,2,2],readAnAbstract) [Node
([0,3,1,5,1,3,2,2,2],selectPaper) [Node
([0,3,1,5,1,3,2,2,2,1],confirmSelection) [Node
([0,3,1,5,1,3,2,2,2,1,1],receivePapers) [Empty]]],Node
([0,3,1,5,1,3,2,2,1],readAnAbstract) [Node
([0,3,1,5,1,3,2,2,1,1],selectPaper) [Node
([0,3,1,5,1,3,2,2,1,1,1],confirmSelection) [Node
([0,3,1,5,1,3,2,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,3,1,5,1,3,2,1],confirmSelection) [Node
([0,3,1,5,1,3,2,1,1],receivePapers) [Empty]]],Node
([0,3,1,5,1,3,1],readAnAbstract) [Node
([0,3,1,5,1,3,1,1],selectPaper) [Node
([0,3,1,5,1,3,1,1,2],readAnAbstract) [Node
([0,3,1,5,1,3,1,1,2,2],selectPaper) [Node
([0,3,1,5,1,3,1,1,2,2,1],confirmSelection) [Node
([0,3,1,5,1,3,1,1,2,2,1,1],receivePapers) [Empty]]],Node
([0,3,1,5,1,3,1,1,2,1],readAnAbstract) [Node
([0,3,1,5,1,3,1,1,2,1,1],selectPaper) [Node
([0,3,1,5,1,3,1,1,2,1,1,1],confirmSelection) [Node
([0,3,1,5,1,3,1,1,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,3,1,5,1,3,1,1,1],confirmSelection) [Node
([0,3,1,5,1,3,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,3,1,5,1,2],completeReviewEForm) [Empty],Node
([0,3,1,5,1,1],checkPaymentsStatus) [Empty]]],Node
([0,3,1,4],readAnAbstract) [Node ([0,3,1,4,2],selectPaper) [Node
([0,3,1,4,2,2],readAnAbstract) [Node ([0,3,1,4,2,2,2],selectPaper)
```

```
[Node ([0,3,1,4,2,2,2,1],confirmSelection) [Node
([0,3,1,4,2,2,2,1,1],receivePapers) [Empty]]],Node
([0,3,1,4,2,2,1],readAnAbstract) [Node
([0,3,1,4,2,2,1,1],selectPaper) [Node
([0,3,1,4,2,2,1,1,1],confirmSelection) [Node
([0,3,1,4,2,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,3,1,4,2,1],confirmSelection) [Node
([0,3,1,4,2,1,1],receivePapers) [Empty]]],Node
([0,3,1,4,1],readAnAbstract) [Node ([0,3,1,4,1,1],selectPaper) [Node
([0,3,1,4,1,1,2],readAnAbstract) [Node
([0,3,1,4,1,1,2,2],selectPaper) [Node
([0,3,1,4,1,1,2,2,1],confirmSelection) [Node
([0,3,1,4,1,1,2,2,1,1],receivePapers) [Empty]]],Node
([0,3,1,4,1,1,2,1],readAnAbstract) [Node
([0,3,1,4,1,1,2,1,1],selectPaper) [Node
([0,3,1,4,1,1,2,1,1,1],confirmSelection) [Node
([0,3,1,4,1,1,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,3,1,4,1,1,1],confirmSelection) [Node
([0,3,1,4,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,3,1,3],enterPassword) [Node ([0,3,1,3,3],readAnAbstract) [Node
([0,3,1,3,3,2],selectPaper) [Node ([0,3,1,3,3,2,2],readAnAbstract)
[Node ([0,3,1,3,3,2,2,2],selectPaper) [Node
([0,3,1,3,3,2,2,2,1],confirmSelection) [Node
([0,3,1,3,3,2,2,2,1,1],receivePapers) [Empty]]],Node
([0,3,1,3,3,2,2,1],readAnAbstract) [Node
([0,3,1,3,3,2,2,1,1],selectPaper) [Node
([0,3,1,3,3,2,2,1,1,1],confirmSelection) [Node
([0,3,1,3,3,2,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,3,1,3,3,2,1],confirmSelection) [Node
([0,3,1,3,3,2,1,1],receivePapers) [Empty]]],Node
([0,3,1,3,3,1],readAnAbstract) [Node ([0,3,1,3,3,1,1],selectPaper)
[Node ([0,3,1,3,3,1,1,2],readAnAbstract) [Node
([0,3,1,3,3,1,1,2,2],selectPaper) [Node
([0,3,1,3,3,1,1,2,2,1],confirmSelection) [Node
([0,3,1,3,3,1,1,2,2,1,1],receivePapers) [Empty]]],Node
([0,3,1,3,3,1,1,2,1],readAnAbstract) [Node
([0,3,1,3,3,1,1,2,1,1],selectPaper) [Node
([0,3,1,3,3,1,1,2,1,1,1],confirmSelection) [Node
([0,3,1,3,3,1,1,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,3,1,3,3,1,1,1],confirmSelection) [Node
([0,3,1,3,3,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,3,1,3,2],completeReviewEForm) [Empty],Node
([0,3,1,3,1],checkPaymentsStatus) [Empty]],Node
([0,3,1,2],completeReviewEForm) [Empty],Node
([0,3,1,1],checkPaymentsStatus) [Empty]]],Node
([0,2],readReviewerGuidelines) [Empty],Node
([0,1],enterPassword) [Node ([0,1,6],Phi) [Empty],Node
([0,1,5],requestPassword) [Node ([0,1,5,2],Phi) [Empty],Node
([0,1,5,1],enterPassword) [Node ([0,1,5,1,3],readAnAbstract) [Node
([0,1,5,1,3,2],selectPaper) [Node ([0,1,5,1,3,2,2],readAnAbstract)
[Node ([0,1,5,1,3,2,2,2],selectPaper) [Node
([0,1,5,1,3,2,2,2,1],confirmSelection) [Node
([0,1,5,1,3,2,2,2,1,1],receivePapers) [Empty]]],Node
([0,1,5,1,3,2,2,1],readAnAbstract) [Node
([0,1,5,1,3,2,2,1,1],selectPaper) [Node
([0,1,5,1,3,2,2,1,1,1],confirmSelection) [Node
([0,1,5,1,3,2,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,1,5,1,3,2,1],confirmSelection) [Node
([0,1,5,1,3,2,1,1],receivePapers) [Empty]]],Node
([0,1,5,1,3,1],readAnAbstract) [Node ([0,1,5,1,3,1,1],selectPaper)
[Node ([0,1,5,1,3,1,1,2],readAnAbstract) [Node
```

```
([0,1,5,1,3,1,1,2,2],selectPaper) [Node
([0,1,5,1,3,1,1,2,2,1],confirmSelection) [Node
([0,1,5,1,3,1,1,2,2,1,1],receivePapers) [Empty]]],Node
([0,1,5,1,3,1,1,2,1],readAnAbstract) [Node
([0,1,5,1,3,1,1,2,1,1],selectPaper) [Node
([0,1,5,1,3,1,1,2,1,1,1],confirmSelection) [Node
([0,1,5,1,3,1,1,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,1,5,1,3,1,1,1],confirmSelection) [Node
([0,1,5,1,3,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,1,5,1,2],completeReviewEForm) [Empty],Node
([0,1,5,1,1],checkPaymentsStatus) [Empty]]],Node
([0,1,4],readAnAbstract) [Node ([0,1,4,2],selectPaper) [Node
([0,1,4,2,2],readAnAbstract) [Node ([0,1,4,2,2,2],selectPaper) [Node
([0,1,4,2,2,2,1],confirmSelection) [Node
([0,1,4,2,2,2,1,1],receivePapers) [Empty]]],Node
([0,1,4,2,2,1],readAnAbstract) [Node ([0,1,4,2,2,1,1],selectPaper)
[Node ([0,1,4,2,2,1,1,1],confirmSelection) [Node
([0,1,4,2,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,1,4,2,1],confirmSelection) [Node ([0,1,4,2,1,1],receivePapers)
[Empty]]],Node ([0,1,4,1],readAnAbstract) [Node
([0,1,4,1,1],selectPaper) [Node ([0,1,4,1,1,2],readAnAbstract) [Node
([0,1,4,1,1,2,2],selectPaper) [Node
([0,1,4,1,1,2,2,1],confirmSelection) [Node
([0,1,4,1,1,2,2,1,1],receivePapers) [Empty]]],Node
([0,1,4,1,1,2,1],readAnAbstract) [Node
([0,1,4,1,1,2,1,1],selectPaper) [Node
([0,1,4,1,1,2,1,1,1],confirmSelection) [Node
([0,1,4,1,1,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,1,4,1,1,1],confirmSelection) [Node
([0,1,4,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,1,3],enterPassword) [Node ([0,1,3,3],readAnAbstract) [Node
([0,1,3,3,2],selectPaper) [Node ([0,1,3,3,2,2],readAnAbstract) [Node
([0,1,3,3,2,2,2],selectPaper) [Node
([0,1,3,3,2,2,2,1],confirmSelection) [Node
([0,1,3,3,2,2,2,1,1],receivePapers) [Empty]]],Node
([0,1,3,3,2,2,1],readAnAbstract) [Node
([0,1,3,3,2,2,1,1],selectPaper) [Node
([0,1,3,3,2,2,1,1,1],confirmSelection) [Node
([0,1,3,3,2,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,1,3,3,2,1],confirmSelection) [Node
([0,1,3,3,2,1,1],receivePapers) [Empty]]],Node
([0,1,3,3,1],readAnAbstract) [Node ([0,1,3,3,1,1],selectPaper) [Node
([0,1,3,3,1,1,2],readAnAbstract) [Node
([0,1,3,3,1,1,2,2],selectPaper) [Node
([0,1,3,3,1,1,2,2,1],confirmSelection) [Node
([0,1,3,3,1,1,2,2,1,1],receivePapers) [Empty]]],Node
([0,1,3,3,1,1,2,1],readAnAbstract) [Node
([0,1,3,3,1,1,2,1,1],selectPaper) [Node
([0,1,3,3,1,1,2,1,1,1],confirmSelection) [Node
([0,1,3,3,1,1,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,1,3,3,1,1,1],confirmSelection) [Node
([0,1,3,3,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,1,3,2],completeReviewEForm) [Empty],Node
([0,1,3,1],checkPaymentsStatus) [Empty]],Node
([0,1,2],completeReviewEForm) [Empty],Node
([0,1,1],checkPaymentsStatus) [Empty]]])
```

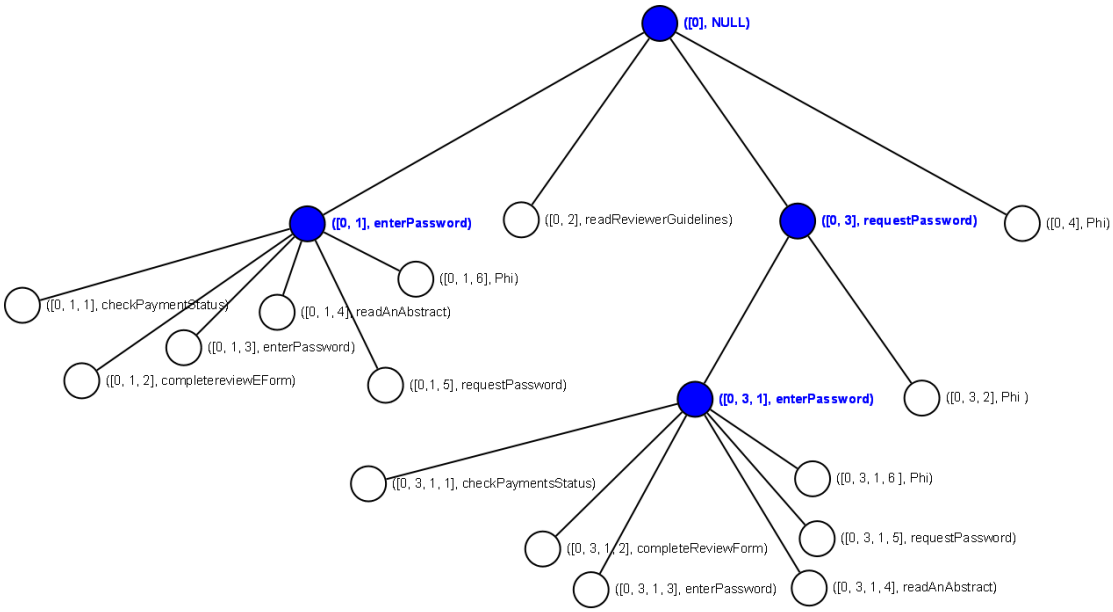

Figure 8.8 A partial tree representation of traces from diagram in Figure 8.5

Or it could be asked, for example, if either the password is always requested initially (the task *requestPassword* happens), or the task *readReviewerGuidelines* is always executed (the full tree representation is omitted):

```
MyCTL> check "let Login={(Phi+Epsilon+(requestPassword; Epsilon +
Phi)); enterPassword} {readReviewerGuidelines + (Mu.x(Login; Epsilon
+ x); checkPaymentsStatus + completeReviewEForm +
(Mu.x((Mu.y(readAnAbstract; Epsilon + y); selectPaper); Epsilon + x);
confirmSelection; receivePapers))}" (AX ( Or (Pr "requestPassword")
(Pr "readReviewerGuidelines")))

([],Node ([0],null) [Node ([0,4],Phi) [Empty],Node
([0,3],requestPassword) ...)
```

In this case, since the CTL theorem is falsified, executing the CTL command returns an empty list in the first part of the result (denoting no solutions) and the second part returns the constructed CTL trace tree, as before. In the next example, just an extract of the diagram in Figure 8.5 is analysed to verify whether, on at least one path, eventually *readAnAbstract* happens, followed by *selectPaper* (see Figure 8.9 for the visual tree representation):

```
MyCTL> check "{Mu.x((Mu.y(readAnAbstract; Epsilon + y); selectPaper);
Epsilon + x); confirmSelection; receivePapers}"
(EF(EU (Pr "readAnAbstract") (Pr "selectPaper")))

([[0],[0,1],[0,1,1],[0,1,1,1],[0,1,2]],Node ([0],null) [Node
([0,1],readAnAbstract) [Node ([0,1,2],selectPaper) [Node
([0,1,2,2],readAnAbstract) [Node ([0,1,2,2,2],selectPaper) [Node
([0,1,2,2,2,1],confirmSelection) [Node
([0,1,2,2,2,1,1],receivePapers) [Empty]]],Node
([0,1,2,2,1],readAnAbstract) [Node ([0,1,2,2,1,1],selectPaper) [Node
([0,1,2,2,1,1,1],confirmSelection) [Node
([0,1,2,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,1,2,1],confirmSelection) [Node ([0,1,2,1,1],receivePapers)
[Empty]]],Node ([0,1,1],readAnAbstract) [Node ([0,1,1,1],selectPaper)
[Node ([0,1,1,1,2],readAnAbstract) [Node ([0,1,1,1,2,2],selectPaper)
```

```
[Node ([0,1,1,1,2,2,1],confirmSelection) [Node
([0,1,1,1,2,2,1,1],receivePapers) [Empty]]],Node
([0,1,1,1,2,1],readAnAbstract) [Node ([0,1,1,1,2,1,1],selectPaper)
[Node ([0,1,1,1,2,1,1,1],confirmSelection) [Node
([0,1,1,1,2,1,1,1,1],receivePapers) [Empty]]]]],Node
([0,1,1,1,1],confirmSelection) [Node ([0,1,1,1,1,1],receivePapers)
[Empty]]]]]])
```

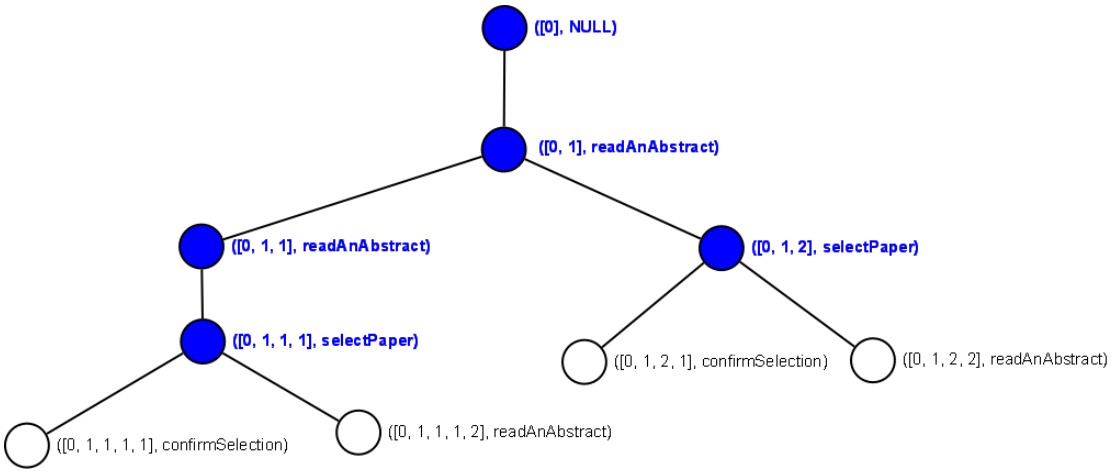

Figure 8.9 A partial tree representation of traces from an extract of the diagram in Figure 8.5

With the examples above it was shown that it is possible to model-check task flow models represented in the algebra, by testing CTL formulas against the corresponding traces. A tree is first built from the traces in order to use this branching-time logic and then the theorem is tested, yielding the set of branches in which the theorem holds.

8.5 Tests of the implementation

There are some attempts to apply LTL and CTL queries to traces. Traces semantics for positive core Xpath(a subset of XML Path Language), which translates into LTL an uses SPIN model checker [147]. Eleftherakis uses X-Machines to model and test software [148]. In [149, 150], Eleftherakis mentions XmCTL, an extended CTL including two memory quantifier operators. In [151], it is described a query checking tool using CTL. This tool works over XChek, a multi-valued model-checker [152].

In our project, the task algebra and the model-checking tools were developed in Haskell. The implementation for the task algebra was tested using small and medium sized examples. So far, no problems of execution were found while computing traces. The case of study, as was mentioned, is a medium sized project, and the limited number of loop-cycles also helps. In addition, the level of detail in the task flows is usually not as high as in programming.

Additionally, the implementation for the LTL and CTL queries with the examples presented here, and others of similar size are executed in a matter of seconds. As can be expected, CTL queries are more time consuming because the tree has to be constructed from the trace sets and because exhaustive searching is needed in some queries. Still, it takes no more than a few seconds to obtain the results from the queries. However, optimising the source code in Haskell or moving the implementation to another language could increase the performance.

As mentioned above, the source code for the task algebra, and the LTL and CTL queries can be seen in Appendix C.

8.6 Summary

The previous chapters demonstrated the soundness and congruence of the task algebra. In this chapter, an implementation of the algebra in the Haskell programming language was described using a previously published Task Flow Diagram case study and translating the diagrams from the case study into the task algebra. The traces generated by the program were then the subject of queries about (in-)equality, tested using set operations, and more general theorems about temporal logic properties, tested using LTL and CTL theorems.

Chapter 9:
Conclusions

In the previous chapter, the implementation of the algebra and the LTL/CTL query tools were presented. In this chapter, the research results are summarized and the expected contribution is shown. The possible future work of this research is also mentioned.

9.1 Results

Software has become increasingly important in everyday life; yet while we are more dependent on it, it does not appear to be getting more reliable. What is needed is a better way to carefully examine software for accuracy and reliability. In order for software to be amenable to such an examination, it should have a precise specification. However, it should be clear that the aim of specification is to communicate the problem that we want to solve between users, designers and programmers. Therefore, as Henderson mentions in [153], a formal specification must be as elegant and precise as mathematics, but must also be comprehensible to, and readable by, people with different backgrounds. These are contradictory needs and, for that reason, it is difficult to find the ideal balance between them. Traditionally, software engineering follows the approach emphasising readability, through intuitive diagrammatic notations, with the expected problem of semantic imprecision and the difficulty of checking the specification.

Consequently, major effort should continue to be directed into modelling software using notations with precise semantics. Pursuing this intention, an experimental approach was first followed, using Alloy to define and verify the abstract syntax of the diagrams in the Discovery Method, where these were chosen for their clarity and simplicity over full UML notations. Initially, the goal was to go further and define the semantics for the Discovery Method using the Alloy analyzer. This approach was eventually abandoned, due to problems encountered in restricting the scope of the analyzer.

It was then decided to limit the scope of the research to providing a precise semantics for the Task Model, consisting of Task Structure and Task Flow diagrams. The abstract syntax representation for the Task Flow model in the Discovery Method was presented in chapter 5. This abstract syntax was used to define a task algebra, which is based on simple and compound tasks structured using operators such as sequence, selection, and parallel composition. Recursion and encapsulation were also considered. The axioms of the algebra were presented as well as a set of examples

showing a combination of basic elements in the expressions denoting simple, and more complex, Task Flow diagrams. The task algebra was able to represent task models in a clear and elegant style. Task Flow diagrams were also related to State Diagrams and Task Structure diagrams, as described in chapter 4.

Subsequently, the precise semantics for the abstract task algebra was developed. The semantics was designed in terms of trace sets representing all possible complete execution paths for a system of tasks. The soundness and congruence of the task algebra was proved in Chapter 7 and Appendix B respectively. In addition, an implementation of the algebra in the Haskell programming language was developed and presented in chapter 8. The traces generated by the program were then analysed by submitting these to queries about (in-)equality, tested using set operations, and more general theorems about temporal logic properties, tested using LTL and CTL theorems.

In addition, a previously published case- study was used to validate the task algebra. The diagrams were translated from the case- study into the abstract syntax; the semantic traces were generated, and these were then submitted to queries in temporal logic to check for a selection of properties.

Subsequently, as an outcome of our research, it is now possible to have formal representations of the Task Model as used in the Discovery Method. Task Models can be used to represent in a precise way the interactions between tasks. The developer creating the task models does not need to learn any complicated formal language to create the intended formal specification. The Task Model is itself equivalent to the formal specification, since it has a fully formal interpretation.

From this perspective, it is believed that the work presented here could be easy to integrate into the process of modelling software. By itself, the task algebra is an easy and simple enough formalism to be used even by software engineers having little previous experience with formal languages. The abstract syntax and axiomatic rules offer a means of proving the equivalence (or otherwise) of different workflow-based systems. However, the intention of this approach is eventually to provide a graphical tool to generate the diagrams and translate them automatically into the abstract syntax, so that the developer need not generate the representation in the formal language by hand.

9.2 Evaluation

All objectives of this research were achieved with different levels of satisfaction. Whilst our approach proposes a precise abstract syntax for tasks and activities, the relationship between the Task Structure and the Task Flow diagram and the restrictions that this relationship defines have to be taken into account by the developer, because these relationships are not established directly in the semantics. Additionally, the task algebra is limited by not representing guards on selections. In the actual proposal, selection is just represented as a commitment that a choice happened in a trace (which is common in other trace-based approaches), but no more information about the guards represented in the Task Flow diagram is mapped into the algebra. This is a desirable characteristic to be considered in future version of the algebra.

The denotational semantics were proposed in term of traces presenting a non-interleaving model for parallel composition due to the necessity to restrict the interleaving for special cases involving the succeed and fail symbols that have the behaviour of finishing the execution of the activity in which they appear. The commit symbol has also a special treatment, compared with an identifier representing a simple task, being inserted at a selection point for each choice. In addition, soundness and congruence is presented, but congruence was proved from the semantics and to be completely valid the completeness property still has to be proved.

The last objective proposed to test the feasibility of the formal representation. This implementation was presented and also included additional implementations of model-checking tools, so taking advantage of the task algebra implementation.

9.3 Future work

Inevitably, this research is never finished, it is merely published at a certain moment when a line must be drawn under the work. Some further work considered important for the future of this research involves in different aspects of the project.

As was mentioned above, a graphical tool is still needed to generate the diagrams and translate them into the algebra. This tool, which is envisaged as future work, will support the automatic construction and simplification of formal models by developers, directly from diagram specifications. In addition, further work should be invested in visualising the results from applying LTL and, particularly, CTL queries, which are not easy to interpret in their current form. A graphical browser, capable of visualising and navigating over trees of traces, should help the developer understand the results of queries.

In addition, for a future version of the algebra, it will be useful to add support for mapping the guards from the task diagrams into the algebra. The guards should allow the use of values, variables and logical operators, which are allowed by the Task Flow diagrams in the Discovery Method. Modelling guards will have clearly an impact in the semantics and it will enhance the kind of queries to be applied over the models.

Finally, the formal semantics developed in this work is sufficient to demonstrate how the representation of simple and compound tasks is unified at the lowest level of representation. This is enough to represent the tasks in some detail, to determine whether or not these tasks succeed or fail in their execution. Nevertheless, the semantics abstracts over the details of choices taken in conditional expressions and does not further analyse simple tasks, which are considered atomic. A future development in the semantics should consider including a suitable abstract representation of the states tested in conditional expressions. This could be developed, if simple tasks were decomposed further to express, in some form, how they affected system states, which could be modelled as the atomic postconditions of each task. This would support more detailed reasoning about the triggering of different branches and the ability to handle exceptional cases, based on satisfied and unsatisfied atomic preconditions. What kind of symbolic calculus would be sufficient to represent such atomic stateful properties is a matter for future research.

References

1. Rumbaugh, J., Jacobson, I. and Booch, G. *The Unified Modeling Language. Reference Manual*. Addison-Wesley, 1999.

2. Booch, G., Rumbaugh, J. and Jacobson, I. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

3. France, R.B., Ghosh, S., Dinh-Trong, T. and Solberg, A. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *COMPUTER*, 59-66, 2006.

4. UML 2.0 Superstructure Specification, Object Management Group, Framingham, Massachusetts, 2004.

5. Bruel, J.-M. and R.B.France, Transforming UML models to formal specifications. In Proceedings of *UML'98 - Beyond the notation*, (1998), Springer Verlag.

6. Brodsky, S., Clark, T., Cook, S., Evans, A.S. and Kent, S., Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach, *Technical Report of pUML Group*, 2000, http://www.puml.org/mmf/mmf.pdf, Last access: April 2004

7. D'Souza, D.F. and Wills, A.C. *Objects, Components, and Frameworks. The Catalysis Approach*. Addison-Wesley, 1999.

8. 2U Consortium, 2U Consortium, Unambiguous UML, 2001, http://www.2uworks.org/, Last access: September 2004

9. Kim, S.K. and Carrington, D. A Formal Mapping between UML Models and Object-Z Specifications. *ZB 2000: Formal Specification and Development in Z and B, Lecture Notes in Computer Science, ed. G. Goos, J. Hartmanis and J. van Leeuwen*, *1878*, 2-21, 2000.

10. Jackson, D. Object Models as Heap Invariants. in McIver, A. and Morgan, C. eds. *Programming Methodology*, Springer Verlag, New York, 2003, 247-268.

11. Massoni, T., Gheyi, R. and Borba, P., Semantics-Preserving Transformation for UML Class Diagrams, *Universidade Federal de Pernambuco*, 2004, http://nazare.cin.ufpe.br/twiki/pub/SPG/WeeklySeminar/paper-uml2004.pdf, Last access: May 2004

12. Naumenko, A. and Wegmann, A. A Metamodel for the Unified Modeling Language. *«UML» 2002 — The Unified Modeling Language, Lecture Notes in Computer Science, ed. G. Goos, J. Hartmanis, and J. van Leeuwen*, *2460*, 2-17, 2002.

13. ISO, IEC and ITU-T. The Reference Model of Open Distributed Processing (RM-ODP, ITU-T Rec. X.901-X.904 | ISO/IEC 10746), 1995-2008.

14. Bordbar, B. and Anastasakis, K., UML2Alloy: A tool for lightweight modelling of Discrete Event Systems. In Proceedings of *the International Conference on Software Engineering Reserach and Practice (SERP05)*, (Las Vegas, USA, 2005), 209–216.

15. Zito, A. and Dingel, J., Modeling UML2 package merge with Alloy. In Proceedings of *First Alloy Workshop*, (Portland, USA, 2006), ACM-MIT.

16. Khurshid, S. and Marinov, D., Aaree: A Recipe for Analyzing Object-Oriented Models, Cambridge, MA, USA, 2001, http://sdg.lcs.mit.edu/pubs/2001/aaree.pdf, Last access: January 2005

17. Gallardo, M.d.M., Merino, P. and Pimentel, E. Debugging UML designs with model checking. *Journal of Object Technology*, *1* (2), 101-117, 2002.

18. Richters, M. A Precise Approach to Validating UML Models and OCL Constraints, PhD Thesis, Universitaet Bremen, Berlin, 2002, 218 pp.

19. OMG. Object Constraint Language Specification. in *OMG Unified Modeling Language Specification*, OMG, 2003.

20. OMG. *Object Constraint Language Specification, version 2.0*. OMG, 2006.

21. Shen, W., Compton, K. and Huggins, J., A toolset for supporting UML static and dynamic model checking. In Proceedings of *IEEE Automated software Engineering*, (2001), 315-318.

22. Gnesi, S. and Mazzanti, F., On the fly model checking of communicating UML state machines, 2003, http://fmt.isti.cnr.it/WEBPAPER/onthefly-SERA04.pdf, Last access: February 2005

23. Störrle, H. Semantics and Verification of Data Flow in UML 2.0 Activities. *Electronic Notes in Theoretical Computer Science*, *127* (4), 35-52, 2005.

24. Xactium, XMF-Mosaic, *Xactium*, 2005, http://albini.xactium.com/content/index.php?option=com_frontpage&Itemid=1, Last access: March 2005

25. Simons, A.J.H., Object Discovery - A process for developing medium-sized applications. In Proceedings of *Tutorial 14, 12th European Conference on Object-Oriented Programming (ECOOP '98)*, (Brussels, 1998), AITO/ACM, 109.

26. Simons, A.J.H., Object Discovery - A process for developing applications. In Proceedings of *Workshop 6, British Computer Society SIG OOPS Conference on Object Technology (OT '98)*, (Oxford, 1998), BCS, 93.

27. Rumpe, B., A Note on Semantics (with Emphasis on UML). In Proceedings of *Second ECOOP Workshop on Precise Behavioral Semantics*, (Brussels, 1998), Technische Universität München, TUM-I9813, 177-197.

28. Simons, A.J.H. COM3410 Systems Analysis and Design, Part 3 : Object Modelling – Command and Control, The University of Sheffield, 2006, 52-55.

29. Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Hierons, R.M., Kapoor, K., Krause, P., Luettgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R. and Zedan, H., Working together: Formal Methods and Testing, *Formal Methods and Testing Network (FORTEST)*, 2003, http://www.fortest.org.uk/documents/landscape3.pdf, Last access: June 2004

30. Wing, J.M. A Specifier's Introduction to Formal Methods. *IEEE Computer*, *23* (9), 8-24, 1990.

31. Spivey, J.M. An Introduction to Z and Formal Specifications. *Software Engineering Journal IEEE/BCS*, *4* (1), 40-50, 1989.

32. Khurshid, S., Marinov, D. and Jackson, D., An analyzable annotation language. In Proceedings of *17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, (2002), 231-245.

33. ISO/IEC. International Standard ISO/IEC 13568: Formal Specification—Z Notation—Syntax, Type and Semantics, 2002, 196.

34. Jackson, D. *Software abstractions : logic, language and analysis*. MIT Press, Cambridge, Mass., 2006.

35. Simons, A.J.H. The Theory of Classification, Part 5: Axioms, Assertions and Suptyping. *Journal of Object Technology*, *2* (1), 13-21, 2003.

36. Jackson, D., Micromodels of Software: Lightweight Modelling and Analysis with Alloy, *MIT Lab for Computer Science*, 2002, http://alloy.mit.edu/reference-manual.pdf, Last access: January 2004

37. Clarke, E.M., Grumberg, O. and Peled, D. *Model checking*. MIT Press, Cambridge, Mass., 1999.

38. Clarke, E.M. and Berezin, S., Model Checking: Historical Perspective and Example. In Proceedings of *Analytic Tableaux and Related Methods (TABLEAU '98)*, (Oisterwijk near Tilburg, The Netherlands, 1988), 18-24.

39. McMillan, K.L. *Symbolic model checking*. Kluwer Academic, Boston, 1993.

40. Claessen, K. and Sorensson, N., New techniques that improve MACE-style finite model finding. In Proceedings of *CADE-19, Workshop W4. Model Computation - Principles, Algorithms, Applications*, (Miami, USA, 2003).

41. Spivey, J.M. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.

42. Potter, B., Sinclair, J. and Till, D. *An Introduction to Formal Specification and Z*. Prentice Hall, 1996.

43. Woodcock, J. and Davies, J. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.

44. Sun, J., Dong, J.S., Liu, J. and Wang, H. Z Family on the Web with their UML Photos. Technical Report TRA1-01. *School of Computing, National University of Singapore*, 2001.

45. Toyn, I. and McDermid, J.A. CADi: An architecture for Z tools and its implementation. *Software: Practice and Experience*, *25* (3), 1995.

46. Malik, P. and Utting, M. CZT: A framework for Z tools. *ZB 2005: Formal Specification and Development in Z and B, Lecture Notes in Computer Science, ed. G. Goos, J. Hartmanis and J. van Leeuwen*, *3455*, 65-84, 2005.

47. Community Z Tools (CZT), *Community Z Tools Project*, 2007, http://czt.sourceforge.net/, Last access: April 2009

48. Dupuy, S., Ledru, Y. and Chabre-Peccoud, M., An Overview of RoZ : A Tool for Integrating UML and Z Specifications. In Proceedings of *Advanced Information Systems Engineering: 12th International Conference, CAiSE 2000*, (Stockholm, Sweden, 2000), Springer-Verlag, 417-430.

49. Jackson, D., Alloy 2.0 Tutorial., *MIT*, Cambridge, 2002, http://web.mit.edu/~rseater/www/tutorial2/alloy-tutorial.html, Last access: January 2004

50. Jackson, D. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *11* (2), 256-290, 2002.

51. Software Design Group, The Alloy Analyzer - 3.0 Beta, *MIT Laboratory for Computer Science*, Cambridge, MA, 2004, http://alloy.mit.edu/beta/index.php, Last access: June 2004

52. Jackson, D., Alloy 3.0: Reference Manual, *MIT*, Cambridge, 2004, http://alloy.mit.edu/beta/reference-manual.pdf, Last access: May 2004

53. Torlat, E. and Jackson, D., Kodkod: A Relational Model Finder. In Proceedings of *Algorithms for Construction and Analysis of Systems (TACAS '07)*, (Braga, Portugal, 2007).

54. Dennis, G., Seater, R., Rayside, D. and Jackson, D., Lightweight formal methods applied to a radiotherapy machine component, 2004, http://sdg.lcs.mit.edu/publications.html, Last access: March 2004

55. Jackson, D. and Vaziri, M., Finding bugs with a constraint solver. In Proceedings of *the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, (Portland, Oregon, USA, 2000), ACM Press, 14-25.

56. Minas, M., Visual specification of visual editors with DiaGen. In Proceedings of *International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'03)*, (Charlottesville, Virginia, USA, 2003), 461-466.

57. Jackson, D., Shlyakhter, I. and Sridharan, M., A Micromodularity Mechanism. In Proceedings of *9th ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC '01)*, (Vienna, Austria, 2001), ACM, 62-73.

58. Jackson, D., Alloy 3.0 Tutorial., *MIT*, Cambridge, 2004, http://web.mit.edu/~rseater/www/tutorial3/alloy-tutorial.html, Last access: May 2004

59. Edwards, J., Jackson, D. and Torlak, E., A Type System for Object Models. In Proceedings of *Foundations of Software Engineering*, (Newport Beach, CA, 2004), To be published.

60. Richters, M. and Gogolla, M. OCL: Syntax, Semantics, and Tools. in Clark, A. and Warmer, J. eds. *Object Modeling with the OCL*, Springer-Verlag, Berlin, 2002, 42-68.

61. Hussmann, H., Demuth, B. and Finger, F., Modular architecture for a toolset supporting OCL. In Proceedings of *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference*, (York, UK, 2000), Springer, 278-293.

62. Toval, A., Requena, V. and Fernández, J.L. Emerging OCL tools. *Software and Systems Modeling*, *2* (4), 248-261, 2003.

63. Richters, M. and Gogolla, M., Validating UML Models and OCL Constraints. In Proceedings of *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference*, (York, UK, 2000), Springer, 265-277.

64. Ramirez, A., Vanpeperstraete, P., Rueckert, A., Odutola, K., Bennett, J. and Tolke, L., ArgoUML User Manual, 2004, http://argouml.tigris.org/documentation/defaulthtml/manual/, Last access: December 2004

65. Gogolla, M. and Richters, M., On Constraints and Queries in UML. In Proceedings of *UML Workshop 1997*, (1997), 109-121.

66. Gogolla, M. and Richters, M., On Formalizing the UML Object Constraint Language OCL. In Proceedings of *Conceptual Modeling - ER '98, 17th International Conference on Conceptual Modeling*, (Singapore, 1998), Springer, 449-464.

67. Richters, M. and Gogolla, M., On the need for a precise OCL semantics. In Proceedings of *Proc. OOPSLA Workshop ``Rigorous Modeling and Analysis with the UML: Challenges and Limitations''*, (Colorado State University, Fort Collins, Colorado, 1999).

68. Hahnle, R. and Ranta, A., Connecting OCL with the Rest of the World. In Proceedings of *ETAPS 2001 Workshop on Transformations in UML*, (Genova, Italy, 2001).

69. Baeten, J.C.M. A brief history of process algebra. CSR 04-02. *Technische Universiteit Eindhoven*, 2004.

70. Cheng, M.H.M., Calculus of Communicating Systems: a synopsis, 1994, citeseer.ist.psu.edu/cheng94calculu.html

71. Glabbeek, R.J.v. Notes on the methodology of CCS and CSP. *Theoretical Computer Science*, *177* (2), 329-349, 1997.

72. Bergstra, J.A. and Klop, J.W. Fixed point semantics in process algebras. IW 208. *Mathematical Centre*, Amsterdam, 1982.

73. Milner, R. *A calculus of communicating systems*. Springer-Verlag, Berlin; New York, 1980.

74. Hoare, C.A.R. *Communicating sequential processes*. Prentice/Hall International, Englewood Cliffs, N.J., 1985.

75. Milner, R. *Communicating and mobile systems: the pi-calculus*. Cambridge University Press, Cambridge, UK, 1999.

76. Bergstra, J.A. and Klop, J.W., The Algebra of Recursively Defined Processes and the Algebra of Regular Processes. In Proceedings of *11th ICALP*, (1984), Springer Verlag, 82-95.

77. Hoare, C.A.R. Communicating sequential processes. *Communications of the ACM*, *21* (8), 666-677, 1978.

78. Hoare, C.A.R. A Model for Communicating Sequential Processes. PRG-22. *University of Oxford*, Oxford, 1981.

79. Brookes, S.D., Hoare, C.A.R. and Roscoe, A.W. A Theory of Communicating Sequential Processes. *J. ACM*, *31* (3), 560-599, 1984.

80. Hall, A. Seven Myths of Formal Methods. *IEEE Software*, *7* (5), 11-19, 1990.

81. Bowen, J.P. and Hinchey, M.G. Seven More Myths of Formal Methods: Dispelling Industrial Prejudices. *IEEE Software*, *12* (4), 34-41, 1995.

82. Bowen, J.P. and Hinchey, M.G. Ten Commandents of Formal Methods. *IEEE Computer*, *28* (4), 56-63, 1995.

83. Jackson, D., A Comparison of Object Modelling Notations: Alloy, UML and Z, *MIT*, 1999, http://sdg.lcs.mit.edu/~dnj/publications/alloy-comparison.pdf, Last access: February 2004

84. Wallace, C. Using Alloy in Process Modelling. *Information and Software Technology*, *45* (15), 1011-1089, 2003.

85. Henderson-Sellers, B., Simons, A. and Youessi, H. *The OPEN toolbox of techniques*. Harlow:Addison-Wesley, 1998.

86. Graham, I., Henderson-Sellers, B. and Younessi, H. *The OPEN Process Specification*. Addison-Wesley, 1997.

87. Shlaer, S. and Mellor, S.J. *Object-oriented systems analysis : modeling the world in data*. Yourdon Press, Englewood Cliffs, N.J., 1988.

88. Beck, K. and Cunningham, W., A laboratory for teaching object-oriented thinking. In Proceedings of *OOPSLA '89, pub. Sigplan Notices*, (1989), ACM Sigplan.

89. Wirfs-Brock, R. and Wilkerson, B., Object-oriented design: a responsibility-driven approach. In Proceedings of *OOPSLA '89, pub. Sigplan Notices*, (1989), ACM Sigplan.

90. Wirfs-Brock, R., Wilkerson, B. and Wiener, L. *Designing object-oriented software*. Prentice Hall, Englewood Cliffs, N.J., 1990.

91. Coad, P. and Yourdon, E. *Object-oriented analysis*. Yourdon Press, Englewood Cliffs, N.J., 1991.

92. Coad, P. and Yourdon, E. *Object-oriented design*. Yourdon Press, Englewood Cliffs, N.J., 1991.

93. Rumbaugh, J. *Object-oriented modeling and design*. Prentice Hall, Englewood Cliffs, N.J., 1991.

94. Booch, G. *Object oriented design with applications*. Benjamin/Cummings Pub. Co., Redwood City, Calif., 1991.

95. Jacobson, I. *Object-oriented software engineering : a use case driven approach*. ACM Press, Addison-Wesley Pub., New York, Wokingham, Eng. ; Reading, Mass., 1992.

96. Booch, G. *Object-oriented analysis and design with applications*. Benjamin/Cummings Pub. Co., Redwood City, Calif., 1994.

97. Rumbaugh, J. *OMT insights : perspectives on modeling from the Journal of Object-Oriented Programming*. SIGS Books, New York, 1996.

98. Hewlett-Packard *The Fusion Object-Oriented Analysis and Design Method*. HP Laboratories, Bristol, England, 1992.

99. Coleman, D. *Object-oriented development : the fusion method*. Prentice Hall, Englewood Cliffs, N.J., 1994.

100. Henderson-Sellers, B. and Edwards, J.M. *Book two of object-oriented knowledge : the working object : object-oriented software engineering : methods and management*. Prentice Hall, Sydney, Australia; New York, 1994.

101. Walden, K. and Nerson, J.-M. *Seamless object-oriented software architecture : analysis and design of reliable systems*. Prentice Hall, New York, 1995.

102. Henderson-Sellers, B. and Simons, A.J.H. The OPEN Software Engineering Process Architecture: From Activities to Techniques. *Journal of Research and Practice in Information Technology*, *32* (1), 47-68, 2000.

103. OPEN Consortium, The OPEN website, *OPEN Consortium*, 2000, http://www.open.org.au/, Last access: April 2004

104. Graham, I. *Migrating to object technology*. Addison-Wesley Pub. Co., Wokingham, England ; Reading, Mass., 1995.

105. Firesmith, D., The Firesmith Method - Quo Vadis. In Proceedings of *TOOLS USA'95 Technology of Object-Oriented Languages and Systems 17 International Conference and Exhibition*, (Interactive Software Engineering, Santa Barbara, California, 1995).

106. Rawson, M. and Allen, P., Synthesis - An Integrated, Object-Oriented Method and Tool for Requirements Specification in Z. In Proceedings of *Methods Integration Workshop*, (Leeds, UK, 1996).

107. Reenskaug, T., Wold, P. and Lehne, O.A. *Working with objects : the OOram software engineering method*. Manning, Greenwich, 1996.

108. Simons, A.J.H., The Discovery EBook, *The Department of Computer Science, University of Sheffield*, Sheffield, 2000, http://www.dcs.shef.ac.uk/~ajhs/discovery/ebook/, Last access: June 2004

109. Derrick, J., Akehurst, D. and Boiten, E., A framework for UML consistency. In Proceedings of *Workshop on Consistency Problems in UML-based Software Development*, (Dresden, Germany, 2002), 30-45.

110. Booch, G., Microsoft and Domain Specific Languages *Handbook of Software Architecture*, 2004, http://www.booch.com/architecture/blog.jsp?archive=2004-12.html, Last access: February 2004

111. OMG, UML Resource Page, *Object Management Group*, 1997, http://www.uml.org/, Last access: June 2007

112. Simons, A.J.H. and Graham, I. 30 things that go wrong in Object Modelling With UML 1.3. in Kilov, H., Rumpe, B. and Simmonds, I. eds. *Behavioral Specifications of Businesses and Systems*, Kluwer Academic Publishers, 1999, 237-257.

113. Simons, A.J.H., Use Cases Considered Harmful. In Proceedings of *29th Conf. Tech. Obj.-Oriented Prog. Lang. and Sys., (TOOLS-29 Europe)*, (Los Alamitos, CA, 1999), IEEE Computer Society, 194-203.

114. Lano, K.C. and Evans, A.S., Rigorous Development in UML. In Proceedings of *ETAPS'99, FASE workshop*, (1999).

115. Beck, K., Beedle, M., Bennekum, A.v., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J. and Thomas, D., Manifesto for Agile Software Development, 2001, http://agilemanifesto.org/, Last access: May 23

116. Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, *8* (3), 231-274, 1987.

117. Simons, A.J.H., On the compositional properties of UML statechart diagrams. In Proceedings of *Electronic Workshops in Computing: Rigorous Object-Oriented Methods 2000*, (British Computer Society, 2000).

118. Chen, P. The Entity-Relationship Model-Toward a Unified View of Data. *ACM Transactions on Database Systems*, *1* (1), 9-36, 1976.

119. Génova, G., Llorens, J. and Palacios, V. Sending Messages in UML. *Journal of Object Technology*, *2* (1), 99-115, 2003.

120. Henderson-Sellers, B., Firesmith, D.G., Graham, I. and Simons, A.J.H. Instanting the process metamodel. *Journal of Object-Oriented Programming (ROAD)*, *12* (3), 51-57, 1999.

121. Downs, E., Clare, P. and Coe, I. *Structured systems analysis and design method : application and context*. Prentice Hall, New York, 1992.

122. Høydalsvik, G.M. and Sindre, G., On the purpose of object-oriented analysis. In Proceedings of *the eighth annual conference on Object-oriented programming systems, languages, and applications*, (1993), 240-255.

123. Beckert, B., Keller, U. and Schmitt, P.H., Translating the object constraint language into first-order predicate logic. In Proceedings of *VERIFY, Workshop at Federated Logic Conferences (FLoC)*, (Copenhagen, Denmark, 2002).

124. Simons, A.J.H., Snoeck, M. and Hung, K.S.Y. Using design patterns to reveal the competence of object-oriented methods in system-level design. *International Journal of Computer Systems Science and Engineering*, *14*, 343, 1999.

125. Simons, A.J.H., Snoeck, M. and Hung, K.S.Y., Design Patterns as Litmus Paper to Test the Strength of Object-Oriented Methods. In Proceedings of *5th. Int. Conf. Object-Oriented Info. Sys*, (1998), 129-147.

126. Gamma, E. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass., 1995.

127. Simons, A.J.H. and Fernández-y-Fernández, C.A., Using Alloy to model-check visual design notations. In Proceedings of *Sixth Mexican International Conference on Computer Science*, (Puebla, México, 2005), IEEE Computer Society, 121-128.

128. Beeck, M.v.d. A Comparison of Statecharts Variants. *Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes In Computer Science, ed. G. Goos, J. Hartmanis and J. van Leeuwen*, *863*, 128-148, 1994.

129. Artiso, Visual Case Tool, *Artiso* 2006, http://www.visualcase.com/kbase/visual_case.htm, Last access: November 2006

130. Mosses, P.D., The Varieties of Programming Language Semantics and their Uses. In Proceedings of *Perspectives of System Informatics : 4th International Andrei Ershov Memorial Conference, PSI 2001*, (Akademgorodok, Novosibirsk, Russia, 2001), Springer-Verlag, 165-190.

131. 2U Consortium, Unambiguous UML (2U) 3rd Revised Submission to UML 2 Superstructure RFP, 2003, http://www.2uworks.org/uml2submission/super0.2/uml2SuperSubmission02.pdf, Last access: February 2005

132. Evans, A., Sammut, P., Willans, J.S., Moore, A. and Maskeri, G. A unified superstructure for UML. *Journal of Object Technology*, *4* (1), 165-181, 2005.

133. Baeten, J.C.M. and Weijland, W.P. *Process algebra*. Cambridge University Press, Cambridge; New York, 1990.

134. Glabbeek, R.J.v. The Linear Time - Branching Time Spectrum I. The semanticts of Concrete, Sequential Process. in *Handbook of Process Algebra*, 2001.

135. Schneider, S.A. *Concurrent and real time systems : the CSP approach*. John Wiley, Chichester [England] ; New York, 2000.

136. Mazurkiewicz, A. Introduction to Trace Theory. in Diekert, V. and Rozenberg, G. eds. *The Book of Traces*, World Scientific, London, 1995, 3-42.

137. Mazurkiewicz, A. Concurrent Program Schemes and Their Interpretation. DAIMI PB-78. *Aarhus University, Comp. Science Depart.*, 1977.

138. Mazurkiewicz, A. Trace theory. in *Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency*, Springer-Verlag New York, Inc., Bad Honnef, 1987, 279-324.

139. Scott, D.S. Domains for Denotational Semantics. in Nielsen, M. and Schmidt, E.M. eds. *Proceedings of the 9th Colloquium on Automata, Languages and Programming*, Springer-Verlag, 1982, 577-613.

140. Cardelli, L. and Wegner, P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, *17* (4), 471 - 523, 1985.

141. Manzano, M. *Extensions of first order logic*. Cambridge University Press, Cambridge ; New York, 1996.

142. Thompson, S. *Haskell : the craft of functional programming*. Addison Wesley, Harlow, Eng. ; Reading, Mass., 1999.

143. Aho, A.V., Sethi, R. and Ullman, J.D. *Compilers, principles, techniques, and tools*. Addison-Wesley Pub. Co., Reading, Mass., 1986.

144. Adams, M. A self resourcing web based electronic journal, Bachelor Thesis, *Computer Science*, The University of Sheffield, Sheffield, 2002, 69 pp.

145. Mäkelä, M. Parallel and Distributed Digital Systems Temporal Logic, *Helsinki University of Technology*, 2002.

146. Clarke, E.M., Emerson, E.A. and Sistla, A.P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, *8* (2), 244-263, 1986.

147. Hartel, P.H., A trace semantics for positive Core XPath. In Proceedings of *Temporal Representation and Reasoning, 2005. TIME 2005. 12th International Symposium on*, (2005), 103-112.

148. Kehris, E., Eleftherakis, G. and Kefalas, P. Using X-machines to model and test discrete event simulation programs. *Systems and Control: Theory and Applications*, 163–171, 2000.

149. Eleftherakis, G., Kefalas, P. and Sotiriadou, A. Xmctl: Extending temporal logic to facilitate formal verification of x-machines. *Matematica-Informatica, Analele Universitatii Bucharest*, *50*, 79-95, 2002.

150. Kefalas, P., Eleftherakis, G. and Sotiriadou, A., Developing tools for formal methods. In Proceedings of *Proceedings of the 9th Panhellenic Conference in Informatics*, (2003), 625–639.

151. Gurfinkel, A., Devereux, B. and Chechik, M., Model exploration with temporal logic query checking. In Proceedings of *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, (2002), ACM New York, NY, USA, 139-148.

152. Gurfinkel, A., Chechik, M. and Devereux, B., Temporal logic query checking: A tool for model exploration. In Proceedings of, (2003), Institute of Electrical and Electronics Engineers.

153. Henderson, P. System design: analysis. *Infotech State of the Art Report*, *Series 9* (6), 5-163, 1981.

154. Marlow, S. and Gill, A., Happy User Guide, 2001, http://www.haskell.org/happy/, Last access: 2006

Appendix A:
Proving Basic Properties

Chapter 7 described the soundness of the axioms for the task algebra illustrated in chapter 5. The trace semantics from chapter 6 and basic properties demonstrated here, were used to prove the soundness of the axioms. Appendix B will demonstrate the entire congruence properties for the axioms of the algebra.

A set of basic operations were used in chapter 7 and assumed as true at that moment. In this appendix, the proof for these properties is presented. The set of basic properties are as follows:

• A.1 Associativity of ⊗

• A.2 Distribution of ⊗ over union

• A.3 Identity for ⊗

• A.4 Associativity of //

• A.5 Commutativity of //

• A.6 Distribution of // over union

• A.7 Identity for //

• A.8 Distribution of unpack over union

A.1 Associativity of ⊗

For:

$seta \otimes setb = \{a \# b \mid a, b \in Trace, a \in seta, b \in setb\}$ (cp1)

We need to prove that:
$(\{Trace1\} \otimes \{Trace2\}) \otimes \{Trace3\} = \{Trace1\} \otimes (\{Trace2\} \otimes \{Trace3\})$

Lemma 1. It holds:

$(Trace1 \# Trace2) \# Trace3 = Trace1 \# (Trace2 \# Trace3)$

where *Trace1*, *Trace2* and *Trace3* are any trace.

Proof by induction.

The formulae depicted from 1 to 5 are the base cases of the formula to prove.

1. (<>#Trace2)#Trace3 = <>#(Trace2#Trace3)

Proof:

(<>#Trace2)#Trace3 = Trace2#Trace3 -- by (tc1)

and

<>#(Trace2#Trace3) = Trace2#Trace3 -- by (tc1)

2. (<σ>#Trace2)#Trace3 = <σ>#(Trace2#Trace3)

Proof:

(<σ>#Trace2)#Trace3 = <σ>#Trace3 -- by (tc2)

= <σ> -- by (tc2)

and

<σ>#(Trace2#Trace3) = <σ> -- by (tc2)

3. (<ϕ>#Trace2)#Trace3 = <ϕ>#(Trace2#Trace3)

Proof:

(<ϕ>#Trace2)#Trace3 = <ϕ>#Trace3 -- by (tc3)

= <ϕ> -- by (tc3)

and

<ϕ>#(Trace2#Trace3) = <ϕ> -- by (tc3)

4. (<↓>#Trace2)#Trace3 = <↓>#(Trace2#Trace3)

Case 4.a: Trace2= ↓.rest

The formula to prove in this case is as follows:

(<↓>#↓.rest)#Trace3 = <↓>#(↓.rest #Trace3)

Equivalent to:

a.1 (<↓>#↓.rest)#Trace3 = (↓.rest)#Trace3

and

a.2 <↓>#(↓.rest #Trace3) = (↓.rest)#Trace3

Proof:

4.a.1:

(<↓>#↓.rest)#Trace3 = (↓.rest)#Trace3 -- by (tc4)

4.a.2:

To prove: <↓>#(↓.rest #Trace3) = (↓.rest)#Trace3

We are going to prove by induction over the length of *rest*:

4.a.2.1 Base cases:

Zero length: rest = < > .

For this case the formula to prove is as follows:

<↓>#(↓.< >#Trace3) = (<↓>)#Trace3

Observe that: (↓.< >)#Trace3 = <↓ >#Trace3

Proof:

<↓>#(↓.< >#Trace3) = <↓>#(<↓ >#Trace3)

4.a.2.1.a) If Trace3 = ↓.Trace4, then

<↓>#(<↓ >#Trace3)= <↓>#(<↓ >#(↓.Trace4))

= <↓>#(Trace3) .

4.a.2.1.b) If a ≠↓ and Trace3 = a.Trace4, then

<↓>#(↓.< >#Trace3) =<↓>#(<↓ >#a.Trace4) = <↓>#(↓. a.Trace4)

=↓. a.Trace4 = <↓>#(Trace3).

4.a.2.2 For base cases of length 1:

Case 4.a.2.2.a : *s* is an identifier.

Formula to prove:

<↓>#(↓.<s> #Trace3) = (↓.<s>)#Trace3

Proof:

<↓>#(↓.< s>#Trace3) = <↓>#(↓.< s> # Trace3)

$=\langle\downarrow\rangle\#(\downarrow.(\langle s\rangle \# Trace3))$ -- by (tc6)

$=\downarrow.(\langle s\rangle \# Trace3)$ -- by (tc4)

In addition,

$(\downarrow.\langle s\rangle)\#Trace3 = \downarrow.(\langle s\rangle \# Trace3)$ -- by (tc6)

Case 4.a.2.2.b : $s = \sigma$.

Formula to prove:

$$\langle\downarrow\rangle\#(\downarrow.\langle\sigma\rangle \#Trace3) = (\downarrow.\langle\sigma\rangle)\#Trace3$$

Proof:

$\langle\downarrow\rangle\#(\downarrow.\langle\sigma\rangle\#Trace3) = \langle\downarrow\rangle\#(\downarrow.(\langle\sigma\rangle \# Trace3))$ -- by (tc6)

$= \downarrow.(\langle\sigma\rangle \# Trace3)$ -- by (tc4)

In addition,

$(\downarrow.\langle\sigma\rangle)\#Trace3 = \downarrow.(\langle\sigma\rangle \# Trace3)$ -- by (tc6)

Case 4.a.2.2.c : $s = \phi$.

Formula to prove:

$$\langle\downarrow\rangle\#(\downarrow.\langle\phi\rangle \#Trace3) = (\downarrow.\langle\phi\rangle)\#Trace3$$

Proof:

$\langle\downarrow\rangle\#(\downarrow.\langle\phi\rangle\#Trace3) = \langle\downarrow\rangle\#(\downarrow.(\langle \phi\rangle \# Trace3))$ -- by (tc6)

$= \downarrow.(\langle\phi\rangle \# Trace3)$ -- by (tc4)

In addition,

$(\downarrow.\langle\phi\rangle)\#Trace3 = \downarrow.(\langle\phi\rangle \# Trace3))$ -- by (tc6)

Case 4.a.2.2.d : $s = \downarrow$

Formula to prove:

$$\langle\downarrow\rangle\#(\downarrow.\langle\downarrow\rangle \#Trace3) = (\downarrow.\langle\downarrow\rangle)\#Trace3$$

Proof:

$\langle\downarrow\rangle\#(\downarrow.\langle\downarrow\rangle\#Trace3) = \langle\downarrow\rangle\#(\downarrow.(\langle\downarrow\rangle\#Trace3))$ -- by (tc6)

$= \downarrow.(\langle\downarrow\rangle\#Trace3)$ -- by (tc4)

4.a.2.3 Induction hypothesis. Supposing it holds for traces in *rest* of until *k* symbols, $k \geq 1$:

$<\downarrow>\#(\downarrow.rest\ \#Trace3) = (\downarrow.rest)\#Trace3$

4.a.2.4 Induction step. Let *rest=s*. *rest1*, where *rest1* is a trace of *k* symbols and *s* is an identifier.

Fot other cases in *s*, *rest* is a trace of length 1, $<\sigma>$ or $<\phi>$. $\downarrow.\downarrow$ is transformed in $\downarrow$ and the formula holds by the induction hypothesis.

Case 4.a.2.4.a : *s* is an identifier.

$<\downarrow>\#(\downarrow.rest\ \#Trace3) = <\downarrow>\#(\downarrow.s.rest1\ \#Trace3)$

$=<\downarrow>\#(\downarrow.(s.rest1\ \#Trace3))$ -- by (tc6)

$= \downarrow.(s.rest1\ \#Trace3)$ -- by (tc4)

$= \downarrow.(rest\ \#Trace3)$

In addition,

$(\downarrow.rest)\#Trace3 = (\downarrow.s.rest1)\#Trace3 = \downarrow.(s.rest1\#Trace3)$ -- by (tc6)

Case 4b: *Trace2= a.rest* where *a* is different of $\downarrow$, σ and ϕ.

$a \neq \sigma$ y $a \neq \phi$ can be assumed since these cases are reduced to length 1 already proved.

The formula to prove is:

$(<\downarrow>\#a.rest)\#Trace3 = <\downarrow>\#(\ a.rest\ \#Trace3)$

Proof:

$(<\downarrow>\#a.rest)\#Trace3 = (\downarrow.a.rest)\#Trace3$ -- by (tc5)

$= \downarrow.(a.rest\#Trace3)$ -- by (tc6)

$= <\downarrow>\#(a.rest\#Trace3)$ -- by (tc6)

5. (<a>#Trace2)#Trace3 = <a>#(Trace2#Trace3) where *a* is an identifier (different to σ, ϕ and $\downarrow$)

Proof:

$(<a>\#Trace2)\#Trace3 = (a.<>\#Trace2)\#Trace3$ -- by *cons* operator

$= (a.(<>\#Trace2))\#Trace3$ -- by (tc6)

= (a.Trace2)#Trace3 -- by (tc1)

= a.(Trace2#Trace3) -- by (tc6)

and

<a>#(Trace2#Trace3) = a.<>#(Trace2#Trace3) -- by *cons* operator

= a.(<>#(Trace2#Trace3)) -- by (tc6)

= a.(Trace2#Trace3) -- by (tc1)

6. **Induction hypothesis.** Assuming the formula

(Trace1#Trace2) #Trace3 = Trace1#(Trace2#Trace3) , holds for every trace *Trace2*, *Trace3*, if *Trace1* is no longer than *k*, where $k \geq 1$.

7. **Induction step.** Assuming that *Trace1=a.Trace*, where *Trace* is a trace of length *k* and *a* is an identifier or *a*=↓. For other cases, the trace has a length of 1.

Case 7.a: *a* is an identifier.

(a.Trace #Trace2) #Trace3 = (a.(Trace #Trace2)) #Trace3 -- by (tc6)

= a.((Trace #Trace2) #Trace3) -- by (tc6)

In addition,

a.Trace #(Trace2#Trace3) = a.(Trace #(Trace2#Trace3)) -- by (tc6)

= a.((Trace #Trace2)#Trace3) -- by 6

Case 7.b: a =↓

(↓.Trace #Trace2) #Trace3 = (↓.(Trace #Trace2)) #Trace3 -- by (tc6)

= ↓.((Trace #Trace2) #Trace3) -- by (tc6)

In addition,

↓.Trace #(Trace2#Trace3) =↓.(Trace #(Trace2#Trace3)) -- by (tc6)

=↓.((Trace #Trace2)#Trace3) -- by 6

Theorem. Let *{Trace1}*, *{Trace2}* and *{Trace3}* be three non-empty sets of traces. Then it holds
({Trace1}⊗{Trace2})⊗{Trace3}={Trace1}⊗({Trace2}⊗{Trace3})

Proof:
Let *c* be an element of *({Trace1}⊗{Trace2})⊗{Trace3}*. Then *c=Trace#Trace3'*,

where $Trace \in (\{Trace1\} \otimes \{Trace2\})$ and $Trace3' \in \{Trace3\}$.
$Trace \in (\{Trace1\} \otimes \{Trace2\})$, implies that $Trace = Trace1'\#Trace2'$, for some $Trace1' \in \{Trace1\}$ and some $Trace2' \in \{Trace2\}$. In consequence, $c = (Trace1'\#Trace2')\#Trace3'$. By lemma 1, $(Trace1'\#Trace2')\#Trace3' = Trace1'\#(Trace2'\#Trace3')$. Then, $c = Trace1'\#(Trace2'\#Trace3')$. This implies $c \in \{Trace1\} \otimes (\{Trace2\} \otimes \{Trace3\})$.

Conversely, let c be an element of $\{Trace1\} \otimes (\{Trace2\} \otimes \{Trace3\})$. Then, there is a trace $Trace1'$ of $\{Trace1\}$ and a trace $Trace$ of $(\{Trace2\} \otimes \{Trace3\})$, such that $c = Trace1'\#Trace$. $Trace$ has the form $Trace2'\#Trace3'$, where $Trace2'$ belongs to $\{Trace2\}$ and $Trace3'$ belongs to $\{Trace3\}$. Then, $c = Trace1'\#(Trace2'\#Trace3')$. By lemma 1, $Trace1'\#(Trace2'\#Trace3') = (Trace1'\#Trace2')\#Trace3'$. Then, $c = (Trace1'\#Trace2')\#Trace3'$. This implies $c \in (\{Trace1\} \otimes \{Trace2\}) \otimes \{Trace3\}$.

This means that both sets are equals.

A.2 Distribution of ⊗ over union

For:

$seta \otimes setb = \{a \# b \mid a, b \in Trace, a \in seta, b \in setb\}$ (cp1)

We need to prove that:

$$(\{Trace1\} \cup \{Trace2\}) \otimes \{Trace3\} = (\{Trace1\} \otimes (\{Trace3\}) \cup (\{Trace2\} \otimes \{Trace3\})$$

Proof.

Let c be an arbitrary element of $(\{Trace1\} \cup \{Trace2\} \otimes \{Trace3\}$, then $c = a\#b$, for some $a \in (\{Trace1\} \cup \{Trace2\})$ and for some $b \in \{Trace3\}$.

Since a is an element of the union of two sets, then $a \in \{Trace1\}$ or $a \in \{Trace2\}$ holds. If $a \in \{Trace1\}$, then $a\#b \in \{Trace1\} \otimes \{Trace3\}$ holds, by definition.

If $a \in \{Trace2\}$, then $a\#b \in \{Trace2\} \otimes \{Trace3\}$, by definition. Thus, $a\#b \in (\{Trace1\} \otimes (\{Trace3\}) \cup (\{Trace2\} \otimes \{Trace3\})$ is true.

It follows that $c \in (\{Trace1\} \otimes \{Trace3\}) \cup (\{Trace2\} \otimes \{Trace3\})$, as desired.

Conversely, let c be an element of $(\{Trace1\} \otimes \{Trace3\}) \cup (\{Trace2\} \otimes \{Trace3)\}$.

In this case, $c \in \{Trace1\} \otimes \{Trace3\}$ or $c \in \{Trace2\} \otimes \{Trace3\}$.

If $c \in \{Trace1\} \otimes \{Trace3\}$ holds. Then, c is of the form $a\#b$, for some $a \in \{Trace1\}$ and for some $b \in \{Trace3\}$. This implies, $a \in (\{Trace1\} \cup \{Trace2\})$, $b \in \{Trace3\}$ and, by definition, $a\#b \in (\{Trace1\} \cup \{Trace2\}) \otimes \{Trace3\}$. Then, $c \in (\{Trace1\} \cup \{Trace2\}) \otimes \{Trace3\}$, as desired.

In the second case, $c \in \{Trace2\} \otimes \{Trace3\}$ holds, the proof is similar.

In both cases, $c \in (\{Trace1\} \otimes \{Trace3\}) \cup (\{Trace2\} \otimes \{Trace3)\}$ implies $c \in (\{Trace1\} \cup \{Trace2\}) \otimes \{Trace3\}$.

A.3 Identity for ⊗

For:

$seta \otimes setb = \{a \# b \mid a, b \in Trace, a \in seta, b \in setb\}$ (cp1)

We need to prove that:

$\{Trace\} \otimes \{<>\} = \{Trace\}$

and

$\{<>\} \otimes \{Trace\} = \{Trace\}$

Proof.

For the first equality, let c be an arbitrary element of $\{Trace\} \otimes \{<>\}$, then $c = Trace1 \# <>$, where $Trace1 \in \{Trace\}$ and $<> \in \{<>\}$, by properties (tc6, tc1 and *cons* operator) $Trace1 \# <> = Trace1$. Therefore, $c = Trace1$ and then $c \in \{Trace\}$.

Reciprocally, if $c \in \{Trace\}$, then $c = Trace1$, where $Trace1 \in \{Trace\}$. However, by (tc6, tc1 and *cons* operator) $Trace1 = Trace1 \# <>$, consequently $Trace1 \in \{Trace\} \otimes \{<>\}$. Therefore, $c \in \{Trace\} \otimes \{<>\}$.

Therefore, we have proved that all the elements in the set $\{Trace\} \otimes \{<>\}$ are elements in the set $\{Trace\}$. In addition, that all the elements in the set $\{Trace\}$ are elements in the set $\{Trace\} \otimes \{<>\}$. This implies the equality of both sets.

For the second equality, let c be and arbitrary element of $\{<>\} \otimes \{Trace\}$, then $c = <> \# Trace1$, where $Trace1 \in \{Trace\}$ and $<> \in \{<>\}$, by (tc1) $<> \# Trace1 = Trace1$. Therefore, $c = Trace1$ and then $c \in \{Trace\}$.

Reciprocally, if $c \in \{Trace\}$, then $c = Trace1$, where $Trace1 \in \{Trace\}$. However, by (tc1) $Trace1 = <> \# Trace1$, consequently $Trace1 \in \{<>\} \otimes \{Trace\}$. Therefore, $c \in \{<>\} \otimes \{Trace\}$.

Therefore, we have proved that all the elements in the set $\{<>\} \otimes \{Trace\}$ are elements in the set $\{Trace\}$. In addition, that all the elements in the set $\{Trace\}$ are elements in the set $\{<>\} \otimes \{Trace\}$. This implies the equality of both sets.

A.4 Associativity of //

In order to prove the associativity of //, we need to calculate the image of the function ~ over all the possible pair of traces. However, note that we only need to know the image of pairs taken from the set $\{<>, <\sigma>, <\phi>, <\downarrow>\}$ and of a particular set of pairs of traces , the called basic traces (Definition 3), to characterise the general behaviour of ~ . Moreover, we claim that it is enough to prove the associativity of // for sets with only one element. This follows as a consequence of

Proposition 1. In this section, we only will show the proof for sets containing one element, to give the idea of our approach.

Definition 1. We will call a' a symbol, if $a' = a$ or $a' = a, \downarrow$ holds, where a is an identifier .

Definition 2 . Let *Trace1*, *Trace2* and *Trace3* be three traces. *Trace2* is a truncation of *Trace1*, if and only if there exists a trace *Trace3*, such that
Trace1 = *Trace2* # *Trace3*.

Definition 3. A trace *Trace* will be called a trace in the basic form or , in short, a basic trace , if *Trace* has one of these forms:

a) $Trace = < a_1', ..., a_k' >$, where $a_i' = \begin{cases} a_i \\ or \\ a_i, \downarrow \end{cases}$, for $1 \le i \le k$ and $a_1, ..., a_k$ are identifiers, k is an integer greater or equal than 1.

b) $Trace = < a_1', ..., a_k', \sigma >$, where $a_i' = \begin{cases} a_i \\ or \\ a_i, \downarrow \end{cases}$, for $1 \le i \le k$ and $a_1, ..., a_k$ are identifiers, k is an integer greater or equal than 1.

c) $Trace = < a_1', ..., a_k', \phi >$, where $a_i' = \begin{cases} a_i \\ or \\ a_i, \downarrow \end{cases}$, for $1 \le i \le k$ and $a_1, ..., a_k$ are identifiers, k is an integer greater or equal than 1.

Notation 1.

1. If we have $< b_1', ..., b_k' >$, then its truncation $< b_1', ..., b_{k-1}' >$ is equal to $< >$, if $k = 1$.
2. $\bigcup_{i=1}^{k-1} A_i = \varnothing$, if $k = 1$ and $\{A_i\}_{i=1}^{m}$ is a collection of sets, $m \ge 1$.

Notation 2.
To avoid confusions we will change our notation:
A will denote {Trace1}, B, {Trace2} and C, {Trace3}.

We want to prove:

(A // B) // C = A // (B // C), where A, B and C are non-empty sets consisting of traces.

Proposition 1. Let A, B and C be non-empty sets of traces. Then it holds:

a) $A // B = \bigcup_{(a,b) \in A \times B} a \sim b$

b) If a and b are two arbitrary traces, then $a \sim b = \{a\} // \{b\}$.

c) $A // B = \bigcup_{(a,b) \in A \times B} (\{a\} // \{b\})$

d) If $\{A_i\}_{i \in I}$ is a non-empty collection of non-empty sets of traces, then $\left(\bigcup_{i \in I} A_i\right) // B = \bigcup_{i \in I} (A_i // B)$.

e) If $\{B_i\}_{i \in I}$ is a non-empty collection of non-empty sets of traces, then $(A) // \bigcup_{i \in I} B_i = \bigcup_{i \in I} (A // B_i)$.

f) Then it holds

$$(A // B) // C = \bigcup_{((a,b),c) \in (A \times B) \times C} ((\{a\} // \{b\}) // \{c\})$$
$$= \bigcup_{(a,b,c) \in A \times B \times C} ((a \sim b) // \{c\})$$

And

$$A // (B // C) = \bigcup_{(a,(b,c)) \in A \times B \times C} (\{a\} // (\{b\} // \{c\}))$$
$$= \bigcup_{(a,b,c) \in A \times B \times C} (\{a\} // (b \sim c))$$

g) $\{<\phi>\} // B = B // \{<\phi>\} = \{<\phi>\}$, if B doesn't contain the trace $<\sigma>$.

h) $\{<\sigma>\} // B = B // \{<\sigma>\} = \{<\sigma>\}$.

i) If B is a set of traces of the form $\downarrow.rest$, where *rest* is a trace. Then $\{\downarrow\} \otimes B = B$.

j) If for each arbitrary chosen traces a, b c, $(a \sim b) // \{c\} = \{a\} // (b \sim c)$ holds, then $(A // B) // C = A // (B // C)$ holds for each non-empty sets of traces A, B and C.

k) Let *Trace1* and *Trace2* be two traces, a and b two identifiers. Then

$k.1$ $(<a,\downarrow> \# Trace1) \sim (<b> \# Trace2) =$

$$= \{<a,\downarrow>\} \otimes (Trace1 \sim (<b> \# Trace2)) \cup \{<b>\} \otimes ((<a,\downarrow> \# Trace1) \sim Trace2)$$

$k.2$ $(<a> \# Trace1) \sim (<b,\downarrow> \# Trace2) =$

$$= (\{<a>\} \otimes (Trace1 \sim (<b,\downarrow> \# Trace2))) \cup (\{<b,\downarrow>\} \otimes ((<a> \# Trace1) \sim Trace2))$$

$k.3 \quad (<a,\downarrow> \# Trace1) \sim (<b,\downarrow> \# Trace2) =$

$= (\{<a,\downarrow>\} \otimes (Trace1 \sim (<b,\downarrow> \# Trace2))) \cup (\{<b,\downarrow>\} \otimes ((<a,\downarrow> \# Trace1) \sim Trace2))$

These statements show that symbols behaviour like identifiers. For this reason we will use the following generalisation of (ti9):

a.as ~ b.bs = ({<a>})⊗(as ~b.bs) ∪ ({<b>}⊗(bs ~ a.as))

where *a* and *b* are symbols.

During this section we will mention this generalisation as (ti9) too.

l) Let *Trace1*, *Trace2* and *Trace3* be traces. Suppose that $(Trace1 \sim Trace2) // \{Trace3\} = \{Trace1\} // (Trace2 \sim Trace3)$ holds.

Thus, the following relations holds:

1. $((<\downarrow> \# Trace1) \sim Trace2) // \{Trace3\} = \{<\downarrow> \# Trace1\} // (Trace2 \sim Trace3)$

2. $(Trace1 \sim (<\downarrow> \# Trace2)) // \{Trace3\} = \{Trace1\} // ((<\downarrow> \# Trace2) \sim Trace3)$

3. $(Trace1 \sim Trace2) // \{<\downarrow> \# Trace3\} = \{Trace1\} // (Trace2 \sim (<\downarrow> \# Trace3))$

4. $((<\downarrow> \# Trace1) \sim (<\downarrow> \# Trace2)) // \{Trace3\} = \{<\downarrow> \# Trace1\} // ((<\downarrow> \# Trace2) \sim Trace3)$

5. $((<\downarrow> \# Trace1) \sim Trace2) // \{<\downarrow> \# Trace3\} = \{<\downarrow> \# Trace1\} // (Trace2 \sim (<\downarrow> \# Trace3))$

6. $((<\downarrow> \# Trace1) \sim Trace2) // \{<\downarrow> \# Trace3\} =$
   $= \{<\downarrow> \# Trace1\} // (Trace2 \sim (<\downarrow> \# Trace3))$

7. $((<\downarrow> \# Trace1) \sim (<\downarrow> \# Trace2)) // \{<\downarrow> \# Trace3\} =$
   $= \{<\downarrow> \# Trace1\} // ((<\downarrow> \# Trace2) \sim (<\downarrow> \# Trace3))$

Proof.

a) It is straightforward from the definitions.

b) It is straightforward from the definitions.

c) It follows by b) and by the definition of //.

d) If $\{A_i\}_{i\in I}$ is a non-empty collection of non-empty sets of traces. Then, by c),

$$\left(\bigcup_{i\in I} A_i\right)//B = \bigcup_{\substack{a\in\bigcup_{i\in I} A_i \\ b\in B}} (\{a\}//\{b\}) = \bigcup_{i\in I}\bigcup_{a\in A_i}\bigcup_{b\in B}(\{a\}//\{b\}) = \bigcup_{i\in I}\bigcup_{\substack{a\in A_i \\ b\in B}}(\{a\}//\{b\})$$
$$= \bigcup_{i\in I}(A_i//B) \quad .$$

e) This proof is similar to the previous.

f) From the previous properties and because there exits a bijection between the sets $(A\times B)\times C$ and $A\times B\times C$ it follows:

$$(A//B)//C = \left(\bigcup_{\substack{a\in A \\ b\in B}}(\{a\}//\{b\})\right)//C = \bigcup_{c\in C}\left(\left(\bigcup_{\substack{a\in A \\ b\in B}}(\{a\}//\{b\})\right)//\{c\}\right)$$
$$= \bigcup_{c\in C}\left(\bigcup_{(a,b)\in A\times B}((\{a\}//\{b\})//\{c\})\right) = \bigcup_{((a,b,),c)\in(A\times B)\times C}((\{a\}//\{b\})//\{c\})$$
$$= \bigcup_{(a,b,c)\in A\times B\times C}((a\sim b)//\{c\}) \quad .$$

The other equality is proven similarly.

g) It follows from the definitions and from (ti5) and (ti6).

h) It follows from (ti3), (ti4) and b).

i) It is consequence of (tc4).

j) It follows straightforward from b), f) and the hypothesis.

k) It follows from (ti9), (ti7) and (ti8).

l) Without lost of generality we can suppose that the first element of the trace (from left to right) is not $\downarrow$. We will prove 1 and 4. The other proofs are similar.

Proof of 1.

By (ti7) and the hypothesis,

$$\left(\left(<\downarrow> \# Trace1\right) \sim Trace2\right) / / \{Trace3\} = \left(\{<\downarrow>\} \otimes \left(Trace1 \sim Trace2\right)\right) / / \{Trace3\} =$$
$$= \{<\downarrow>\} \otimes \left(\left(Trace1 \sim Trace2\right) / / \{Trace3\}\right) = \{<\downarrow>\} \otimes \left(\{Trace1\} / / \left(Trace2 \sim Trace3\right)\right).$$

Thus,

$$\left(\left(<\downarrow> \# Trace1\right) \sim Trace2\right) / / \{Trace3\} = \{<\downarrow>\} \otimes \left(\{Trace1\} / / \left(Trace2 \sim Trace3\right)\right).$$

On the other hand, using (ti7) and the hypothesis:

$$\left(\left(<\downarrow> \# Trace1\right) \sim Trace2\right) / / \{Trace3\} = \left(\{<\downarrow>\} \otimes \left(Trace1 \sim Trace2\right)\right) / / \{Trace3\}$$
$$= \{<\downarrow>\} \otimes \left(\left(Trace1 \sim Trace2\right) / / Trace3\right) = \{<\downarrow>\} \otimes \left(\{Trace1\} / / \left(Trace2 \sim Trace3\right)\right)$$

Therefore, both members of the equality to prove are the same set.

Proof of 4.

By (ti7) the hypothesis and (ti8) we have:

$$\left(\left(<\downarrow> \# Trace1\right) \sim \left(<\downarrow> \# Trace2\right)\right) / / \{Trace3\} = \left(\{<\downarrow>\} \otimes \left(Trace1 \sim Trace2\right)\right) / / \{Trace3\}$$
$$= \{<\downarrow>\} \otimes \left(\left(Trace1 \sim Trace2\right) / / \{Trace3\}\right)$$
$$= \{<\downarrow>\} \otimes \left(\{Trace1\} / / \left(Trace2 \sim Trace3\right)\right)$$
$$= \left(<\downarrow> \# Trace1\right) / / \left(\left(Trace2\right) \sim Trace3\right)$$
$$= \left(<\downarrow> \# Trace1\right) / / \{<\downarrow>\} \otimes \left(\left(Trace2\right) \sim Trace3\right)$$
$$= \left(<\downarrow> \# Trace1\right) / / \left(\left(<\downarrow> \# Trace2\right) \sim Trace3\right)$$

Remember : $<\downarrow> \# <\downarrow> = <\downarrow>$.

A.5 Commutativity of //

For:

seta // setb = ∪{ a ~ b | a, b ∈ Trace, a ∈ seta, b ∈ setb } (di1)

We need to prove that:

{Trace1} // {Trace2} = {Trace2} // {Trace1}

Lemma 1. It holds:

Trace1~Trace2 = Trace2~ Trace1

where *Trace1* and *Trace2* are any trace

Proof by cases.

Case 1. <>~Trace2 = Trace2~<> -- by (ti1) and (ti2).

Case 2. <σ>~Trace2 = Trace2~<σ> -- by (ti3) and (ti4).

Case 3. <ϕ>~Trace2 = <ϕ>~Trace2 -- by (ti5) and (ti6).

Case 4. <↓>~Trace2 = Trace2~<↓>

We just need to prove the equality on the next case. The other cases have been proved in cases 1-3.

Trace2 = a.rest, where *a* is different of σ and ϕ.

$a \neq \sigma$ y $a \neq \phi$ can be assumed since these cases are reduced to length 1 already proved.

The formula to prove is:

<↓>~a.rest = a.rest ~<↓>

Proof:

<↓>~a.rest = ↓.<> ~ a.rest -- by *cons* operator

= {<↓>} ⊗ (<> ~ a.rest) -- by (ti7)

= {<↓>} ⊗ {a.rest} -- by (ti1)

In addition,

a.rest ~<↓> = a.rest ~↓.<> -- by *cons* operator

= {<↓>} ⊗ (a.rest ~<>) -- by (ti8)

= {<↓>} ⊗ {a.rest} -- by (ti2)

Case 5. <a>~Trace2 = Trace2~<a> where *a* is an identifier (different to σ, ϕ and ↓).

Only two cases remain to be proved. The other cases are already proved in cases 1-4.

Case 5.a: *Trace2=b.rest* where *b* is different of σ, ↓ and ϕ.

Proof:

<a>~b.rest = a.<> ~ b.rest -- by *cons* operator

= ({<a>} ⊗ (<> ~ b.rest)) ∪ ({<b>} ⊗ (rest ~ a.<>)) -- by (ti9)

In addition,

b.rest~<a> = b.rest ~ a.<> -- by *cons* operator

= ({<b>} ⊗ (rest ~ a.<>)) ∪ ({<a>} ⊗ (<> ~ b.rest)) -- by (ti9)

= ({<a>} ⊗ (<> ~ b.rest)) ∪ ({<b>} ⊗ (rest ~ a.<>))

-- by commutativity of the union of sets.

Case 5.b: *Trace2=↓.rest* .

Proof:

<a>~↓.rest = {< ↓>}⊗ *(rest ~<a>)* -- by (ti8).

In addition,

↓.rest ~<a> = {< ↓>}⊗ *(rest ~<a>)* -- by (ti7).

Cases 6. Assuming that *Trace1=a.Trace*, where *Trace* is a trace of length *k, k ≥1,* and *a* is an identifier or *a*=↓. For other cases, the trace has a length of 1, and these cases have been already proved.

Case 6.a: *a* is an identifier.

Case 6.a.a: *Trace2=b.rest*, where *rest* is a trace of length *k* and *b* is an identifier or *b*=↓.

Case 6.a.a.a: *b* is an identifier

a.Trace ~b.rest

= ({<a>} ⊗ (Trace ~ b.rest)) ∪ ({<b>} ⊗ (rest ~ a.Trace))

-- by (ti9)

In addition,

b.rest ~ a.Trace

= ({<b>} ⊗ (rest ~ a.Trace)) ∪ ({<a>} ⊗ (Trace ~ b.rest))

-- by (ti9)

= ({<a>} ⊗ (Trace ~ b.rest)) ∪ ({<b>} ⊗ (rest ~ a.Trace))

-- by commutativity of the union of sets.

Case 6.a.a.b: $b=\downarrow$

a.Trace ~ ↓.rest

= {<↓>} ⊗ (rest ~ a.Trace) -- by (ti8)

In addition,

↓.rest ~ a.Trace

= {<↓>} ⊗ (rest ~ a.Trace) -- by (ti7)

Case 6.b: a =↓

↓.Trace ~Trace2 = {<↓>} ⊗ (Trace ~Trace2) -- by (ti7)

In addition,

Trace2 ~↓.Trace = {<↓>} ⊗ (Trace ~Trace2) -- by (ti8)

Theorem. Let *{Trace1}* and *{Trace2}* be two non-empty sets of traces. Then it holds *{Trace1}//{Trace2}={Trace2}//{Trace1}*

Proof:
Let *c* be an arbitrary element of *{Trace1}//{Trace2}*. Then *c∈Trace1'~Trace2'*, where Trace1' *∈{Trace1}* and Trace2' *∈{Trace2}*. By lemma 1, we know *Trace1'~Trace2'=Trace2'~Trace1'*. Then, c *∈ Trace2'~Trace1'*. This set is a subset of *{Trace2}//{Trace1}*, by definition.

Conversely, let c be an element of *{Trace2}//{Trace1}*. Then, there is a trace *Trace2'* of *{Trace2}* and a trace *Trace1'* of *{Trace1}* such that *c∈Trace2'~Trace1'*. By lemmas 1, *Trace2'~Trace1'=Trace1'~Trace2'*. Then, *c∈Trace1'~Trace2'*. This implies *c∈{Trace1}//{Trace2}*, by definition.

This means that both sets are equals.

A.6 Distribution of // over union

For:

seta // setb = ∪{ a ~ b | a, b ∈ Trace, a ∈ seta, b ∈ setb } (di1)

We need to prove that:

({Trace1}∪{Trace2})//{Trace3}
= ({Trace1}//({Trace3})∪({Trace2}// {Trace3})

Proof.

Let *c* be an arbitrary element of *({Trace1}∪{Trace2})//{Trace3},* then *c=a~b*, for some *a∈({Trace1}∪{Trace2})* and for some *b∈{Trace3}*.

Since *a* is an element of the union of two sets, then *a∈{Trace1}* or *a∈{Trace2}* holds. If *a∈{Trace1}*, then *a~b∈{Trace1}//{Trace3}* holds, by definition.

If *a∈{Trace2}*, then *a~b∈{Trace2}//{Trace3}*, by definition. Thus, *a~b∈({Trace1}//({Trace3})∪({Trace2}//{Trace3})* is true.

It follows that *c∈({Trace1}//{Trace3})∪({Trace2}//{Trace3})*, as desired.

Conversely, let *c* be an element of *({Trace1}//({Trace3})∪({Trace2}// {Trace3}).*

In this case, *c∈({Trace1}//({Trace3})* or *c∈({Trace2}// {Trace3})*.

If *c∈({Trace1}//({Trace3})*. Then *c* is of the form *a~b*, for some *a∈{Trace1}* and for some *b∈{Trace3}*. This implies, *a∈({Trace1}∪{Trace2}), b∈{Trace3}* and, by definition, *a~b∈({Trace1}∪{Trace2})//{Trace3}*. Then, *c∈({Trace1}∪{Trace2})//{Trace3}*, as desired.

In the second case, *c∈{Trace2}//{Trace3}* holds, the proof is similar.

In both cases, *c∈({Trace1}//{Trace3})∪({Trace2}//{Trace3)}* implies *c∈({Trace1}∪{Trace2})//{Trace3}*.

A.7 Identity for //

For:

seta // setb = ∪{ a ~ b | a, b ∈ Trace, a ∈ seta, b ∈ setb } (di1)

We need to prove that:

{Trace}//{<>}={Trace}

Proof.

Let *c* be an arbitrary element of *{Trace}//{<>}*, then *c∈Trace1~<>*, where *Trace1∈{Trace}* and *<>∈{<>}*, by property (ti2) *Trace1~<>={Trace1}*. Therefore *c∈{Trace1},* because *Trace1∈{Trace}* then *{Trace1}* ⊂{Trace}. Thus *c∈{Trace}.*

Reciprocally, if *c∈{Trace}*, then *c=Trace1*, where *Trace1∈{Trace}*. However, by (ti2) *{Trace1}=Trace1~<>;* consequently, c*∈{Trace1},* then *c∈Trace1~<>*. Because by definition *Trace1~<>*is a subset of *{Trace}//{<>}, w*e can conclude that *c∈{Trace}//{<>}*.

We have proved that the elements of *{Trace}//{<>}* are in *{Trace}* and vice versa.

A.8 Distribution of unpack *over union*

For:

unpack seta = { lift a | a ∈ Trace, a ∈ seta } (up1)

We need to prove that:

unpack({Trace1}∪{Trace2 }) =unpack({Trace1})∪ unpack({Trace2})

Proof.

Let *b* be an arbitrary element of *unpack({Trace1}∪{Trace2}),* then *b* is of the form of *lift a*, where *a∈{Trace1}∪{Trace2}*.

Since *a* is part of the union of two set of traces, then *a∈{Trace1}* or *a∈{Trace2}* holds. If *a∈{Trace1}*, then *lift a∈ unpack{Trace1}* holds, by definition.

If *a∈{Trace2}*, then *lift a∈ unpack{Trace2}*, by definition. Thus

lift a∈ unpack{Trace1}∪unpack{Trace2} is true.

It follows that *b∈ unpack {Trace1} ∪unpack {Trace2}*, as desired.

Reciprocally, let *b* be an element of *unpack{Trace1}∪unpack{Trace2}*.
In this case, *b∈ unpack{Trace1}* or *b∈ unpack{Trace2}*.
If *b∈ unpack{Trace1}* holds. Then, *b* is of the form *lift a*, for some *a∈{Trace1}*.
This implies, *a∈ {Trace1}* and, by definition, *lift a∈ unpack({Trace1}∪{Trace2})*.
Then, *b∈ unpack({Trace1}{Trace2})*, as desired.

In the second case, *b∈ unpack{Trace2}*, the proof is similar.

In both cases, *b∈ unpack({Trace1}∪{Trace2 })* implies
b∈ unpack({Trace1})∪unpack({Trace2}).

Appendix B:
Congruence for the Semantics of Tasks

The chapter 7 described the soundness of the axioms for the task algebra illustrated in the chapter 5. The trace semantics from chapter 6 and basic properties explained in Appendix A were used to prove the soundness of the axioms. In the present appendix, the entire congruence properties are demonstrated for axioms of the algebra.

B.1 Introduction

Informally, congruence in an algebra can be checked by taking equivalent expressions and adding a subexpression to each of the equivalences. The result has to be the same if the expressions are congruent.

B.2 Showing congruence for sequential composition

Congruence for sequential composition is depicted in this section for the axioms of associative sequence, right distributivity of sequence over selection, empty sequence, fail on sequence, and succeed on sequence. Every axiom is represented in combination with one of the basic operators defined for the task algebra.

B.2.1 Showing congruence for basic operators in the associative sequence axiom

In this section, the congruence for the associative sequence axiom (s.1) is demonstrated for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.2.1.1 Congruence in s.1 with the sequence operator

If $\forall a, b, c \in Activity \bullet [\![a; (b; c)]\!] \equiv [\![(a; b); c]\!]$, **then**

$\forall a, b, c, d \in Activity \bullet [\![(a; (b; c)); d]\!] \equiv [\![((a; b); c); d]\!]$

$\Rightarrow [\![a; (b; c)]\!] \otimes [\![d]\!] \equiv [\![(a; b); c]\!] \otimes [\![d]\!]$ -- by ts1

$\Rightarrow$⟦a⟧ ⊗ ⟦(b; c)⟧ ⊗ ⟦d⟧ ≡ ⟦(a; b)⟧ ⊗ ⟦c⟧ ⊗ ⟦d⟧ -- by ts1

$\Rightarrow$⟦a⟧ ⊗ ⟦b⟧ ⊗ ⟦c⟧ ⊗ ⟦d⟧ ≡ ⟦a⟧ ⊗ ⟦b⟧ ⊗ ⟦c⟧ ⊗ ⟦d⟧ -- by ts1

B.2.1.2 Congruence in s.1 with the selection operator

If $\forall a, b, c \in Activity \bullet$ ⟦a; (b; c)⟧ ≡ ⟦ (a; b); c ⟧, **then**

$\forall a, b, c, d \in Activity \bullet$ ⟦(a; (b; c)) + d⟧ ≡ ⟦ ((a; b); c) + d⟧

$\Rightarrow$ {<↓>} ⊗ (⟦(a; (b; c))⟧ ∪ ⟦d⟧) ≡ {<↓>} ⊗ (⟦((a; b); c)⟧ ∪ ⟦d⟧) -- by ta2

$\Rightarrow$ {<↓>} ⊗ ((⟦a⟧ ⊗ ⟦(b; c)⟧) ∪ ⟦d⟧)

≡ {<↓>} ⊗ ((⟦(a; b)⟧ ⊗ ⟦c⟧) ∪ ⟦d⟧) -- by ts1

$\Rightarrow$ {<↓>} ⊗ ((⟦a⟧ ⊗ ⟦b⟧ ⊗ ⟦c⟧) ∪ ⟦d⟧)

≡ {<↓>} ⊗ ((⟦a⟧ ⊗ ⟦b⟧ ⊗ ⟦c⟧) ∪ ⟦d⟧) -- by ts1

B.2.1.3 Congruence in s.1 with the parallel composition operator

If $\forall a, b, c \in Activity \bullet$ ⟦a; (b; c)⟧ ≡ ⟦ (a; b); c ⟧, **then**

$\forall a, b, c, d \in Activity \bullet$ ⟦(a; (b; c)) || d⟧ ≡ ⟦ ((a; b); c) || d⟧

$\Rightarrow$ (⟦(a; (b; c))⟧ // ⟦d⟧) ≡ (⟦((a; b); c)⟧ // ⟦d⟧) -- by tp1

$\Rightarrow$ ((⟦a⟧ ⊗ ⟦(b; c)⟧) // ⟦d⟧) ≡ ((⟦(a; b)⟧ ⊗ ⟦c⟧) // ⟦d⟧) -- by ts1

$\Rightarrow$ ((⟦a⟧ ⊗ ⟦b⟧ ⊗ ⟦c⟧) // ⟦d⟧) ≡ ((⟦a⟧ ⊗ ⟦b⟧ ⊗ ⟦c⟧) // ⟦d⟧) -- by ts1

B.2.1.4 Congruence in s.1 with the until-loop

If $\forall a, b, c \in Activity \bullet$ ⟦a; (b; c)⟧ ≡ ⟦ (a; b); c ⟧, **then**

$\forall a, b, c \in Activity \bullet$ ⟦μx.((a; (b; c)); ε + x) ⟧ ≡ ⟦ μx.(((a; b); c); ε + x)⟧

$\Rightarrow$ μt.(⟦a; (b; c)⟧ ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.(⟦(a; b); c⟧ ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by tr2

$\Rightarrow$ μt.(⟦a⟧ ⊗ ⟦(b; c)⟧ ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.(⟦(a; b)⟧ ⊗ ⟦c⟧ ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by ts1

$\Rightarrow$ μt.(⟦a⟧ ⊗ ⟦b⟧ ⊗ ⟦c⟧ ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

$\equiv \mu t.(\ [\![a]\!] \otimes [\![b]\!] \otimes [\![c]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by ts1

B.2.1.5 Congruence in s.1 with the while-loop

If $\forall a, b, c \in Activity \bullet [\![a; (b; c)]\!] \equiv [\![(a; b); c\ ]\!]$, **then**

$\forall a, b, c \in Activity \bullet [\![\mu x.(\varepsilon + (a; (b; c)); x)]\!] \equiv [\![\mu x.(\varepsilon + ((a; b); c); x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a; (b; c)]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![(a; b); c]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes [\![(b; c)]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![(a; b)]\!] \otimes [\![c]\!] \otimes t)))$ -- by ts1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes [\![b]\!] \otimes [\![c]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes [\![b]\!] \otimes [\![c]\!] \otimes t)))$ -- by ts1

B.2.1.6 Congruence in s.1 with the encapsulation

If $\forall a, b, c \in Activity \bullet [\![a; (b; c)]\!] \equiv [\![(a; b); c\ ]\!]$, **then**

$\forall a, b, c \in Activity \bullet [\![\{a; (b; c)\}_T\ ]\!] \equiv [\![\{(a; b); c\}_T]\!]$

$\Rightarrow$unpack($[\![a; (b; c)]\!]$) $\equiv$ unpack($[\![(a; b); c]\!]$) -- by tu1

$\Rightarrow$ unpack($[\![a]\!] \otimes [\![(b; c)]\!]$) $\equiv$ unpack($[\![(a; b)]\!] \otimes [\![c]\!]$) -- by ts1

$\Rightarrow$ unpack($[\![a]\!] \otimes [\![b]\!] \otimes [\![c]\!]$) $\equiv$ unpack($[\![a]\!] \otimes [\![b]\!] \otimes [\![c]\!]$) -- by ts1

B.2.2 Showing congruence for basic operators in the right distributivity of sequence over selection axiom

The right distributivity of sequence over selection axiom (s.2) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.2.2.1 Congruence in s.2 with the sequence operator

If $\forall a, b, c \in Activity \bullet [\![(a + b); c]\!] \equiv [\![(a; c) + (b; c)]\!]$, **then**

$\forall a, b, c, d \in Activity \bullet [\![((a + b); c); d]\!] \equiv [\![\ ((a; c) + (b; c)); d]\!]$

$\Rightarrow [\![(a + b); c\ ]\!] \otimes [\![d]\!] \equiv\ [\![\ (a; c) + (b; c)]\!] \otimes [\![d]\!]$ -- by ts1

$\Rightarrow [\![a + b]\!] \otimes [\![c]\!] \otimes [\![d]\!] \equiv\ [\![\ (a; c) + (b; c)]\!] \otimes [\![d]\!]$ -- by ts1

$\Rightarrow$ {<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧) ⊗ ⟦c⟧ ⊗ ⟦d⟧

≡ {<↓>} ⊗ (⟦a; c⟧ ∪ ⟦b; c⟧) ⊗ ⟦d⟧ -- by ta2

$\Rightarrow$ {<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧) ⊗ ⟦c⟧ ⊗ ⟦d⟧

≡ {<↓>} ⊗ ((⟦a⟧ ⊗ ⟦c⟧) ∪ (⟦b⟧ ⊗ ⟦c⟧)) ⊗ ⟦d⟧ -- by ts1

$\Rightarrow$ {<↓>} ⊗ ((⟦a⟧ ⊗ ⟦c⟧) ∪ (⟦b⟧ ⊗ ⟦c⟧)) ⊗ ⟦d⟧

≡ {<↓>} ⊗ ((⟦a⟧ ⊗ ⟦c⟧) ∪ (⟦b⟧ ⊗ ⟦c⟧)) ⊗ ⟦d⟧

-- by distribution of ⊗ over union

B.2.2.2 Congruence in s.2 with the selection operator

If $\forall a, b, c \in Activity \bullet$ ⟦(a + b); c⟧ ≡ ⟦(a; c) + (b; c)⟧, **then**

$\forall a, b, c, d \in Activity \bullet$ ⟦((a + b); c) + d⟧ = ⟦ ((a; c) + (b; c)) + d⟧

$\Rightarrow$ {<↓>} ⊗ (⟦(a + b); c ⟧ ∪ ⟦d⟧)

≡ {<↓>} ⊗ (⟦ (a; c) + (b; c)⟧ ∪ ⟦d⟧) -- by ta2

$\Rightarrow$ {<↓>} ⊗ ((⟦a + b⟧ ⊗ ⟦c ⟧)∪ ⟦d⟧)

≡ {<↓>} ⊗ (⟦ (a; c) + (b; c)⟧ ∪ ⟦d⟧) -- by ts1

$\Rightarrow$ {<↓>} ⊗ (({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧)) ⊗ ⟦c ⟧) ∪ ⟦d⟧)

≡ {<↓>} ⊗ (({<↓>} ⊗ (⟦a; c⟧ ∪ ⟦b; c⟧)) ∪ ⟦d⟧) -- by ta2

$\Rightarrow$ {<↓>} ⊗ (({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧)) ⊗ ⟦c ⟧) ∪ ⟦d⟧)

≡ {<↓>} ⊗ (({<↓>} ⊗ ((⟦a⟧ ⊗ ⟦c⟧) ∪ (⟦b⟧ ⊗ ⟦c⟧))) ∪ ⟦d⟧) -- by ts1

$\Rightarrow$ {<↓>} ⊗ (({<↓>} ⊗ ((⟦a⟧ ∪ ⟦b⟧) ⊗ ⟦c ⟧)) ∪ ⟦d⟧)

≡ {<↓>} ⊗ (({<↓>} ⊗ ((⟦a⟧ ⊗ ⟦c⟧) ∪ (⟦b⟧ ⊗ ⟦c⟧))) ∪ ⟦d⟧)

-- by associativity of ⊗

$\Rightarrow$ {<↓>} ⊗ (({<↓>} ⊗ ((⟦a⟧ ⊗ ⟦c⟧) ∪ (⟦b⟧ ⊗ ⟦c⟧))) ∪ ⟦d⟧)

≡ {<↓>} ⊗ (({<↓>} ⊗ ((⟦a⟧ ⊗ ⟦c⟧) ∪ (⟦b⟧ ⊗ ⟦c⟧))) ∪ ⟦d⟧)

-- by distribution of ⊗ over union

B.2.2.3 Congruence in s.2 with the parallel composition operator

If $\forall a,b,c \in Activity \bullet$ ⟦(a + b); c⟧ ≡ ⟦(a; c) + (b; c)⟧, **then**

$\forall a,b,c,d \in Activity \bullet$ ⟦((a + b); c) || d⟧ ≡ ⟦ ((a; c) + (b; c)) || d⟧

⇒ ⟦(a + b); c ⟧ // ⟦d⟧ ≡ ⟦ (a; c) + (b; c)⟧ // ⟦d⟧ -- by tp1

⇒ (⟦a + b⟧ ⊗ ⟦c⟧) // ⟦d⟧ ≡ ⟦ (a; c) + (b; c)⟧ // ⟦d⟧ -- by ts1

⇒ ({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧) ⊗ ⟦c⟧) // ⟦d⟧

≡ ({<↓>} ⊗ (⟦a; c⟧ ∪ ⟦b; c⟧)) // ⟦d⟧ -- by ta2

⇒ ({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧) ⊗ ⟦c⟧) // ⟦d⟧

≡ ({<↓>} ⊗ ((⟦a⟧ ⊗ ⟦c⟧) ∪ (⟦b⟧ ⊗ ⟦c⟧))) // ⟦d⟧ -- by ts1

⇒ {<↓>} ⊗ ((⟦a⟧ ⊗ ⟦c⟧) ∪ (⟦b⟧ ⊗ ⟦c⟧)) // ⟦d⟧

≡ {<↓>} ⊗ ((⟦a⟧ ⊗ ⟦c⟧) ∪ (⟦b⟧ ⊗ ⟦c⟧)) // ⟦d⟧

-- by distribution of ⊗ over union

B.2.2.4 Congruence in s.2 with the until-loop

If $\forall a,b,c \in Activity \bullet$ ⟦(a + b); c⟧ ≡ ⟦(a; c) + (b; c)⟧, **then**

$\forall a,b,c \in Activity \bullet$ ⟦μx.(((a + b); c) ; ε + x)⟧ ≡ ⟦μx.(((a; c) + (b; c)); ε + x)⟧

⇒ μt.(⟦(a + b); c ⟧ ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.(⟦ (a; c) + (b; c)⟧ ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by tr2

⇒ μt.((⟦a + b⟧ ⊗ ⟦c⟧) ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.(⟦ (a; c) + (b; c)⟧ ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by ts1

⇒ μt.(({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧) ⊗ ⟦c⟧) ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.(({<↓>} ⊗ (⟦a; c⟧ ∪ ⟦b; c⟧)) ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

-- by ta2

⇒ μt.(({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧) ⊗ ⟦c⟧) ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.(({<↓>} ⊗ ((⟦a⟧ ⊗ ⟦c⟧) ∪ (⟦b⟧ ⊗ ⟦c⟧))) ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

-- by ts1

$\Rightarrow \mu t.(\ (\{<\downarrow>\} \otimes (([\![a]\!] \otimes [\![c]\!]) \cup ([\![b]\!] \otimes [\![c]\!]))) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow>\} \otimes (([\![a]\!] \otimes [\![c]\!]\ ) \cup ([\![b]\!] \otimes [\![c]\!]))) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

-- by distribution of $\otimes$ over union

B.2.2.5 Congruence in s.2 with the while-loop

If $\forall a, b, c \in Activity \bullet [\![(a + b); c]\!] \equiv [\![(a; c) + (b; c)]\!]$, **then**

$\forall a, b, c \in Activity \bullet [\![\mu x.(\varepsilon + ((a + b); c); x)]\!] \equiv [\![\mu x.(\varepsilon + ((a; c) + (b; c)); x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![(a + b); c\ ]\!] \otimes t)))$

$\equiv\ \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\ (a; c) + (b; c)]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ ([\![a + b]\!] \otimes [\![c]\!])\ \otimes t)))$

$\equiv\ \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\ (a; c) + (b; c)]\!] \otimes t)))$ -- by ts1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!]) \otimes [\![c]\!]\ ) \otimes t)))$

$\equiv\ \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\{<\downarrow>\} \otimes ([\![a; c]\!] \cup [\![b; c]\!])) \otimes t)))$ -- by ta2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!]) \otimes [\![c]\!]) \otimes t)))$

$\equiv\ \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\{<\downarrow>\} \otimes (([\![a]\!] \otimes [\![c]\!]\ )$

$\cup ([\![b]\!] \otimes [\![c]\!])))\ \otimes t)))$ -- by ts1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\{<\downarrow>\} \otimes (([\![a]\!] \otimes [\![c]\!]) \cup ([\![b]\!] \otimes [\![c]\!])))\ \otimes t)))$

$\equiv\ \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\{<\downarrow>\} \otimes (([\![a]\!] \otimes [\![c]\!]\ )$

$\cup ([\![b]\!] \otimes [\![c]\!])))\ \otimes t)))$ -- by distribution of $\otimes$ over union

B.2.2.6 Congruence in s.2 with the encapsulation

If $\forall a, b, c \in Activity \bullet [\![(a + b); c]\!] \equiv [\![(a; c) + (b; c)]\!]$, **then**

$\forall a, b, c \in Activity \bullet [\![\{(a + b); c\}_T]\!] \equiv [\![\{(a; c) + (b; c)\}_T]\!]$

$\Rightarrow$ unpack($[\![(a + b); c]\!]$) $\equiv$ unpack($[\![\ (a; c) + (b; c)]\!]$) -- by tu1

$\Rightarrow$ unpack($[\![a + b]\!] \otimes [\![c]\!]$) $\equiv$ unpack($[\![\ (a; c) + (b; c)]\!]$) -- by ts1

$\Rightarrow$ unpack({<↓>} $\otimes$ ([[a]] $\cup$ [[b]]) $\otimes$ [[c]])

$\equiv$ unpack({<↓>} $\otimes$ ([[a; c]] $\cup$ [[b; c]])) -- by ta2

$\Rightarrow$ unpack({<↓>} $\otimes$ ([[a]] $\cup$ [[b]]) $\otimes$ [[c]])

$\equiv$ unpack({<↓>} $\otimes$ (([[a]] $\otimes$ [[c]]) $\cup$ ([[b]] $\otimes$ [[c]]))) -- by ts1

$\Rightarrow$ unpack({<↓>} $\otimes$ (([[a]] $\otimes$ [[c]]) $\cup$ ([[b]] $\otimes$ [[c]])))

$\equiv$ unpack({<↓>} $\otimes$ (([[a]] $\otimes$ [[c]]) $\cup$ ([[b]] $\otimes$ [[c]])))

-- by distribution of $\otimes$ over union

B.2.3 Showing congruence for basic operators in the empty sequence axiom

The empty sequence axiom (s.3) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.2.3.1 Congruence in s.3 with the sequence operator

If $\forall a \in Activity \bullet$ [[a; ε]] $\equiv$ [[ε; a]] $\equiv$ [[a]], **then**

$\forall a, b \in Activity \bullet$ [[(a; ε); b]] $\equiv$ [[(ε; a); b]] $\equiv$ [[a; b]]

$\Rightarrow$[[a; ε]] $\otimes$ [[b]] $\equiv$ [[ε; a]] $\otimes$ [[b]] $\equiv$ [[a]] $\otimes$ [[b]] -- by ts1

$\Rightarrow$[[a]] $\otimes$ [[ε]] $\otimes$ [[b]] $\equiv$ [[ε]] $\otimes$ [[a]] $\otimes$ [[b]] $\equiv$ [[a]] $\otimes$ [[b]] -- by ts1

$\Rightarrow$[[a]] $\otimes$ {<>} $\otimes$ [[b]] $\equiv$ {<>} $\otimes$ [[a]] $\otimes$ [[b]] $\equiv$ [[a]] $\otimes$ [[b]] -- by tb1

$\Rightarrow$[[a]] $\otimes$ [[b]] $\equiv$ [[a]] $\otimes$ [[b]] $\equiv$ [[a]] $\otimes$ [[b]] -- by identity for $\otimes$

B.2.3.2 Congruence in s.3 with the selection operator

If $\forall a \in Activity \bullet$ [[a; ε]] $\equiv$ [[ε; a]] $\equiv$ [[a]], **then**

$\forall a, b \in Activity \bullet$ [[(a; ε) + b]] $\equiv$ [[(ε; a) + b]] $\equiv$ [[a + b]]

$\Rightarrow$ {<↓>} $\otimes$ ([[a; ε]] $\cup$ [[b]]) $\equiv$ {<↓>} $\otimes$ ([[ε; a]] $\cup$ [[b]])

$\equiv$ {<↓>} $\otimes$ ([[a]] $\cup$ [[b]]) -- by ta2

$\Rightarrow$ {<↓>} $\otimes$ (([[a]] $\otimes$ [[ε]]) $\cup$ [[b]]) $\equiv$ {<↓>} $\otimes$ (([[ε]] $\otimes$ [[a]]) $\cup$ [[b]])

$\equiv$ {<↓>} $\otimes$ ([[a]] $\cup$ [[b]]) -- by ts1

$\Rightarrow \{<\downarrow>\} \otimes ((⟦a⟧ \otimes \{<>\}) \cup ⟦b⟧) \equiv \{<\downarrow>\} \otimes ((\{<>\} \otimes ⟦a⟧) \cup ⟦b⟧)$

$\equiv \{<\downarrow>\} \otimes (⟦a⟧ \cup ⟦b⟧)$ -- by tb1

$\Rightarrow \{<\downarrow>\} \otimes (⟦a⟧ \cup ⟦b⟧) \equiv \{<\downarrow>\} \otimes (⟦a⟧ \cup ⟦b⟧)$

$\equiv \{<\downarrow>\} \otimes (⟦a⟧ \cup ⟦b⟧)$ -- by identity for $\otimes$

B.2.3.3 Congruence in s.3 with the parallel composition operator

If $\forall a \in Activity \bullet ⟦a; \varepsilon⟧ \equiv ⟦\varepsilon; a⟧ \equiv ⟦a⟧$, **then**

$\forall a, b \in Activity \bullet ⟦(a; \varepsilon) \parallel b⟧ \equiv ⟦(\varepsilon; a) \parallel b⟧ \equiv ⟦a \parallel b⟧$

$\Rightarrow ⟦a; \varepsilon⟧ // ⟦b⟧ \equiv ⟦\varepsilon; a⟧ // ⟦b⟧ \equiv ⟦a⟧ // ⟦b⟧$ -- by tp1

$\Rightarrow (⟦a⟧ \otimes ⟦\varepsilon⟧) // ⟦b⟧ \equiv (⟦\varepsilon⟧ \otimes ⟦a⟧) // ⟦b⟧ \equiv ⟦a⟧ // ⟦b⟧$ -- by ts1

$\Rightarrow (⟦a⟧ \otimes \{<>\}) // ⟦b⟧ \equiv (\{<>\} \otimes ⟦a⟧) // ⟦b⟧ \equiv ⟦a⟧ // ⟦b⟧$ -- by tb1

$\Rightarrow ⟦a⟧ // ⟦b⟧ \equiv ⟦a⟧ // ⟦b⟧ \equiv ⟦a⟧ // ⟦b⟧$ -- by identity for $\otimes$

B.2.3.4 Congruence in s.3 with the until-loop

If $\forall a \in Activity \bullet ⟦a; \varepsilon⟧ \equiv ⟦\varepsilon; a⟧ \equiv ⟦a⟧$, **then**

$\forall a \in Activity \bullet ⟦\mu x.((a; \varepsilon); \varepsilon + x)⟧ \equiv ⟦\mu x.((\varepsilon; a); \varepsilon + x)⟧ \equiv ⟦\mu x.(a; \varepsilon + x)⟧$

$\Rightarrow \mu t.(⟦a; \varepsilon⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(⟦\varepsilon; a⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(⟦a⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.(⟦a⟧ \otimes ⟦\varepsilon⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(⟦\varepsilon⟧ \otimes ⟦a⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(⟦a⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by ts1

$\Rightarrow \mu t.(⟦a⟧ \otimes \{<>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<>\} \otimes ⟦a⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(⟦a⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tb1

$\Rightarrow \mu t.(⟦a⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

≡ μt.([[a]] ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.([[a]] ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by identity for ⊗

B.2.3.5 Congruence in s.3 with the while-loop

If $\forall a \in Activity \bullet [[a; \varepsilon]] \equiv [[\varepsilon; a]] \equiv [[a]]$, **then**

$\forall a \in Activity \bullet [[\mu x.(\varepsilon + (a; \varepsilon); x)]] \equiv [[\mu x.(\varepsilon + (\varepsilon; a); x)]] \equiv [[\mu x.(\varepsilon + a; x)]]$

⇒μt.({<↓>} ∪ ({<↓>} ⊗ ([[a; ε]] ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ([[ε; a]] ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ([[a]] ⊗ t))) -- by tr4

⇒μt.({<↓>} ∪ ({<↓>} ⊗ ([[a]] ⊗ [[ε]] ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ([[ε]] ⊗ [[a]] ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ([[a]] ⊗ t))) -- by ts1

⇒μt.({<↓>} ∪ ({<↓>} ⊗ ([[a]] ⊗ {<>} ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ({<>} ⊗ [[a]] ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ([[a]] ⊗ t))) -- by tb1

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ ([[a]] ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ([[a]] ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ([[a]] ⊗ t))) -- by identity for ⊗

B.2.3.6 Congruence in s.3 with the encapsulation

If $\forall a \in Activity \bullet [[a; \varepsilon]] \equiv [[\varepsilon; a]] \equiv [[a]]$, **then**

$\forall a \in Activity \bullet [[\{a; \varepsilon\}_T]] \equiv [[\{\varepsilon; a\}_T]] \equiv [[\{a\}_T]]$

⇒ unpack([[a; ε]]) ≡ unpack([[ε; a]]) ≡ unpack([[a]]) -- by tu1

⇒ unpack([[a]] ⊗ [[ε]]) ≡ unpack([[ε]] ⊗ [[a]]) ≡ unpack([[a]]) -- by ts1

⇒ unpack([[a]] ⊗ {<>}) ≡ unpack({<>} ⊗ [[a]]) ≡ unpack([[a]]) -- by tb1

⇒ unpack([[a]]) ≡ unpack([[a]]) ≡ unpack([[a]]) -- by identity for ⊗

B.2.4 Showing congruence for basic operators in the fail on sequence

The fail on sequence axiom (S.4) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.2.4.1 Congruence in s.4 with the sequence operator

If $\forall a \in Activity \bullet [\![\phi; a]\!] \equiv [\![\phi]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(\phi; a); b]\!] \equiv [\![\phi; b]\!]$

$\Rightarrow [\![\phi; a]\!] \otimes [\![b]\!] \equiv [\![\phi]\!] \otimes [\![b]\!]$ -- by ts1

$\Rightarrow [\![\phi]\!] \otimes [\![a]\!] \otimes [\![b]\!] \equiv [\![\phi]\!] \otimes [\![b]\!]$ -- by ts1

$\Rightarrow \{<\phi>\} \otimes [\![a]\!] \otimes [\![b]\!] \equiv \{<\phi>\} \otimes [\![b]\!]$ -- by tb2

$\Rightarrow \{<\phi>\} \otimes \{t_1, t_2, \ldots, t_n\} \otimes [\![b]\!] \equiv \{<\phi>\} \otimes [\![b]\!]$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\phi>\} \otimes [\![b]\!] \equiv \{<\phi>\} \otimes [\![b]\!]$ $\bigcup_{i=1}^{n} \{<\phi># t_i\}$

B.2.4.2 Congruence in s.4 with the selection operator

If $\forall a \in Activity \bullet [\![\phi; a]\!] \equiv [\![\phi]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(\phi; a) + b]\!] \equiv [\![\phi + b]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![\phi; a]\!] \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes ([\![\phi]\!] \cup [\![b]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes (([\![\phi]\!] \otimes [\![a]\!]) \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes ([\![\phi]\!] \cup [\![b]\!])$ -- by ts1

$\Rightarrow \{<\downarrow>\} \otimes ((\{<\phi>\} \otimes [\![a]\!]) \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes (\{<\phi>\} \cup [\![b]\!])$ -- by tb2

$\Rightarrow \{<\downarrow>\} \otimes ((\{<\phi>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (\{<\phi>\} \cup [\![b]\!])$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\downarrow>\} \otimes (\{<\phi>\} \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes (\{<\phi>\} \cup [\![b]\!])$ $\bigcup_{i=1}^{n} \{<\phi># t_i\}$

B.2.4.3 Congruence in s.4 with the parallel composition operator

If $\forall a \in Activity \bullet [\![\phi; a]\!] \equiv [\![\phi]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(\phi; a) \,||\, b]\!] \equiv [\![\phi \,||\, b]\!]$

$\Rightarrow [\![\phi; a]\!] // [\![b]\!] \equiv [\![\phi]\!] // [\![b]\!]$ -- by tp1

$\Rightarrow ([\![\phi]\!] \otimes [\![a]\!]) // [\![b]\!] \equiv [\![\phi]\!] // [\![b]\!]$ -- by ts1

$\Rightarrow (\{<\phi>\} \otimes [\![a]\!]) // [\![b]\!] \equiv \{<\phi>\} // [\![b]\!]$ -- by tb2

$\Rightarrow (\{<\phi>\} \otimes \{t_1, t_2, \ldots, t_n\}) // [\![b]\!] \equiv \{<\phi>\} // [\![b]\!]$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\phi>\} // [\![b]\!] \equiv \{<\phi>\} // [\![b]\!]$ $\bigcup_{i=1}^{n} \{<\phi>\#t_i\}$

B.2.4.4 Congruence in s.4 with the until-loop

If $\forall a \in Activity \bullet [\![\phi; a]\!] \equiv [\![\phi]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.((\phi; a); \varepsilon + x)]\!] \equiv [\![\mu x.(\phi; \varepsilon + x)]\!]$

$\Rightarrow \mu t.([\![\phi; a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![\phi]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.([\![\phi]\!] \otimes [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![\phi]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by ts1

$\Rightarrow \mu t.(\{<\phi>\} \otimes [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<\phi>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(\{<\phi>\} \otimes \{t_1, t_2, \ldots, t_n\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<\phi>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \mu t.(\{<\phi>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<\phi>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ $\bigcup_{i=1}^{n} \{<\phi>\#t_i\}$

B.2.4.5 Congruence in s.4 with the while-loop

If $\forall a \in Activity \bullet [\![\phi; a]\!] \equiv [\![\phi]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.(\varepsilon + (\phi; a); x)]\!] \equiv [\![\mu x.(\varepsilon + \phi; x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\phi; a]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\phi]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\phi]\!] \otimes [\![a]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\phi]\!] \otimes t)))$ -- by ts1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\phi>\} \otimes [\![a]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\phi>\} \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\phi>\} \otimes \{t_1, t_2, \ldots, t_n\} \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\phi>\} \otimes t)))$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\phi>\} \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\phi>\} \otimes t)))$ $\bigcup_{i=1}^{n} \{<\phi>\#t_i\}$

B.2.4.6 Congruence in s.4 with the encapsulation

If $\forall a \in Activity \bullet [\![\phi; a]\!] \equiv [\![\phi]\!]$, **then**

$\forall a \in Activity \bullet [\![\{\phi; a\}_T]\!] \equiv [\![\{\phi\}_T]\!]$

$\Rightarrow unpack([\![\phi; a]\!]) \equiv unpack([\![\phi]\!])$ -- by tu1

$\Rightarrow unpack([\![\phi]\!] \otimes [\![a]\!]) \equiv unpack([\![\phi]\!])$ -- by ts1

$\Rightarrow unpack(\{<\phi>\} \otimes [\![a]\!]) \equiv unpack(\{<\phi>\})$ -- by tb2

$\Rightarrow unpack(\{<\phi>\} \otimes \{t_1, t_2, \ldots, t_n\})$

$\equiv unpack(\{<\phi>\})$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow unpack(\{<\phi>\}) \equiv unpack(\{<\phi>\})$ $\bigcup_{i=1}^{n} \{<\phi>\#t_i\}$

B.2.5 Showing congruence for basic operators in the succeed on sequence axiom

The succeed on sequence axiom (s.5) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.2.5.1 Congruence in s.5 with the sequence operator

If $\forall a \in Activity \bullet [\![\sigma; a]\!] \equiv [\![\sigma]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(\sigma; a); b]\!] \equiv [\![\sigma; b]\!]$

$\Rightarrow [\![\sigma; a]\!] \otimes [\![b]\!] \equiv [\![\sigma]\!] \otimes [\![b]\!]$ -- by ts1

$\Rightarrow [\![\sigma]\!] \otimes [\![a]\!] \otimes [\![b]\!] \equiv [\![\sigma]\!] \otimes [\![b]\!]$ -- by ts1

$\Rightarrow \{<\sigma>\} \otimes [\![a]\!] \otimes [\![b]\!] \equiv \{<\sigma>\} \otimes [\![b]\!]$ -- by tb2

$\Rightarrow \{<\sigma>\} \otimes \{t_1, t_2, \ldots, t_n\} \otimes [\![b]\!] \equiv \{<\sigma>\} \otimes [\![b]\!]$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\sigma>\} \otimes [\![b]\!] \equiv \{<\sigma>\} \otimes [\![b]\!]$ $\bigcup_{i=1}^{n} \{<\sigma>\#t_i\}$

B.2.5.2 Congruence in s.5 with the selection operator

If $\forall a \in Activity \bullet [\![\sigma; a]\!] \equiv [\![\sigma]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(\sigma; a) + b]\!] \equiv [\![\sigma + b]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![\sigma; a]\!] \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes ([\![\sigma]\!] \cup [\![b]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes (([\![\sigma]\!] \otimes [\![a]\!]) \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes ([\![\sigma]\!] \cup [\![b]\!])$ -- by ts1

$\Rightarrow \{<\downarrow>\} \otimes ((\{<\sigma>\} \otimes [\![a]\!]) \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes (\{<\sigma>\} \cup [\![b]\!])$ -- by tb2

$\Rightarrow \{<\downarrow>\} \otimes ((\{<\sigma>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (\{<\sigma>\} \cup [\![b]\!])$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\downarrow>\} \otimes (\{<\sigma>\} \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes (\{<\sigma>\} \cup [\![b]\!])$ $\bigcup_{i=1}^{n} \{<\sigma>\#t_i\}$

B.2.5.3 Congruence in s.5 with the parallel composition operator

If $\forall a \in Activity \bullet [\![\sigma; a]\!] \equiv [\![\sigma]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(\sigma; a) \parallel b]\!] \equiv [\![\sigma \parallel b]\!]$

$\Rightarrow [\![\sigma; a]\!] // [\![b]\!] \equiv [\![\sigma]\!] // [\![b]\!]$ -- by tp1

$\Rightarrow ([\![\sigma]\!] \otimes [\![a]\!]) // [\![b]\!] \equiv [\![\sigma]\!] // [\![b]\!]$ -- by ts1

$\Rightarrow \{<\sigma>\} \otimes [\![a]\!] // [\![b]\!] \equiv \{<\sigma>\} // [\![b]\!]$ -- by tb2

$\Rightarrow \{<\sigma>\} \otimes \{t_1, t_2, \ldots, t_n\} // [\![b]\!] \equiv \{<\sigma>\} // [\![b]\!]$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\sigma>\} // [\![b]\!] \equiv \{<\sigma>\} // [\![b]\!]$ $\bigcup_{i=1}^{n} \{<\sigma>\#t_i\}$

B.2.5.4 Congruence in s.5 with the until-loop

If $\forall a \in Activity \bullet [\![\sigma; a]\!] \equiv [\![\sigma]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.((\sigma; a); \varepsilon + x)]\!] \equiv [\![\mu x.(\sigma; \varepsilon + x)]\!]$

$\Rightarrow \mu t.([\![\sigma; a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![\sigma]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.([\![\sigma]\!] \otimes [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![\sigma]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by ts1

$\Rightarrow \mu t.(\{<\sigma>\} \otimes [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<\sigma>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(\{<\sigma>\} \otimes \{t_1, t_2, \ldots, t_n\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<\phi>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \mu t.(\{<\sigma>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<\sigma>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ $\bigcup_{i=1}^{n} \{<\sigma>\# t_i\}$

B.2.5.5 Congruence in s.5 with the while-loop

If $\forall a \in Activity \bullet [\![\sigma; a]\!] \equiv [\![\sigma]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.(\varepsilon + (\sigma; a); x)]\!] \equiv [\![\mu x.(\varepsilon + \sigma; x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\sigma; a]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\sigma]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\sigma]\!] \otimes [\![a]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\sigma]\!] \otimes t)))$ -- by ts1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\sigma>\} \otimes [\![a]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\sigma>\} \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\sigma>\} \otimes \{t_1, t_2, \ldots, t_n\} \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\sigma>\} \otimes t)))$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\sigma>\} \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\sigma>\} \otimes t)))$ $\bigcup_{i=1}^{n} \{<\sigma>\# t_i\}$

B.2.5.6 Congruence in s.5 with the encapsulation

If $\forall a \in Activity \bullet [\![\sigma; a]\!] \equiv [\![\sigma]\!]$, **then**

$\forall a \in Activity \bullet [\![\{\sigma; a\}_T]\!] \equiv [\![\{\sigma\}_T]\!]$

$\Rightarrow$ unpack($[\![\sigma; a]\!]$) $\equiv$ unpack($[\![\sigma]\!]$) -- by tu1

$\Rightarrow$ unpack($[\![\sigma]\!] \otimes [\![a]\!]$) $\equiv$ unpack($[\![\sigma]\!]$) -- by ts1

$\Rightarrow$ unpack($\{<\sigma>\} \otimes [\![a]\!]$) $\equiv$ unpack($\{<\sigma>\}$) -- by tb2

$\Rightarrow$ unpack($\{<\sigma>\} \otimes \{t_1, t_2, \ldots, t_n\}$)

$\equiv$ unpack($\{<\sigma>\}$) Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow$ unpack($\{<\sigma>\}$) $\equiv$ unpack($\{<\sigma>\}$) $\bigcup_{i=1}^{n} \{<\sigma>\#t_i\}$

B.3 Showing congruence for selection

Congruence for selection is depicted in this section for the axioms of associative selection, commutative selection, and idempotent selection. Every axiom is represented in combination with one of the basic operators defined for the task algebra.

B.3.1 Showing congruence for basic operators in the associative selection axiom

The associative selection axiom (sel.1) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.3.1.1 Congruence in sel.1 with the sequence operator

If $\forall a, b, c \in Activity \bullet [\![(a + b) + c]\!] \equiv [\![a + (b + c)]\!] \equiv [\![a + b + c]\!]$, **then**

$\forall a, b, c, d \in Activity \bullet [\![((a + b) + c); d]\!] \equiv [\![(a + (b + c)) ; d]\!] \equiv [\![(a + b + c) ; d]\!]$

$\Rightarrow [\![((a + b) + c)]\!] \otimes [\![d]\!] \equiv [\![(a + (b + c))]\!] \otimes [\![d]\!]$

$\equiv [\![(a + b + c)]\!] \otimes [\![d]\!]$ -- by ts1

$\Rightarrow \{<\downarrow>\} \otimes ([\![a + b]\!] \cup [\![c]\!]) \otimes [\![d]\!]$

$\equiv \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b + c]\!]) \otimes [\![d]\!]$

$\equiv \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b + c]\!]) \otimes [\![d]\!]$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (⟦a⟧ \cup ⟦b⟧)) \cup ⟦c⟧) \otimes ⟦d⟧$

$\equiv \{<\downarrow>\} \otimes (⟦a⟧ \cup (\{<\downarrow>\} \otimes (⟦b⟧ \cup ⟦c⟧))) \otimes ⟦d⟧$

$\equiv \{<\downarrow>\} \otimes (⟦a⟧ \cup (\{<\downarrow>\} \otimes (⟦b⟧ \cup ⟦c⟧))) \otimes ⟦d⟧$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes (((\{<\downarrow>\} \otimes ⟦a⟧) \cup (\{<\downarrow>\} \otimes ⟦b⟧)) \cup ⟦c⟧) \otimes ⟦d⟧$

$\equiv \{<\downarrow>\} \otimes (⟦a⟧ \cup ((\{<\downarrow>\} \otimes ⟦b⟧) \cup (\{<\downarrow>\} \otimes ⟦c⟧))) \otimes ⟦d⟧$

$\equiv \{<\downarrow>\} \otimes (⟦a⟧ \cup ((\{<\downarrow>\} \otimes ⟦b⟧) \cup (\{<\downarrow>\} \otimes ⟦c⟧))) \otimes ⟦d⟧$

-- by distribution of $\otimes$ over union

$\Rightarrow (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ⟦a⟧) \cup (\{<\downarrow>\} \otimes ⟦b⟧))) \cup (\{<\downarrow>\} \otimes ⟦c⟧) \otimes ⟦d⟧$

$\equiv (\{<\downarrow>\} \otimes ⟦a⟧) \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ⟦b⟧) \cup (\{<\downarrow>\} \otimes ⟦c⟧))) \otimes ⟦d⟧$

$\equiv (\{<\downarrow>\} \otimes ⟦a⟧) \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ⟦b⟧) \cup (\{<\downarrow>\} \otimes ⟦c⟧))) \otimes ⟦d⟧$

-- by distribution of $\otimes$ over union

$\Rightarrow ((\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes ⟦a⟧)) \cup (\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes ⟦b⟧)))$

$\cup (\{<\downarrow>\} \otimes ⟦c⟧) \otimes ⟦d⟧$

$\equiv (\{<\downarrow>\} \otimes ⟦a⟧) \cup ((\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes ⟦b⟧))$

$\cup (\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes ⟦c⟧))) \otimes ⟦d⟧$

$\equiv (\{<\downarrow>\} \otimes ⟦a⟧) \cup ((\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes ⟦b⟧))$

$\cup (\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes ⟦c⟧))) \otimes ⟦d⟧$ -- by distribution of $\otimes$ over union

$\Rightarrow ((\{<\downarrow>\} \otimes ⟦a⟧) \cup (\{<\downarrow>\} \otimes ⟦b⟧)) \cup (\{<\downarrow>\} \otimes ⟦c⟧) \otimes ⟦d⟧$

$\equiv (\{<\downarrow>\} \otimes ⟦a⟧) \cup ((\{<\downarrow>\} \otimes ⟦b⟧) \cup (\{<\downarrow>\} \otimes ⟦c⟧)) \otimes ⟦d⟧$

$\equiv (\{<\downarrow>\} \otimes ⟦a⟧) \cup ((\{<\downarrow>\} \otimes ⟦b⟧) \cup (\{<\downarrow>\} \otimes ⟦c⟧)) \otimes ⟦d⟧$ -- by cp1

B.3.1.2 Congruence in sel.1 with the selection operator

If $\forall a, b, c \in Activity \bullet ⟦(a + b) + c⟧ \equiv ⟦a + (b + c)⟧ \equiv ⟦a + b + c⟧$, **then**

$\forall a, b, c, d \in Activity \bullet ⟦((a + b) + c) + d⟧ \equiv ⟦ (a + (b + c)) + d⟧ = ⟦ (a + b + c) + d⟧$

$\Rightarrow \{<\downarrow>\} \otimes (⟦(a + b) + c⟧ \cup ⟦d⟧) \equiv \{<\downarrow>\} \otimes (⟦a + (b + c)⟧ \cup ⟦d⟧)$

$\equiv \{<\downarrow>\} \otimes ([\![a + b + c]\!] \cup [\![d]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![a + b]\!] \cup [\![c]\!])) \cup [\![d]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b + c]\!])) \cup [\![d]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b + c]\!])) \cup [\![d]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])) \cup [\![c]\!])) \cup [\![d]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![a]\!] \cup (\{<\downarrow>\} \otimes ([\![b]\!] \cup [\![c]\!])))) \cup [\![d]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![a]\!] \cup (\{<\downarrow>\} \otimes ([\![b]\!] \cup [\![c]\!])))) \cup [\![d]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes (((\{<\downarrow>\} \otimes [\![a]\!]) \cup (\{<\downarrow>\} \otimes [\![b]\!])) \cup [\![c]\!]) \cup [\![d]\!])$

$\equiv \{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes ([\![a]\!] \cup ((\{<\downarrow>\} \otimes [\![b]\!]) \cup (\{<\downarrow>\} \otimes [\![c]\!]))) \cup [\![d]\!])$

$\equiv \{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes ([\![a]\!] \cup ((\{<\downarrow>\} \otimes [\![b]\!]) \cup (\{<\downarrow>\} \otimes [\![c]\!]))) \cup [\![d]\!])$

-- by distribution of $\otimes$ over union

$\Rightarrow \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes [\![a]\!]) \cup (\{<\downarrow>\} \otimes [\![b]\!])))$

$\cup (\{<\downarrow>\} \otimes [\![c]\!]) \cup [\![d]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes [\![a]\!]) \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes [\![b]\!])$

$\cup (\{<\downarrow>\} \otimes [\![c]\!]))) \cup [\![d]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes [\![a]\!]) \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes [\![b]\!])$

$\cup (\{<\downarrow>\} \otimes [\![c]\!]))) \cup [\![d]\!])$ -- by distribution of $\otimes$ over union

$\Rightarrow \{<\downarrow>\} \otimes (((\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes [\![a]\!])) \cup (\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes [\![b]\!])))$

$\cup (\{<\downarrow>\} \otimes [\![c]\!]) \cup [\![d]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes [\![a]\!]) \cup ((\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes [\![b]\!]))$

$\cup (\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes [\![c]\!]))) \cup [\![d]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes [\![a]\!]) \cup ((\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes [\![b]\!]))$

$\cup (\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes [\![c]\!]))) \cup [\![d]\!])$ -- by distribution of $\otimes$ over union

$\Rightarrow \{<\downarrow>\} \otimes (((\{<\downarrow>\} \otimes [\![a]\!]) \cup (\{<\downarrow>\} \otimes [\![b]\!])) \cup (\{<\downarrow>\} \otimes [\![c]\!]) \cup [\![d]\!])$

≡ {<↓>} ⊗ (({<↓>} ⊗⟦a⟧) ∪ (({<↓>} ⊗ ⟦b⟧)

∪ ({<↓>} ⊗ ⟦c⟧)) ∪ ⟦d⟧)

≡ {<↓>} ⊗ (({<↓>} ⊗⟦a⟧) ∪ (({<↓>} ⊗ ⟦b⟧)

∪ ({<↓>} ⊗ ⟦c⟧)) ∪ ⟦d⟧) -- by cp1

B.3.1.3 Congruence in sel.1 with the parallel composition operator

If $\forall a,b,c \in Activity \bullet$ ⟦(a + b) + c⟧ ≡ ⟦a + (b + c)⟧ ≡ ⟦a + b + c⟧, **then**

$\forall a,b,c,d \in Activity \bullet$ ⟦((a + b) + c) || d⟧ ≡ ⟦(a + (b + c)) || d⟧ ≡ ⟦(a + b + c) || d⟧

⇒ ⟦(a + b) + c⟧ // ⟦d⟧ ≡ ⟦a + (b + c)⟧ // ⟦d⟧ ≡ ⟦a + b + c⟧ // ⟦d⟧ -- by tp1

⇒ ({<↓>} ⊗ (⟦a + b⟧ ∪ ⟦c⟧)) // ⟦d⟧ ≡ ({<↓>} ⊗ (⟦a⟧ ∪ ⟦b + c⟧)) // ⟦d⟧

≡ ({<↓>} ⊗ (⟦a⟧ ∪ ⟦b + c⟧)) // ⟦d⟧ -- by ta2

⇒ ({<↓>} ⊗ (({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧)) ∪ ⟦c⟧)) // ⟦d⟧

≡ ({<↓>} ⊗ (⟦a⟧ ∪ ({<↓>} ⊗ (⟦b⟧ ∪ ⟦c⟧)))) // ⟦d⟧

≡ ({<↓>} ⊗ (⟦a⟧ ∪ ({<↓>} ⊗ (⟦b⟧ ∪ ⟦c⟧)))) // ⟦d⟧ -- by ta2

⇒ {<↓>} ⊗ ((({<↓>} ⊗ ⟦a⟧) ∪ ({<↓>} ⊗ ⟦b⟧)) ∪ ⟦c⟧) // ⟦d⟧

≡ {<↓>} ⊗ (⟦a⟧ ∪ (({<↓>} ⊗ ⟦b⟧) ∪ ({<↓>} ⊗ ⟦c⟧))) // ⟦d⟧

≡ {<↓>} ⊗ (⟦a⟧ ∪ (({<↓>} ⊗ ⟦b⟧) ∪ ({<↓>} ⊗ ⟦c⟧))) // ⟦d⟧

-- by distribution of ⊗ over union

⇒ ({<↓>} ⊗ (({<↓>} ⊗ ⟦a⟧) ∪ ({<↓>} ⊗ ⟦b⟧))) ∪ ({<↓>} ⊗ ⟦c⟧) // ⟦d⟧

≡ ({<↓>} ⊗⟦a⟧) ∪ ({<↓>} ⊗ (({<↓>} ⊗ ⟦b⟧) ∪ ({<↓>} ⊗ ⟦c⟧))) // ⟦d⟧

≡ ({<↓>} ⊗⟦a⟧) ∪ ({<↓>} ⊗ (({<↓>} ⊗ ⟦b⟧) ∪ ({<↓>} ⊗ ⟦c⟧))) // ⟦d⟧

-- by distribution of ⊗ over union

⇒ (({<↓>} ⊗ ({<↓>} ⊗ ⟦a⟧)) ∪ ({<↓>} ⊗ ({<↓>} ⊗ ⟦b⟧)))

∪ ({<↓>} ⊗ ⟦c⟧) // ⟦d⟧

≡ ({<↓>} ⊗⟦a⟧) ∪ (({<↓>} ⊗ ({<↓>} ⊗ ⟦b⟧))

$\cup$ ({<↓>} $\otimes$ ({<↓>} $\otimes$ ⟦c⟧))) // ⟦d⟧

$\equiv$ ({<↓>} $\otimes$⟦a⟧) $\cup$ (({<↓>} $\otimes$ ({<↓>} $\otimes$ ⟦b⟧))

$\cup$ ({<↓>} $\otimes$ ({<↓>} $\otimes$ ⟦c⟧))) // ⟦d⟧ -- by distribution of $\otimes$ over union

$\Rightarrow$ (({<↓>} $\otimes$ ⟦a⟧) $\cup$ ({<↓>} $\otimes$ ⟦b⟧)) $\cup$ ({<↓>} $\otimes$ ⟦c⟧) // ⟦d⟧

$\equiv$ ({<↓>} $\otimes$⟦a⟧) $\cup$ (({<↓>} $\otimes$ ⟦b⟧) $\cup$ ({<↓>} $\otimes$ ⟦c⟧)) // ⟦d⟧

$\equiv$ ({<↓>} $\otimes$⟦a⟧) $\cup$ (({<↓>} $\otimes$ ⟦b⟧) $\cup$ ({<↓>} $\otimes$ ⟦c⟧)) // ⟦d⟧ -- by cp1

B.3.1.4 Congruence in sel.1 with the until-loop

If $\forall a, b, c \in Activity \bullet$ ⟦(a + b) + c⟧ $\equiv$ ⟦a + (b + c)⟧ $\equiv$ ⟦a + b + c⟧, **then**

$\forall a, b, c \in Activity \bullet$ ⟦μx.(((a + b) + c); ε + x)⟧ $\equiv$ ⟦μx.((a + (b + c)); ε + x)⟧

$\equiv$ ⟦μx.((a + b + c); ε + x)⟧

$\Rightarrow$ μt.(⟦((a + b) + c)⟧ $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.(⟦ (a + (b + c))⟧ $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.(⟦ (a + b + c)⟧ $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t))) -- by tr2

$\Rightarrow$ μt.({<↓>} $\otimes$ (⟦a + b⟧ $\cup$ ⟦c⟧) $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.({<↓>} $\otimes$ (⟦a⟧ $\cup$ ⟦ b + c⟧) $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.({<↓>} $\otimes$ (⟦a⟧ $\cup$ ⟦ b + c⟧) $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t))) -- by ta2

$\Rightarrow$ μt.({<↓>} $\otimes$ (({<↓>} $\otimes$ (⟦a⟧ $\cup$ ⟦b⟧)) $\cup$ ⟦c⟧) $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.({<↓>} $\otimes$ (⟦a⟧ $\cup$ ({<↓>} $\otimes$ (⟦b⟧ $\cup$ ⟦c⟧)))

$\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.({<↓>} $\otimes$ (⟦a⟧ $\cup$ ({<↓>} $\otimes$ (⟦b⟧ $\cup$ ⟦c⟧)))

$\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t))) -- by ta2

$\Rightarrow$ μt.({<↓>} $\otimes$ ((({<↓>} $\otimes$ ⟦a⟧) $\cup$ ({<↓>} $\otimes$ ⟦b⟧)) $\cup$ ⟦c⟧)

$\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.({<↓>} $\otimes$ (⟦a⟧ $\cup$ (({<↓>} $\otimes$ ⟦b⟧) $\cup$ ({<↓>} $\otimes$ ⟦c⟧)))

$\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.({<↓>} $\otimes$ ([[a]] $\cup$ (({<↓>} $\otimes$ [[b]]) $\cup$ ({<↓>} $\otimes$ [[c]])))

$\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t))) -- by distribution of $\otimes$ over union

$\Rightarrow$ μt.(({<↓>} $\otimes$ (({<↓>} $\otimes$ [[a]]) $\cup$ ({<↓>} $\otimes$ [[b]]))) $\cup$ ({<↓>} $\otimes$ [[c]])

$\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.(({<↓>} $\otimes$[[a]]) $\cup$ ({<↓>} $\otimes$ (({<↓>} $\otimes$ [[b]]) $\cup$ ({<↓>} $\otimes$ [[c]])))

$\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.(({<↓>} $\otimes$[[a]]) $\cup$ ({<↓>} $\otimes$ (({<↓>} $\otimes$ [[b]]) $\cup$ ({<↓>} $\otimes$ [[c]])))

$\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

-- by distribution of $\otimes$ over union

$\Rightarrow$ μt.(((({<↓>} $\otimes$ ({<↓>} $\otimes$ [[a]])) $\cup$ ({<↓>} $\otimes$ ({<↓>} $\otimes$ [[b]])))

$\cup$ ({<↓>} $\otimes$ [[c]])) $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.((({<↓>} $\otimes$ [[a]]) $\cup$ (({<↓>} $\otimes$ ({<↓>} $\otimes$ [[b]]))

$\cup$ ({<↓>} $\otimes$ ({<↓>} $\otimes$ [[c]])))) $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.((({<↓>} $\otimes$ [[a]]) $\cup$ (({<↓>} $\otimes$ ({<↓>} $\otimes$ [[b]]))

$\cup$ ({<↓>} $\otimes$ ({<↓>} $\otimes$ [[c]])))) $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

-- by distribution of $\otimes$ over union

$\Rightarrow$ μt.(((({<↓>} $\otimes$ [[a]]) $\cup$ ({<↓>} $\otimes$ [[b]])) $\cup$ ({<↓>} $\otimes$ [[c]]))

$\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.((({<↓>} $\otimes$[[a]]) $\cup$ (({<↓>} $\otimes$ [[b]]) $\cup$ ({<↓>} $\otimes$ [[c]])))

$\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.((({<↓>} $\otimes$[[a]]) $\cup$ (({<↓>} $\otimes$ [[b]]) $\cup$ ({<↓>} $\otimes$ [[c]])))

$\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t))) -- by cp1

B.3.1.5 Congruence in sel.1 with the while-loop

If $\forall a, b, c \in Activity \bullet$ [[(a + b) + c]] $\equiv$ [[a + (b + c)]] $\equiv$ [[a + b + c]], **then**

$\forall a, b, c \in Activity \bullet$ ⟦μx.(ε + ((a + b) + c); x)⟧ ≡ ⟦μx.(ε + (a + (b + c)); x)⟧

≡ ⟦μx.(ε + (a + b + c); x)⟧

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ (⟦((a + b) + c)⟧ ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ (⟦ (a + (b + c))⟧ ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ (⟦ (a + b + c)⟧ ⊗ t))) -- by tr4

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ ({<↓>} ⊗ (⟦a + b⟧ ∪ ⟦c⟧) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ({<↓>} ⊗ (⟦a⟧ ∪ ⟦ b + c⟧) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ({<↓>} ⊗ (⟦a⟧ ∪ ⟦ b + c⟧)

⊗ t))) -- by ta2

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ ({<↓>} ⊗ (({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧)) ∪ ⟦c⟧) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ({<↓>} ⊗ (⟦a⟧ ∪ ({<↓>}

⊗ (⟦b⟧ ∪ ⟦c⟧))) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ({<↓>} ⊗ (⟦a⟧ ∪ ({<↓>}

⊗ (⟦b⟧ ∪ ⟦c⟧))) ⊗ t))) -- by ta2

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ ({<↓>} ⊗ ((({<↓>} ⊗ ⟦a⟧)

∪ ({<↓>} ⊗ ⟦b⟧)) ∪ ⟦c⟧) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ({<↓>} ⊗ (⟦a⟧ ∪ (({<↓>} ⊗ ⟦b⟧)

∪ ({<↓>} ⊗ ⟦c⟧))) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ({<↓>} ⊗ (⟦a⟧ ∪ (({<↓>} ⊗ ⟦b⟧)

∪ ({<↓>} ⊗ ⟦c⟧))) ⊗ t))) -- by distribution of ⊗ over union

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ ((({<↓>} ⊗ (({<↓>} ⊗ ⟦a⟧) ∪ ({<↓>} ⊗ ⟦b⟧)))

∪ ({<↓>} ⊗ ⟦c⟧)) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ((({<↓>} ⊗⟦a⟧) ∪ ({<↓>} ⊗ (({<↓>}

⊗ ⟦b⟧) ∪ ({<↓>} ⊗ ⟦c⟧)))) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ((({<↓>} ⊗⟦a⟧) ∪ ({<↓>} ⊗ (({<↓>}

⊗ ⟦b⟧) ∪ ({<↓>} ⊗ ⟦c⟧)))) ⊗ t)))

-- by distribution of ⊗ over union

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ (((({<↓>} ⊗ ({<↓>} ⊗ ⟦a⟧)) ∪ ({<↓>}

⊗ ({<↓>} ⊗ ⟦b⟧))) ∪ ({<↓>} ⊗ ⟦c⟧)) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ((({<↓>} ⊗⟦a⟧) ∪ (({<↓>} ⊗ ({<↓>}

⊗ ⟦b⟧)) ∪ ({<↓>} ⊗ ({<↓>} ⊗ ⟦c⟧)))) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ((({<↓>} ⊗⟦a⟧) ∪ (({<↓>} ⊗ ({<↓>}

⊗ ⟦b⟧)) ∪ ({<↓>} ⊗ ({<↓>} ⊗ ⟦c⟧)))) ⊗ t)))

-- by distribution of ⊗ over union

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ (((({<↓>} ⊗ ⟦a⟧) ∪ ({<↓>} ⊗ ⟦b⟧)) ∪ ({<↓>}

⊗ ⟦c⟧)) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ((({<↓>} ⊗⟦a⟧) ∪ (({<↓>} ⊗ ⟦b⟧)

∪ ({<↓>} ⊗ ⟦c⟧))) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ((({<↓>} ⊗⟦a⟧) ∪ (({<↓>} ⊗ ⟦b⟧)

∪ ({<↓>} ⊗ ⟦c⟧))) ⊗ t))) -- by cp1

B.3.1.6 Congruence in sel.1 with the encapsulation

If $\forall a, b, c \in Activity \bullet$ ⟦(a + b) + c⟧ ≡ ⟦a + (b + c)⟧ ≡ ⟦a + b + c⟧, **then**

$\forall a, b, c \in Activity \bullet$ ⟦{(a + b) + c}$_T$⟧ ≡ ⟦{a + (b + c)}$_T$⟧ ≡ ⟦{a + b + c}$_T$⟧

⇒ unpack(⟦(a + b) + c⟧) ≡ unpack(⟦a + (b + c)⟧)

≡ unpack(⟦a + b + c⟧) -- by tu1

⇒ unpack({<↓>} ⊗ (⟦a + b⟧ ∪ ⟦c⟧))

≡ unpack({<↓>} ⊗ (⟦a⟧ ∪ ⟦ b + c⟧))

≡ unpack({<↓>} ⊗ (⟦a⟧ ∪ ⟦ b + c⟧)) -- by ta2

$\Rightarrow$ unpack({<↓>} ⊗ (({<↓>} ⊗ ([a] ∪ [b])) ∪ [c]))

≡ unpack({<↓>} ⊗ ([a] ∪ ({<↓>} ⊗ ([b] ∪ [c]))))

≡ unpack({<↓>} ⊗ ([a] ∪ ({<↓>} ⊗ ([b] ∪ [c])))) -- by ta2

$\Rightarrow$ unpack({<↓>} ⊗ ((({<↓>} ⊗ [a]) ∪ ({<↓>} ⊗ [b])) ∪ [c]))

≡ unpack({<↓>} ⊗ ([a] ∪ (({<↓>} ⊗ [b]) ∪ ({<↓>} ⊗ [c]))))

≡ unpack({<↓>} ⊗ ([a] ∪ (({<↓>} ⊗ [b]) ∪ ({<↓>} ⊗ [c]))))

-- by distribution of ⊗ over union

$\Rightarrow$ unpack(({<↓>} ⊗ (({<↓>} ⊗ [a]) ∪ ({<↓>} ⊗ [b]))) ∪ ({<↓>} ⊗ [c]))

≡ unpack(({<↓>} ⊗[a]) ∪ ({<↓>} ⊗ (({<↓>} ⊗ [b]) ∪ ({<↓>}

⊗ [c]))))

≡ unpack(({<↓>} ⊗[a]) ∪ ({<↓>} ⊗ (({<↓>} ⊗ [b]) ∪ ({<↓>}

⊗ [c])))) -- by distribution of ⊗ over union

$\Rightarrow$ unpack((({<↓>} ⊗ ({<↓>} ⊗ [a])) ∪ ({<↓>} ⊗ ({<↓>} ⊗ [b])))

∪ ({<↓>} ⊗ [c]))

≡ unpack(({<↓>} ⊗[a]) ∪ (({<↓>} ⊗ ({<↓>} ⊗ [b]))

∪ ({<↓>} ⊗ ({<↓>} ⊗ [c]))))

≡ unpack(({<↓>} ⊗[a]) ∪ (({<↓>} ⊗ ({<↓>} ⊗ [b]))

∪ ({<↓>} ⊗ ({<↓>} ⊗ [c])))) -- by distribution of ⊗ over union

$\Rightarrow$ unpack((({<↓>} ⊗ [a]) ∪ ({<↓>} ⊗ [b])) ∪ ({<↓>} ⊗ [c]))

≡ unpack(({<↓>} ⊗[a]) ∪ (({<↓>} ⊗ [b]) ∪ ({<↓>} ⊗ [c])))

≡ unpack(({<↓>} ⊗[a]) ∪ (({<↓>} ⊗ [b]) ∪ ({<↓>} ⊗ [c])))

-- by cp1

B.3.2 Showing congruence for basic operators in the commutative selection axiom

The commutative selection axiom (sel.2) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.3.2.1 Congruence in sel.2 with the sequence operator

If $\forall a, b \in Activity \bullet$ ⟦a + b⟧ ≡ ⟦b + a⟧, **then**

$\forall a, b, c \in Activity \bullet$ ⟦(a + b); c⟧ ≡ ⟦(b + a); c⟧

⇒ ⟦a + b⟧ ⊗ ⟦c⟧ ≡ ⟦b + a⟧ ⊗ ⟦c⟧ -- by ts1

⇒ ({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧)) ⊗ ⟦c⟧ ≡ ({<↓>} ⊗ (⟦b⟧ ∪ ⟦a⟧)) ⊗ ⟦c⟧ -- by ta2

⇒ ({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧)) ⊗ ⟦c⟧ ≡ ({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧)) ⊗ ⟦c⟧

-- by commutativity of union

B.3.2.2 Congruence in sel.2 with the selection operator

If $\forall a, b \in Activity \bullet$ ⟦a + b⟧ ≡ ⟦b + a⟧, **then**

$\forall a, b, c \in Activity \bullet$ ⟦(a + b) + c⟧ ≡ ⟦(b + a) + c⟧

⇒ {<↓>} ⊗ (⟦a + b⟧ ∪ ⟦c⟧) ≡ {<↓>} ⊗ (⟦b + a⟧ ∪ ⟦c⟧) -- by ta2

⇒ {<↓>} ⊗ (({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧)) ∪ ⟦c⟧)

≡ {<↓>} ⊗ (({<↓>} ⊗ (⟦b⟧ ∪ ⟦a⟧)) ∪ ⟦c⟧) -- by ta2

⇒ {<↓>} ⊗ (({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧)) ∪ ⟦c⟧)

≡ {<↓>} ⊗ (({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧)) ∪ ⟦c⟧)

-- by commutativity of union

B.3.2.3 Congruence in sel.2 with the parallel composition operator

If $\forall a, b \in Activity \bullet$ ⟦a + b⟧ ≡ ⟦b + a⟧, **then**

$\forall a, b, c \in Activity \bullet$ ⟦(a + b) || c⟧ ≡ ⟦(b + a) || c⟧

⇒ ⟦a + b⟧ // ⟦c⟧ ≡ ⟦b + a⟧ // ⟦c⟧ -- by tp1

⇒ ({<↓>} ⊗ (⟦a⟧ ∪ ⟦b⟧)) // ⟦c⟧ = ({<↓>} ⊗ (⟦b⟧ ∪ ⟦a⟧)) // ⟦c⟧ -- by ta2

$\Rightarrow$ ({<↓>} $\otimes$ ([a] $\cup$ [b])) // [c] $\equiv$ ({<↓>} $\otimes$ ([a] $\cup$ [b])) // [c]

-- by commutativity of union

B.3.2.4 Congruence in sel.2 with the until-loop

If $\forall a, b \in Activity \bullet$ [a + b] $\equiv$ [b + a], **then**

$\forall a, b \in Activity \bullet$ [μx.((a + b); ε + x)] $\equiv$ [μx.((b + a); ε + x)]

$\Rightarrow$ μt.([a + b] $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.([b + a] $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t))) -- by tr2

$\Rightarrow$ μt.(({<↓>} $\otimes$ ([a] $\cup$ [b])) $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.(({<↓>} $\otimes$ ([b] $\cup$ [a])) $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t))) -- by ta2

$\Rightarrow$ μt.(({<↓>} $\otimes$ ([a] $\cup$ [b])) $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

$\equiv$ μt.(({<↓>} $\otimes$ ([a] $\cup$ [b])) $\otimes$ ({<↓>} $\cup$ ({<↓>} $\otimes$ t)))

-- by commutativity of union

B.3.2.5 Congruence in sel.2 with the while-loop

If $\forall a, b \in Activity \bullet$ [a + b] $\equiv$ [b + a], **then**

$\forall a, b \in Activity \bullet$ [μx.(ε + (a + b); x)] $\equiv$ [μx.(ε + (b + a); x)]

$\Rightarrow$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ ([a + b] $\otimes$ t)))

$\equiv$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ ([b + a] $\otimes$ t))) -- by tr4

$\Rightarrow$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ (({<↓>} $\otimes$ ([a] $\cup$ [b])) $\otimes$ t)))

$\equiv$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ (({<↓>} $\otimes$ ([b] $\cup$ [a])) $\otimes$ t))) -- by ta2

$\Rightarrow$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ (({<↓>} $\otimes$ ([a] $\cup$ [b])) $\otimes$ t)))

$\equiv$ μt.({<↓>} $\cup$ ({<↓>} $\otimes$ (({<↓>} $\otimes$ ([a] $\cup$ [b])) $\otimes$ t)))

-- by commutativity of union

B.3.2.6 Congruence in sel.2 with the encapsulation

If $\forall a, b \in Activity \bullet$ [a + b] $\equiv$ [b + a], **then**

$\forall a, b \in Activity \bullet [\![\{a + b\}_T]\!] \equiv [\![\{b + a\}_T]\!]$

$\Rightarrow$ unpack($[\![a + b]\!]$) $\equiv$ unpack($[\![b + a]\!]$) -- by tu1

$\Rightarrow$ unpack($\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])$) $\equiv$ unpack($\{<\downarrow>\} \otimes ([\![b]\!] \cup [\![a]\!])$) -- by ta2

$\Rightarrow$ unpack($\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])$) $\equiv$ unpack($\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])$)

-- by commutativity of union

B.3.3 Showing congruence for basic operators in the idempotent selection axiom

The idempotent selection action (sel.3) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.3.3.1 Congruence in sel.3 with the sequence operator

If $\forall a \in Activity \bullet [\![a + a]\!] \equiv [\![a]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(a + a); b]\!] \equiv [\![a; b]\!]$

$\Rightarrow [\![a + a]\!] \otimes [\![b]\!] \equiv [\![a]\!] \otimes [\![b]\!]$ -- by ts1

$\Rightarrow [\![a]\!] \otimes [\![b]\!] \equiv [\![a]\!] \otimes [\![b]\!]$ -- by ta1

B.3.3.2 Congruence in sel.3 with the selection operator

If $\forall a \in Activity \bullet [\![a + a]\!] \equiv [\![a]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(a + a) + b]\!] \equiv [\![a + b]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![a + a]\!] \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])$ -- by ta1

B.3.3.3 Congruence in sel.3 with the parallel composition operator

If $\forall a \in Activity \bullet [\![a + a]\!] \equiv [\![a]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(a + a) \,||\, b]\!] \equiv [\![a \,||\, b]\!]$

$\Rightarrow [\![a + a]\!] \,//\, [\![b]\!] \equiv [\![a]\!] \,//\, [\![b]\!]$ -- by tp1

$\Rightarrow [\![a]\!] \,//\, [\![b]\!] \equiv [\![a]\!] \,//\, [\![b]\!]$ -- by ta1

B.3.3.4 Congruence in sel.3 with the until-loop

If $\forall a \in Activity \bullet [\![a + a]\!] \equiv [\![a]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.((a + a); \varepsilon + x)]\!] \equiv [\![\mu x.(a; \varepsilon + x)]\!]$

$\Rightarrow \mu t.([\![a + a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by ta1

B.3.3.5 Congruence in sel.3 with the while-loop

If $\forall a \in Activity \bullet [\![a + a]\!] \equiv [\![a]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.(\varepsilon + (a + a); x)]\!] \equiv [\![\mu x.(\varepsilon + a; x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a + a]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$ -- by ta1

B.3.3.6 Congruence in sel.3 with the encapsulation

If $\forall a \in Activity \bullet [\![a + a]\!] \equiv [\![a]\!]$, **then**

$\forall a \in Activity \bullet [\![\{a + a\}_T]\!] \equiv [\![\{a\}_T]\!]$

$\Rightarrow unpack([\![a + a]\!]) \equiv unpack([\![a]\!])$ -- by tu1

$\Rightarrow unpack([\![a]\!]) \equiv unpack([\![a]\!])$ -- by ta1

B.4 Showing congruence for parallel composition

Parallel composition has the axioms of associative parallel composition, commutative composition, right distributivity of concurrency over selection, instant synchronisation, *fail* in parallel composition, and *succeed* in parallel composition. In this section, the congruence is demonstrated for each of these axioms.

B.4.1 Showing congruence for basic operators in the associative parallel composition axiom

The associative parallel composition axiom (p.1) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.4.1.1 Congruence in p.1 with the sequence operator

If $\forall a,b,c \in Activity \bullet$ ⟦(a || b) || c⟧ ≡ ⟦a || (b || c)⟧, **then**

$\forall a,b,c,d \in Activity \bullet$ ⟦((a || b) || c); d⟧ ≡ ⟦(a || (b || c)); d⟧

⇒ ⟦(a || b) || c⟧ ⊗ ⟦d⟧ ≡ ⟦a || (b || c)⟧ ⊗ ⟦d⟧ -- by ts1

⇒ (⟦a || b⟧ // ⟦c⟧) ⊗ ⟦d⟧ ≡ (⟦a⟧ // ⟦b || c⟧) ⊗ ⟦d⟧ -- by tp1

⇒ (⟦a⟧ // ⟦b⟧ // ⟦c⟧) ⊗ ⟦d⟧ ≡ (⟦a⟧ // ⟦b⟧ // ⟦c⟧) ⊗ ⟦d⟧ -- by tp1

B.4.1.2 Congruence in p.1 with the selection operator

If $\forall a,b,c \in Activity \bullet$ ⟦(a || b) || c⟧ ≡ ⟦a || (b || c)⟧, **then**

$\forall a,b,c,d \in Activity \bullet$ ⟦((a || b) || c) + d⟧ ≡ ⟦(a || (b || c)) + d⟧

⇒ {<↓>} ⊗ (⟦(a || b) || c⟧ ∪ ⟦d⟧) ≡ {<↓>} ⊗ (⟦a || (b || c)⟧ ∪ ⟦d⟧) -- by ta2

⇒ {<↓>} ⊗ ((⟦a || b⟧ // ⟦c⟧) ∪ ⟦d⟧)

≡ {<↓>} ⊗ ((⟦a⟧ // ⟦b || c⟧) ∪ ⟦d⟧) -- by tp1

⇒ {<↓>} ⊗ ((⟦a⟧ // ⟦b⟧ // ⟦c⟧) ∪ ⟦d⟧)

≡ {<↓>} ⊗ ((⟦a⟧ // ⟦b⟧ // ⟦c⟧) ∪ ⟦d⟧) -- by tp1

B.4.1.3 Congruence in p.1 with the parallel composition operator

If $\forall a,b,c \in Activity \bullet$ ⟦(a || b) || c⟧ ≡ ⟦a || (b || c)⟧, **then**

$\forall a,b,c,d \in Activity \bullet$ ⟦((a || b) || c) || d⟧ ≡ ⟦(a || (b || c)) || d⟧

⇒ ⟦(a || b) || c⟧ // ⟦d⟧ ≡ ⟦a || (b || c)⟧ // ⟦d⟧ -- by tp1

⇒ ⟦a || b⟧ // ⟦c⟧ // ⟦d⟧ ≡ ⟦a⟧ // ⟦b || c⟧ // ⟦d⟧ -- by tp1

⇒ ⟦a⟧ // ⟦b⟧ // ⟦c⟧ // ⟦d⟧ ≡ ⟦a⟧ // ⟦b⟧ // ⟦c⟧ // ⟦d⟧ -- by tp1

B.4.1.4 Congruence in p.1 with the until-loop

If $\forall a,b,c \in Activity \bullet$ ⟦(a || b) || c⟧ ≡ ⟦a || (b || c)⟧, **then**

$\forall a,b,c \in Activity \bullet$ ⟦μx.(((a || b) || c); ε + x)⟧ ≡ ⟦μx.((a || (b || c)); ε + x)⟧

⇒ μt.(⟦(a || b) || c⟧ ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.(⟦a || (b || c)⟧ ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by tr2

⇒ μt.((⟦a || b⟧ // ⟦c⟧) ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.((⟦a⟧ // ⟦b || c⟧) ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by tp1

⇒ μt.((⟦a⟧ // ⟦b⟧ // ⟦c⟧) ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.((⟦a⟧ // ⟦b⟧ // ⟦c⟧) ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by tp1

B.4.1.5 Congruence in p.1 with the while-loop

If $\forall a,b,c \in Activity \bullet$ ⟦(a || b) || c⟧ ≡ ⟦a || (b || c)⟧, **then**

$\forall a,b,c \in Activity \bullet$ ⟦μx.(ε + ((a || b) || c); x)⟧ ≡ ⟦μx.(ε + (a || (b || c)); x)⟧

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ (⟦(a || b) || c⟧ ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ (⟦a || (b || c)⟧ ⊗ t))) -- by tr4

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ ((⟦a || b⟧ // ⟦c⟧) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ((⟦a⟧ // ⟦b || c⟧) ⊗ t))) -- by tp1

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ ((⟦a⟧ // ⟦b⟧ // ⟦c⟧) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ((⟦a⟧ // ⟦b⟧ // ⟦c⟧) ⊗ t))) -- by tp1

B.4.1.6 Congruence in p.1 with the encapsulation

If $\forall a,b,c \in Activity \bullet$ ⟦(a || b) || c⟧ ≡ ⟦a || (b || c)⟧, **then**

$\forall a,b,c \in Activity \bullet$ ⟦{(a || b) || c}$_T$⟧ ≡ ⟦{a || (b || c)}$_T$⟧

⇒ unpack(⟦(a || b) || c⟧) ≡ unpack(⟦a || (b || c)⟧) -- by tu1

⇒ unpack(⟦a || b⟧ // ⟦c⟧) ≡ unpack(⟦a⟧ // ⟦b || c⟧) -- by tp1

⇒ unpack(⟦a⟧ // ⟦b⟧ // ⟦c⟧) ≡ unpack(⟦a⟧ // ⟦b⟧ // ⟦c⟧) -- by tp1

B.4.2 Showing congruence for basic operators in the commutative parallel composition axiom

The commutative parallel composition axiom (p.2) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.4.2.1 Congruence in p.2 with the sequence operator

If $\forall a, b \in Activity \bullet$ ⟦a || b ⟧ ≡ ⟦b || a ⟧, **then**

$\forall a, b, c \in Activity \bullet$ ⟦(a || b); c⟧ ≡ ⟦(b || a); c⟧

⇒ ⟦a || b⟧ ⊗ ⟦c⟧ ≡ ⟦b || a⟧ ⊗ ⟦c⟧ -- by ts1

⇒ (⟦a⟧ // ⟦b⟧) ⊗ ⟦c⟧ ≡ (⟦b⟧ // ⟦a⟧) ⊗ ⟦c⟧ -- by tp1

⇒ (⟦a⟧ // ⟦b⟧) ⊗ ⟦c⟧ ≡ (⟦a⟧ // ⟦b⟧) ⊗ ⟦c⟧ -- by commutativity of //

B.4.2.2 Congruence in p.2 with the selection operator

If $\forall a, b \in Activity \bullet$ ⟦a || b ⟧ ≡ ⟦b || a ⟧, **then**

$\forall a, b, c \in Activity \bullet$ ⟦(a || b) + c⟧ ≡ ⟦(b || a) + c⟧

⇒ {<↓>} ⊗ (⟦a || b⟧ ∪ ⟦c⟧) ≡ {<↓>} ⊗ (⟦b || a⟧ ∪ ⟦c⟧) -- by ta2

⇒ {<↓>} ⊗ ((⟦a⟧ // ⟦b⟧) ∪ ⟦c⟧) ≡ {<↓>} ⊗ ((⟦b⟧ // ⟦a⟧) ∪ ⟦c⟧) -- by tp1

⇒ {<↓>} ⊗ ((⟦a⟧ // ⟦b⟧) ∪ ⟦c⟧) ≡ {<↓>} ⊗ ((⟦a⟧ // ⟦b⟧) ∪ ⟦c⟧)

-- by commutativity of //

B.4.2.3 Congruence in p.2 with the parallel composition operator

If $\forall a, b \in Activity \bullet$ ⟦a || b ⟧ ≡ ⟦b || a ⟧, **then**

$\forall a, b, c \in Activity \bullet$ ⟦(a || b) || c⟧ ≡ ⟦(b || a) || c⟧

⇒ ⟦a || b⟧ // ⟦c⟧ ≡ ⟦b || a⟧ // ⟦c⟧ -- by tp1

⇒ (⟦a⟧ // ⟦b⟧) // ⟦c⟧ ≡ (⟦b⟧ // ⟦a⟧) // ⟦c⟧ -- by tp1

⇒ (⟦a⟧ // ⟦b⟧) // ⟦c⟧ ≡ (⟦a⟧ // ⟦b⟧) // ⟦c⟧ -- by commutativity of //

B.4.2.4 Congruence in p.2 with the until-loop

If $\forall a, b \in Activity \bullet$ ⟦a || b ⟧ ≡ ⟦b || a ⟧, **then**

$\forall a, b \in Activity \bullet [\![\mu x.((a \parallel b); \varepsilon + x)]\!] \equiv [\![\mu x.((b \parallel a); \varepsilon + x)]\!]$

$\Rightarrow \mu t.([\![a \parallel b]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![b \parallel a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.(([\![a]\!] // [\![b]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(([\![b]\!] // [\![a]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tp1

$\Rightarrow \mu t.(([\![a]\!] // [\![b]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(([\![a]\!] // [\![b]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

-- by commutativity of //

B.4.2.5 Congruence in p.2 with the while-loop

If $\forall a, b \in Activity \bullet [\![a \parallel b]\!] \equiv [\![b \parallel a]\!]$, **then**

$\forall a, b \in Activity \bullet [\![\mu x.(\varepsilon + (a \parallel b); x)]\!] \equiv [\![\mu x.(\varepsilon + (b \parallel a); x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a \parallel b]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![b \parallel a]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (([\![a]\!] // [\![b]\!]) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (([\![b]\!] // [\![a]\!]) \otimes t)))$ -- by tp1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (([\![a]\!] // [\![b]\!]) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (([\![a]\!] // [\![b]\!]) \otimes t)))$ -- by commutativity of //

B.4.2.6 Congruence in p.2 with the encapsulation

If $\forall a, b \in Activity \bullet [\![a \parallel b]\!] \equiv [\![b \parallel a]\!]$, **then**

$\forall a, b \in Activity \bullet [\![\{a \parallel b\}_T]\!] \equiv [\![\{b \parallel a\}_T]\!]$

$\Rightarrow unpack([\![a \parallel b]\!]) \equiv unpack([\![b \parallel a]\!])$ -- by tu1

$\Rightarrow unpack([\![a]\!] // [\![b]\!]) \equiv unpack([\![b]\!] // [\![a]\!])$ -- by tp1

$\Rightarrow unpack([\![a]\!] // [\![b]\!]) \equiv unpack([\![a]\!] // [\![b]\!])$ -- by commutativity of //

B.4.3 Showing congruence for basic operators in the right distributivity of concurrency over selection axiom

The right distributivity of concurrency over selection axiom (p.3) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.4.3.1 Congruence in p.3 with the sequence operator

If $\forall a,b,c \in Activity \bullet [\![(a + b) \,||\, c\,]\!] \equiv [\![(a \,||\, c) + (b \,||\, c)]\!]$, **then**

$\forall a,b,c,d \in Activity \bullet [\![((a + b) \,||\, c); d]\!] \equiv [\![((a \,||\, c) + (b \,||\, c)); d]\!]$

$\Rightarrow [\![(a + b) \,||\, c]\!] \otimes [\![d]\!] \equiv [\![(a \,||\, c) + (b \,||\, c)]\!] \otimes [\![d]\!]$ -- by ts1

$\Rightarrow ([\![a + b]\!] \,//\, [\![c]\!]) \otimes [\![d]\!] \equiv [\![(a \,||\, c) + (b \,||\, c)]\!] \otimes [\![d]\!]$ -- by tp1

$\Rightarrow ((\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])) \,//\, [\![c]\!]) \otimes [\![d]\!]$

$\equiv (\{<\downarrow>\} \otimes ([\![a \,||\, c]\!] \cup [\![b \,||\, c]\!])) \otimes [\![d]\!]$ -- by ta2

$\Rightarrow ((\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])) \,//\, [\![c]\!]) \otimes [\![d]\!]$

$\equiv (\{<\downarrow>\} \otimes (([\![a]\!] \,//\, [\![c]\!]) \cup ([\![b]\!] \,//\, [\![c]\!]))) \otimes [\![d]\!]$ -- by tp1

$\Rightarrow \{<\downarrow>\} \otimes (([\![a]\!] \,//\, [\![c]\!]) \cup ([\![b]\!] \,//\, [\![c]\!])) \otimes [\![d]\!]$

$\equiv \{<\downarrow>\} \otimes (([\![a]\!] \,//\, [\![c]\!]) \cup ([\![b]\!] \,//\, [\![c]\!])) \otimes [\![d]\!]$ -- distribution of // over $\cup$

B.4.3.2 Congruence in p.3 with the selection operator

If $\forall a,b,c \in Activity \bullet [\![(a + b) \,||\, c\,]\!] \equiv [\![(a \,||\, c) + (b \,||\, c)]\!]$, **then**

$\forall a,b,c,d \in Activity \bullet [\![((a + b) \,||\, c) + d]\!] \equiv [\![((a \,||\, c) + (b \,||\, c)) + d]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![(a + b) \,||\, c]\!] \cup [\![d]\!])$

$\equiv \{<\downarrow>\} \otimes ([\![(a \,||\, c) + (b \,||\, c)]\!] \cup [\![d]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes (([\![a + b]\!] \,//\, [\![c]\!]) \cup [\![d]\!])$

$\equiv \{<\downarrow>\} \otimes ([\![(a \,||\, c) + (b \,||\, c)]\!] \cup [\![d]\!])$ -- by tp1

$\Rightarrow \{<\downarrow>\} \otimes (((\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])) \,//\, [\![c]\!]) \cup [\![d]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![a \,||\, c]\!] \cup [\![b \,||\, c]\!])) \cup [\![d]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes (((\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])) \,//\, [\![c]\!]) \cup [\![d]\!])$

≡ {<↓>} ⊗ (({<↓>} ⊗ (([[a]] // [[c]]) ∪ ([[b]] // [[c]]))) ∪ [[d]]) -- by tp1

⇒ {<↓>} ⊗ ({<↓>} ⊗ (([[a]] // [[c]]) ∪ ([[b]] // [[c]])) ∪ [[d]])

≡ {<↓>} ⊗ ({<↓>} ⊗ (([[a]] // [[c]]) ∪ ([[b]] // [[c]])) ∪ [[d]])

-- distribution of // over ∪

B.4.3.3 Congruence in p.3 with the parallel composition operator

If $\forall a, b, c \in Activity \bullet$ [[(a + b) || c]] ≡ [[(a || c) + (b || c)]], **then**

$\forall a, b, c, d \in Activity \bullet$ [[((a + b) || c) || d]] ≡ [[((a || c) + (b || c)) || d]]

⇒ [[(a + b) || c]] // [[d]] ≡ [[(a || c) + (b || c)]] // [[d]] -- by tp1

⇒ ([[a + b]] // [[c]]) // [[d]] ≡ [[(a || c) + (b || c)]] // [[d]] -- by tp1

⇒ (({<↓>} ⊗ ([[a]] ∪ [[b]])) // [[c]]) // [[d]]

≡ ({<↓>} ⊗ ([[a || c]] ∪ [[b || c]])) // [[d]] -- by ta2

⇒ (({<↓>} ⊗ ([[a]] ∪ [[b]])) // [[c]]) // [[d]]

≡ ({<↓>} ⊗ (([[a]] // [[c]]) ∪ ([[b]] // [[c]]))) // [[d]] -- by tp1

⇒ {<↓>} ⊗ (([[a]] // [[c]]) ∪ ([[b]] // [[c]])) // [[d]]

≡ {<↓>} ⊗ (([[a]] // [[c]]) ∪ ([[b]] // [[c]])) // [[d]] -- distribution of // over ∪

B.4.3.4 Congruence in p.3 with the until-loop

If $\forall a, b, c \in Activity \bullet$ [[(a + b) || c]] ≡ [[(a || c) + (b || c)]], **then**

$\forall a, b, c \in Activity \bullet$ [[μx.(((a + b) || c); ε + x)]] ≡ [[μx.(((a || c) + (b || c)); ε + x)]]

⇒ μt.([[(a + b) || c]] ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.([[(a || c) + (b || c)]] ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by tr2

⇒ μt.(([[a + b]] // [[c]]) ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.([[(a || c) + (b || c)]] ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) -- by tp1

⇒ μt.((({<↓>} ⊗ ([[a]] ∪ [[b]])) // [[c]]) ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

≡ μt.(({<↓>} ⊗ ([[a || c]] ∪ [[b || c]])) ⊗ ({<↓>} ∪ ({<↓>} ⊗ t)))

-- by ta2

$\Rightarrow \mu t.(\ (\{<\downarrow>\} \otimes (\llbracket a \rrbracket \cup \llbracket b \rrbracket)) \,//\, \llbracket c \rrbracket) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow>\} \otimes ((\llbracket a \rrbracket \,//\, \llbracket c \rrbracket) \cup (\llbracket b \rrbracket \,//\, \llbracket c \rrbracket))) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

-- by tp1

$\Rightarrow \mu t.(\{<\downarrow>\} \otimes ((\llbracket a \rrbracket \,//\, \llbracket c \rrbracket) \cup (\llbracket b \rrbracket \,//\, \llbracket c \rrbracket)) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \otimes ((\llbracket a \rrbracket \,//\, \llbracket c \rrbracket) \cup (\llbracket b \rrbracket \,//\, \llbracket c \rrbracket)) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

-- distribution of // over $\cup$

B.4.3.5 Congruence in p.3 with the while-loop

If $\forall a, b, c \in Activity \bullet \llbracket (a + b) \,||\, c \rrbracket \equiv \llbracket (a \,||\, c) + (b \,||\, c) \rrbracket$, **then**

$\forall a, b, c \in Activity \bullet \llbracket \mu x.(\varepsilon + ((a + b) \,||\, c); x) \rrbracket \equiv \llbracket \mu x.(\varepsilon + ((a \,||\, c) + (b \,||\, c)); x) \rrbracket$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes \llbracket (a + b) \,||\, c \rrbracket \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes \llbracket (a \,||\, c) + (b \,||\, c) \rrbracket \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes\ (\llbracket a + b \rrbracket \,//\, \llbracket c \rrbracket) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes \llbracket (a \,||\, c) + (b \,||\, c) \rrbracket \otimes t)))$ -- by tp1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (\llbracket a \rrbracket \cup \llbracket b \rrbracket)) \,//\, \llbracket c \rrbracket) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes (\llbracket a \,||\, c \rrbracket \cup \llbracket b \,||\, c \rrbracket)) \otimes t)))$

-- by ta2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (\llbracket a \rrbracket \cup \llbracket b \rrbracket)) \,//\, \llbracket c \rrbracket) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes ((\llbracket a \rrbracket \,//\, \llbracket c \rrbracket) \cup (\llbracket b \rrbracket \,//\, \llbracket c \rrbracket))) \otimes t)))$

-- by tp1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes\{<\downarrow>\} \otimes ((\llbracket a \rrbracket \,//\, \llbracket c \rrbracket) \cup (\llbracket b \rrbracket \,//\, \llbracket c \rrbracket)) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes \{<\downarrow>\} \otimes ((\llbracket a \rrbracket \,//\, \llbracket c \rrbracket) \cup (\llbracket b \rrbracket \,//\, \llbracket c \rrbracket)) \otimes t)))$

-- distribution of // over $\cup$

B.4.3.6 Congruence in p.3 with the encapsulation

If $\forall a, b, c \in Activity \bullet \llbracket (a + b) \,||\, c \rrbracket \equiv \llbracket (a \,||\, c) + (b \,||\, c) \rrbracket$, **then**

$\forall a,b,c \in Activity \bullet [\![\{(a + b) \parallel c\}_T]\!] \equiv [\![\{(a \parallel c) + (b \parallel c)\}_T]\!]$

$\Rightarrow$ unpack($[\![(a + b) \parallel c]\!]$) $\equiv$ unpack($[\![(a \parallel c) + (b \parallel c)]\!]$) -- by tu1

$\Rightarrow$ unpack($[\![a + b]\!] // [\![c]\!]$) $\equiv$ unpack($[\![(a \parallel c) + (b \parallel c)]\!]$) -- by tp1

$\Rightarrow$ unpack(($\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])) // [\![c]\!]$)

$\equiv$ unpack($\{<\downarrow>\} \otimes ([\![a \parallel c]\!] \cup [\![b \parallel c]\!])$) -- by ta2

$\Rightarrow$ unpack($\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])) // [\![c]\!]$)

$\equiv$ unpack($\{<\downarrow>\} \otimes (([\![a]\!] // [\![c]\!]) \cup ([\![b]\!] // [\![c]\!]))$) -- by tp1

$\Rightarrow$ unpack($\{<\downarrow>\} \otimes (([\![a]\!] // [\![c]\!]) \cup ([\![b]\!] // [\![c]\!]))$)

$\equiv$ unpack($\{<\downarrow>\} \otimes (([\![a]\!] // [\![c]\!]) \cup ([\![b]\!] // [\![c]\!]))$)

-- distribution of // over $\cup$

B.4.4 Showing congruence for basic operators in the instant synchronisation axiom

The instant synchronisation axiom (p.4) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.4.4.1 Congruence in p.4 with the sequence operator

If $\forall a \in Activity \bullet [\![a \parallel \varepsilon]\!] \equiv [\![a]\!]$, **then**

$\forall a,b \in Activity \bullet [\![(a \parallel \varepsilon); b]\!] \equiv [\![a; b]\!]$

$\Rightarrow [\![a \parallel \varepsilon]\!] \otimes [\![b]\!] \equiv [\![a]\!] \otimes [\![b]\!]$ -- by ts1

$\Rightarrow ([\![a]\!] // [\![\varepsilon]\!]) \otimes [\![b]\!] \equiv [\![a]\!] \otimes [\![b]\!]$ -- by tp1

$\Rightarrow ([\![a]\!] // \{<>\}) \otimes [\![b]\!] \equiv [\![a]\!] \otimes [\![b]\!]$ -- by tb1

$\Rightarrow [\![a]\!] \otimes [\![b]\!] \equiv [\![a]\!] \otimes [\![b]\!]$ -- by di1

B.4.4.2 Congruence in p.4 with the selection operator

If $\forall a \in Activity \bullet [\![a \parallel \varepsilon]\!] \equiv [\![a]\!]$, **then**

$\forall a,b \in Activity \bullet [\![(a \parallel \varepsilon) + b]\!] \equiv [\![a + b]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![a \parallel \varepsilon]\!] \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes ((⟦a⟧ // ⟦\varepsilon⟧) \cup ⟦b⟧) \equiv \{<\downarrow>\} \otimes (⟦a⟧ \cup ⟦b⟧)$ -- by tp1

$\Rightarrow \{<\downarrow>\} \otimes ((⟦a⟧ // \{<>\}) \cup ⟦b⟧) \equiv \{<\downarrow>\} \otimes (⟦a⟧ \cup ⟦b⟧)$ -- by tb1

$\Rightarrow \{<\downarrow>\} \otimes (⟦a⟧ \cup ⟦b⟧) \equiv \{<\downarrow>\} \otimes (⟦a⟧ \cup ⟦b⟧)$ -- by di1

B.4.4.3 Congruence in p.4 with the parallel composition operator

If $\forall a \in Activity \bullet ⟦a \parallel \varepsilon⟧ \equiv ⟦a⟧$, **then**

$\forall a, b \in Activity \bullet ⟦(a \parallel \varepsilon) \parallel b⟧ \equiv ⟦a \parallel b⟧$

$\Rightarrow ⟦a \parallel \varepsilon⟧ // ⟦b⟧ \equiv ⟦a⟧ // ⟦b⟧$ -- by tp1

$\Rightarrow (⟦a⟧ // ⟦\varepsilon⟧) // ⟦b⟧ \equiv ⟦a⟧ // ⟦b⟧$ -- by tp1

$\Rightarrow (⟦a⟧ // \{<>\}) // ⟦b⟧ \equiv ⟦a⟧ // ⟦b⟧$ -- by tb1

$\Rightarrow ⟦a⟧ // ⟦b⟧ \equiv ⟦a⟧ // ⟦b⟧$ -- by di1

B.4.4.4 Congruence in p.4 with the until-loop

If $\forall a \in Activity \bullet ⟦a \parallel \varepsilon⟧ \equiv ⟦a⟧$, **then**

$\forall a \in Activity \bullet ⟦\mu x.((a \parallel \varepsilon); \varepsilon + x)⟧ \equiv ⟦\mu x.(a; \varepsilon + x)⟧$

$\Rightarrow \mu t.(⟦a \parallel \varepsilon⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(⟦a⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.((⟦a⟧ // ⟦\varepsilon⟧) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(⟦a⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tp1

$\Rightarrow \mu t.((⟦a⟧ // \{<>\}) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(⟦a⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tb1

$\Rightarrow \mu t.(⟦a⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(⟦a⟧ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by di1

B.4.4.5 Congruence in p.4 with the while-loop

If $\forall a \in Activity \bullet ⟦a \parallel \varepsilon⟧ \equiv ⟦a⟧$, **then**

$\forall a \in Activity \bullet ⟦\mu x.(\varepsilon + (a \parallel \varepsilon); x)⟧ \equiv ⟦\mu x.(\varepsilon + a; x)⟧$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a \parallel \varepsilon]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (([\![a]\!] // [\![\varepsilon]\!]) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$ -- by tp1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (([\![a]\!] // \{<>\}) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$ -- by tb1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$ -- by di1

B.4.4.6 Congruence in p.4 with the encapsulation

If $\forall a \in Activity \bullet [\![a \parallel \varepsilon]\!] \equiv [\![a]\!]$, **then**

$\forall a \in Activity \bullet [\![\{a \parallel \varepsilon\}_T]\!] \equiv [\![\{a\}_T]\!]$

$\Rightarrow unpack([\![a \parallel \varepsilon]\!]) \equiv unpack([\![a]\!])$ -- by tu1

$\Rightarrow unpack([\![a]\!] // [\![\varepsilon]\!]) \equiv unpack([\![a]\!])$ -- by tp1

$\Rightarrow unpack([\![a]\!] // \{<>\}) \equiv unpack([\![a]\!])$ -- by tb1

$\Rightarrow unpack([\![a]\!]) \equiv unpack([\![a]\!])$ -- by di1

B.4.5 Showing congruence for basic operators in the fail in parallel composition axiom

The fail in parallel composition axiom (p.5) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.4.5.1 Congruence in p.5 with the sequence operator

If $\forall a \in Activity \bullet [\![a \parallel \phi]\!] \equiv [\![\phi]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(a \parallel \phi); b]\!] \equiv [\![\phi; b]\!]$

$\Rightarrow [\![a \parallel \phi]\!] \otimes [\![b]\!] \equiv [\![\phi]\!] \otimes [\![b]\!]$ -- by ts1

$\Rightarrow ([\![a]\!] // [\![\phi]\!]) \otimes [\![b]\!] \equiv [\![\phi]\!] \otimes [\![b]\!]$ -- by tp1

$\Rightarrow ([\![a]\!] // \{<\phi>\}) \otimes [\![b]\!] \equiv \{<\phi>\} \otimes [\![b]\!]$ -- by tb2

$\Rightarrow (\{t_1, t_2, \ldots, t_n\} // \{<\phi>\}) \otimes [\![b]\!] \equiv \{<\phi>\} \otimes [\![b]\!]$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\phi>\} \otimes [\![b]\!] \equiv \{<\phi>\} \otimes [\![b]\!]$ $\bigcup_{i=1}^{n} \{t_i \sim < \phi >\}$

B.4.5.2 Congruence in p.5 with the selection operator

If $\forall a \in Activity \bullet [\![a \parallel \phi]\!] \equiv [\![\phi]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(a \parallel \phi) + b]\!] \equiv [\![\phi + b]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![a \parallel \phi]\!] \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes ([\![\phi]\!] \cup [\![b]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes (([\![a]\!] // [\![\phi]\!]) \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes ([\![\phi]\!] \cup [\![b]\!])$ -- by tp1

$\Rightarrow \{<\downarrow>\} \otimes (([\![a]\!] // \{<\phi>\}) \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes (\{<\phi>\} \cup [\![b]\!])$ -- by tb2

$\Rightarrow \{<\downarrow>\} \otimes ((\{t_1, t_2, \ldots, t_n\} // \{<\phi>\}) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (\{<\phi>\} \cup [\![b]\!])$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\downarrow>\} \otimes (\{<\phi>\} \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes (\{<\phi>\} \cup [\![b]\!])$ $\bigcup_{i=1}^{n} \{t_i \sim < \phi >\}$

B.4.5.3 Congruence in p.5 with the parallel composition operator

If $\forall a \in Activity \bullet [\![a \parallel \phi]\!] \equiv [\![\phi]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(a \parallel \phi) \parallel b]\!] \equiv [\![\phi \parallel b]\!]$

$\Rightarrow [\![a \parallel \phi]\!] // [\![b]\!] \equiv [\![\phi]\!] // [\![b]\!]$ -- by tp1

$\Rightarrow ([\![a]\!] // [\![\phi]\!]) // [\![b]\!] \equiv [\![\phi]\!] // [\![b]\!]$ -- by tp1

$\Rightarrow ([\![a]\!] // \{<\phi>\}) // [\![b]\!] \equiv \{<\phi>\} // [\![b]\!]$ -- by tb2

$\Rightarrow (\{t_1, t_2, \ldots, t_n\} // \{<\phi>\}) // [\![b]\!] \equiv \{<\phi>\} // [\![b]\!]$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\phi>\} // [\![b]\!] \equiv \{<\phi>\} // [\![b]\!]$ $\bigcup_{i=1}^{n} \{t_i \sim < \phi >\}$

B.4.5.4 Congruence in p.5 with the until-loop

If $\forall a \in Activity \bullet [\![a \parallel \phi]\!] \equiv [\![\phi]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.((a \parallel \phi); \varepsilon + x)]\!] \equiv [\![\mu x.(\phi; \varepsilon + x)]\!]$

$\Rightarrow \mu t.([\![a \parallel \phi]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![\phi]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.(([\![a]\!] // [\![\phi]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![\phi]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tp1

$\Rightarrow \mu t.(([\![a]\!] // \{<\phi>\}) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<\phi>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tb2

$\Rightarrow \mu t.((\{t_1, t_2, \ldots, t_n\} // \{<\phi>\}) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<\phi>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \mu t.(\{<\phi>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<\phi>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ $\bigcup_{i=1}^{n} \{t_i \sim <\phi>\}$

B.4.5.5 Congruence in p.5 with the while-loop

If $\forall a \in Activity \bullet [\![a \parallel \phi]\!] \equiv [\![\phi]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.(\varepsilon + (a \parallel \phi); x)]\!] \equiv [\![\mu x.(\varepsilon + \phi; x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a \parallel \phi]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\phi]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (([\![a]\!] // [\![\phi]\!]) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\phi]\!] \otimes t)))$ -- by tp1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (([\![a]\!] // \{<\phi>\}) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\phi>\} \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{t_1, t_2, \ldots, t_n\} // \{<\phi>\}) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\phi>\} \otimes t)))$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\phi>\} \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<\phi>\} \otimes t)))$ $\bigcup_{i=1}^{n} \{t_i \sim <\phi>\}$

B.4.5.6 Congruence in p.5 with the encapsulation

If $\forall a \in Activity \bullet [\![a \parallel \phi]\!] \equiv [\![\phi]\!]$, **then**

$\forall a \in Activity \bullet [\![\{a \parallel \phi\}_T]\!] \equiv [\![\{\phi\}_T]\!]$

$\Rightarrow$ unpack($[\![a \parallel \phi]\!]$) $\equiv$ unpack($[\![\phi]\!]$) -- by tu1

$\Rightarrow$ unpack($[\![a]\!]$ // $[\![\phi]\!]$) $\equiv$ unpack($[\![\phi]\!]$) -- by tp1

$\Rightarrow$ unpack($[\![a]\!]$ // $\{<\phi>\}$) $\equiv$ unpack($\{<\phi>\}$) -- by tb2

$\Rightarrow$ unpack($\{t_1, t_2, \ldots, t_n\}$ // $\{<\phi>\}$)

$\equiv$ unpack($\{<\phi>\}$) Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow$ unpack($\{<\phi>\}$) $\equiv$ unpack($\{<\phi>\}$) $\bigcup_{i=1}^{n} \{t_i \sim < \phi >\}$

B.4.6 Showing congruence for basic operators in the succeed in parallel composition axiom

The succeed in parallel composition axiom (p.6) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.4.6.1 Congruence in p.6 with the sequence operator

If $\forall a \in Activity \bullet [\![a \parallel \sigma]\!] \equiv [\![\sigma]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(a \parallel \sigma); b]\!] \equiv [\![\sigma; b]\!]$

$\Rightarrow [\![a \parallel \sigma]\!] \otimes [\![b]\!] \equiv [\![\sigma]\!] \otimes [\![b]\!]$ -- by ts1

$\Rightarrow ([\![a]\!] // [\![\sigma]\!]) \otimes [\![b]\!] \equiv [\![\sigma]\!] \otimes [\![b]\!]$ -- by tp1

$\Rightarrow ([\![a]\!] // \{<\sigma>\}) \otimes [\![b]\!] \equiv \{<\sigma>\} \otimes [\![b]\!]$ -- by tb2

$\Rightarrow (\{t_1, t_2, \ldots, t_n\} // \{<\sigma>\}) \otimes [\![b]\!] \equiv \{<\sigma>\} \otimes [\![b]\!]$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\sigma>\} \otimes [\![b]\!] \equiv \{<\sigma>\} \otimes [\![b]\!]$ $\bigcup_{i=1}^{n} \{t_i \sim < \phi >\}$

B.4.6.2 Congruence in p.6 with the selection operator

If $\forall a \in Activity \bullet [\![a \parallel \sigma]\!] \equiv [\![\sigma]\!]$, **then**

$\forall a, b \in Activity \bullet [\![(a \parallel \sigma) + b]\!] \equiv [\![\sigma + b]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![a \parallel \sigma]\!] \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes ([\![\sigma]\!] \cup [\![b]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes (([\![a]\!] // [\![\sigma]\!]) \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes ([\![\sigma]\!] \cup [\![b]\!])$ -- by tp1

$\Rightarrow \{<\downarrow>\} \otimes ((\llbracket a \rrbracket \,//\, \{<\sigma>\}) \cup \llbracket b \rrbracket) \equiv \{<\downarrow>\} \otimes (\{<\sigma>\} \cup \llbracket b \rrbracket)$ -- by tb2

$\Rightarrow \{<\downarrow>\} \otimes ((\{t_1, t_2, \ldots, t_n\} \,//\, \{<\sigma>\}) \cup \llbracket b \rrbracket)$

$\equiv \{<\downarrow>\} \otimes (\{<\sigma>\} \cup \llbracket b \rrbracket)$ Let $\llbracket a \rrbracket = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\downarrow>\} \otimes (\{<\sigma>\} \cup \llbracket b \rrbracket) \equiv \{<\downarrow>\} \otimes (\{<\sigma>\} \cup \llbracket b \rrbracket)$ $\bigcup_{i=1}^{n} \{t_i \sim < \sigma >\}$

B.4.6.3 Congruence in p.6 with the parallel composition operator

If $\forall a \in Activity \bullet \llbracket a \parallel \sigma \rrbracket \equiv \llbracket \sigma \rrbracket$, **then**

$\forall a, b \in Activity \bullet \llbracket (a \parallel \sigma) \parallel b \rrbracket \equiv \llbracket \sigma \parallel b \rrbracket$

$\Rightarrow \llbracket a \parallel \sigma \rrbracket \,//\, \llbracket b \rrbracket \equiv \llbracket \sigma \rrbracket \,//\, \llbracket b \rrbracket$ -- by tp1

$\Rightarrow (\llbracket a \rrbracket \,//\, \llbracket \sigma \rrbracket) \,//\, \llbracket b \rrbracket \equiv \llbracket \sigma \rrbracket \,//\, \llbracket b \rrbracket$ -- by tp1

$\Rightarrow (\llbracket a \rrbracket \,//\, \{<\sigma>\}) \,//\, \llbracket b \rrbracket \equiv \{<\sigma>\} \,//\, \llbracket b \rrbracket$ -- by tb2

$\Rightarrow (\{t_1, t_2, \ldots, t_n\} \,//\, \{<\sigma>\}) \,//\, \llbracket b \rrbracket \equiv \{<\sigma>\} \,//\, \llbracket b \rrbracket$ Let $\llbracket a \rrbracket = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \{<\sigma>\} \,//\, \llbracket b \rrbracket \equiv \{<\sigma>\} \,//\, \llbracket b \rrbracket$ $\bigcup_{i=1}^{n} \{t_i \sim < \sigma >\}$

B.4.6.4 Congruence in p.6 with the until-loop

If $\forall a \in Activity \bullet \llbracket a \parallel \sigma \rrbracket \equiv \llbracket \sigma \rrbracket$, **then**

$\forall a \in Activity \bullet \llbracket \mu x.((a \parallel \sigma); \varepsilon + x) \rrbracket \equiv \llbracket \mu x.(\sigma; \varepsilon + x) \rrbracket$

$\Rightarrow \mu t.(\llbracket a \parallel \sigma \rrbracket \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\llbracket \sigma \rrbracket \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.((\llbracket a \rrbracket \,//\, \llbracket \sigma \rrbracket) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\llbracket \sigma \rrbracket \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tp1

$\Rightarrow \mu t.((\llbracket a \rrbracket \,//\, \{<\sigma>\}) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<\sigma>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tb2

$\Rightarrow \mu t.((\{t_1, t_2, \ldots, t_n\} \,//\, \{<\sigma>\}) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<\sigma>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ Let $\llbracket a \rrbracket = \{t_1, t_2, \ldots, t_n\}$

$\Rightarrow \mu t.(\{<\sigma>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

≡ μt.({<σ>} ⊗ ({<↓>} ∪ ({<↓>} ⊗ t))) $\bigcup_{i=1}^{n} \{t_i \sim < \sigma >\}$

B.4.6.5 Congruence in p.6 with the while-loop

If $\forall a \in Activity \bullet [\![a \parallel \sigma]\!] \equiv [\![\sigma]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.(\varepsilon + (a \parallel \sigma); x)]\!] \equiv [\![\mu x.(\varepsilon + \sigma; x)]\!]$

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ ([[a || σ]] ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ([[σ]] ⊗ t))) -- by tr4

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ (([[a]] // [[σ]]) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ([[σ]] ⊗ t))) -- by tp1

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ (([[a]] // {<σ>}) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ({<σ>}⊗ t))) -- by tb2

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ (({t_1, t_2, …, t_n} // {<σ>}) ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ({<σ>}⊗ t))) Let [[a]] = {t_1, t_2, …, t_n}

⇒ μt.({<↓>} ∪ ({<↓>} ⊗ ({<σ>} ⊗ t)))

≡ μt.({<↓>} ∪ ({<↓>} ⊗ ({<σ>}⊗ t))) $\bigcup_{i=1}^{n} \{t_i \sim < \sigma >\}$

B.4.6.6 Congruence in p.6 with the encapsulation

If $\forall a \in Activity \bullet [\![a \parallel \sigma]\!] \equiv [\![\sigma]\!]$, **then**

$\forall a \in Activity \bullet [\![\{a \parallel \sigma\}_T]\!] \equiv [\![\{\sigma\}_T]\!]$

⇒ unpack([[a || σ]]) ≡ unpack([[σ]]) -- by tu1

⇒ unpack([[a]] // [[σ]]) ≡ unpack([[σ]]) -- by tp1

⇒ unpack([[a]] // {<σ>}) ≡ unpack({<σ>}) -- by tb2

⇒ unpack({t_1, t_2, …, t_n} // {<σ>})

≡ unpack({<σ>}) Let [[a]] = {t_1, t_2, …, t_n}

⇒ unpack({<σ>}) ≡ unpack({<σ>}) $\bigcup_{i=1}^{n} \{t_i \sim < \phi >\}$

B.5 Showing congruence for repetition

In this section, congruence for repetition is illustrated for each its axioms. Repetition is formed by only two axioms: unrolling one cycle of until-loop repetition and unrolling one cycle of while-loop repetition.

B.5.1 Showing congruence for basic operators in the unrolling one cycle of until-loop repetition axiom

The unrolling one cycle of until-loop axiom (r.1) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.5.1.1 Congruence in r.1 with the sequence operator

If $\forall a \in Activity \bullet [\![\mu x.(a; \varepsilon + x)]\!] \equiv [\![(a; \varepsilon + \mu x.(a; \varepsilon + x))]\!]$, **then**

$\forall a, b \in Activity \bullet [\![\mu x.(a; \varepsilon + x); b]\!] \equiv [\![(a; \varepsilon + \mu x.(a; \varepsilon + x)); b]\!]$

$\Rightarrow [\![\mu x.(a; \varepsilon + x)]\!] \otimes [\![b]\!] \equiv [\![a; \varepsilon + \mu x.(a; \varepsilon + x)]\!] \otimes [\![b]\!]$ -- by ts1

$\Rightarrow [\![\mu x.(a; \varepsilon + x)]\!] \otimes [\![b]\!] \equiv ([\![a]\!] \otimes [\![\varepsilon + \mu x.(a; \varepsilon + x)]\!]) \otimes [\![b]\!]$ -- by ts1

$\Rightarrow [\![\mu x.(a; \varepsilon + x)]\!] \otimes [\![b]\!]$

$\equiv ([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![\mu x.(a; \varepsilon + x)]\!]))) \otimes [\![b]\!]$ -- by ta2

$\Rightarrow \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t))) \otimes [\![b]\!]$

$\equiv ([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))) \otimes [\![b]\!]$

-- by tr2

$\Rightarrow ([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))) \otimes [\![b]\!]$

$\equiv ([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))) \otimes [\![b]\!]$

-- by tr5

B.5.1.2 Congruence in r.1 with the selection operator

If $\forall a \in Activity \bullet [\![\mu x.(a; \varepsilon + x)]\!] \equiv [\![(a; \varepsilon + \mu x.(a; \varepsilon + x))]\!]$, **then**

$\forall a, b \in Activity \bullet [\![\mu x.(a; \varepsilon + x) + b]\!] \equiv [\![(a; \varepsilon + \mu x.(a; \varepsilon + x)) + b]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![\mu x.(a; \varepsilon + x)]\!] \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ([\![a; \varepsilon + \mu x.(a; \varepsilon + x)]\!] \cup [\![b]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes ([\![\mu x.(a;\ \varepsilon + x)]\!] \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (([\![a]\!] \otimes [\![\varepsilon + \mu x.(a;\ \varepsilon + x)]\!]) \cup [\![b]\!])$ -- by ts1

$\Rightarrow \{<\downarrow>\} \otimes ([\![\mu x.(a;\ \varepsilon + x)]\!] \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![\mu x.(a;\ \varepsilon + x)]\!])))$

$\cup [\![b]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes (\mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t))) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup$

$(\{<\downarrow>\} \otimes t)))))) \cup [\![b]\!])$ -- by tr2

$\Rightarrow \{<\downarrow>\} \otimes (([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t))))))$

$\cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.([\![a]\!] \otimes (\{<\downarrow>\}$

$\cup (\{<\downarrow>\} \otimes t)))))) \cup [\![b]\!])$ -- by tr5

B.5.1.3 Congruence in r.1 with the parallel composition operator

If $\forall a \in Activity \bullet [\![\mu x.(a;\ \varepsilon + x)]\!] \equiv [\![(a;\ \varepsilon + \mu x.(a;\ \varepsilon + x))]\!]$, **then**

$\forall a, b \in Activity \bullet [\![\mu x.(a;\ \varepsilon + x) \parallel b]\!] \equiv [\![(a;\ \varepsilon + \mu x.(a;\ \varepsilon + x)) \parallel b]\!]$

$\Rightarrow [\![\mu x.(a;\ \varepsilon + x)]\!] \,//\, [\![b]\!] \equiv [\![a;\ \varepsilon + \mu x.(a;\ \varepsilon + x)]\!] \,//\, [\![b]\!]$ -- by tp1

$\Rightarrow [\![\mu x.(a;\ \varepsilon + x)]\!] \,//\, [\![b]\!] \equiv ([\![a]\!] \otimes [\![\varepsilon + \mu x.(a;\ \varepsilon + x)]\!]) \,//\, [\![b]\!]$ -- by ts1

$\Rightarrow [\![\mu x.(a;\ \varepsilon + x)]\!] \,//\, [\![b]\!]$

$\equiv ([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![\mu x.(a;\ \varepsilon + x)]\!]))) \,//\, [\![b]\!]$ -- by ta2

$\Rightarrow \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t))) \,//\, [\![b]\!]$

$\equiv ([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))) \,//\, [\![b]\!]$

-- by tr2

$\Rightarrow ([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))) \,//\, [\![b]\!]$

$\equiv\ ([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))))) \,//\, [\![b]\!]$

-- by tr5

B.5.1.4 Congruence in r.1 with the until-loop

If $\forall a \in Activity \bullet [\![\mu x.(a;\ \varepsilon + x)]\!] \equiv [\![a;\ \varepsilon + \mu x.(a;\ \varepsilon + x)]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.(\mu x.(a;\ \varepsilon + x);\ \varepsilon + x)]\!] \equiv [\![\mu x.(\ (a;\ \varepsilon + \mu x.(a;\ \varepsilon + x));\ \varepsilon + x)]\!]$

$\Rightarrow \mu t.([\![\mu x.(a;\ \varepsilon + x)]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![a;\ \varepsilon + \mu x.(a;\ \varepsilon + x)]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.([\![\mu x.(a;\ \varepsilon + x)]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ ([\![a]\!] \otimes [\![\varepsilon + \mu x.(a;\ \varepsilon + x)]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by ts1

$\Rightarrow \mu t.([\![\mu x.(a;\ \varepsilon + x)]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\ [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![\mu x.(a;\ \varepsilon + x)]\!]))\ )$

$\otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by ta2

$\Rightarrow \mu t.(\mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t))) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\ [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t))))))\ )$

$\otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.(\ ([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))\ ))\ )$

$\otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\ [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))\ ))\ )$

$\otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr5

B.5.1.5 Congruence in r.1 with the while-loop

If $\forall a \in Activity \bullet [\![\mu x.(a;\ \varepsilon + x)]\!] \equiv [\![a;\ \varepsilon + \mu x.(a;\ \varepsilon + x)]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.(\varepsilon + \mu x.(a;\ \varepsilon + x);\ x)]\!] \equiv [\![\mu x.(\varepsilon + (a;\ \varepsilon + \mu x.(a;\ \varepsilon + x));\ x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\mu x.(a;\ \varepsilon + x)]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a;\ \varepsilon + \mu x.(a;\ \varepsilon + x)]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\mu x.(a;\ \varepsilon + x)]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ ([\![a]\!] \otimes [\![\varepsilon + \mu x.(a;\ \varepsilon + x)]\!]) \otimes t)))$ -- by ts1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\mu x.(a;\ \varepsilon + x)]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\ [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![\mu x.(a;\ \varepsilon + x)]\!])) \ ) \otimes t)))$ -- by ta2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t))) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\ [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t))))) \ ) \otimes t)))$ -- by tr2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ ([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))) \ ) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\ [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))) \ ) \otimes t)))$ -- by tr5

B.5.1.6 Congruence in r.1 with the encapsulation

If $\forall a \in Activity \bullet [\![\mu x.(a;\ \varepsilon + x)]\!] \equiv [\![(a;\ \varepsilon + \mu x.(a;\ \varepsilon + x))]\!]$, **then**

$\forall a \in Activity \bullet [\![\{\mu x.(a;\ \varepsilon + x)\}_T]\!] \equiv [\![\{a;\ \varepsilon + \mu x.(a;\ \varepsilon + x)\}_T]\!]$

$\Rightarrow unpack([\![\mu x.(a;\ \varepsilon + x)]\!]) \equiv unpack([\![a;\ \varepsilon + \mu x.(a;\ \varepsilon + x)]\!])$ -- by tu1

$\Rightarrow unpack([\![\mu x.(a;\ \varepsilon + x)]\!]) \equiv unpack([\![a]\!] \otimes [\![\varepsilon + \mu x.(a;\ \varepsilon + x)]\!])$ -- by ts1

$\Rightarrow unpack([\![\mu x.(a;\ \varepsilon + x)]\!])$

$\equiv unpack(\ [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![\mu x.(a;\ \varepsilon + x)]\!])) \ )$ -- by ta2

$\Rightarrow unpack(\mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t))) \ )$

$\equiv unpack(\ [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t))))) \ )$ -- by tr2

$\Rightarrow unpack([\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t))))) \ )$

$\equiv\ unpack(\ [\![a]\!] \otimes (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup \mu t.(\ [\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))) \ )$ -- by tr5

B.5.2 Showing congruence for basic operators in the unrolling one cycle of while-loop repetition axiom

The unrolling one cycle of while-loop repetition axiom (r.2) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.5.2.1 Congruence in r.2 with the sequence operator

If $\forall a \in Activity \bullet [\![\mu x.(\varepsilon + a\ ;\ x)]\!] \equiv [\![(\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x))]\!]$, **then**

$\forall a, b \in Activity \bullet [\![\mu x.(\varepsilon + a\ ;\ x);\ b]\!] \equiv [\![(\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x));\ b]\!]$

$\Rightarrow [\![\mu x.(\varepsilon + a\ ;\ x)]\!] \otimes [\![b]\!] \equiv [\![\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x)]\!] \otimes [\![b]\!]$ -- by ts1

$\Rightarrow [\![\mu x.(\varepsilon + a\ ;\ x)]\!] \otimes [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![a;\ \mu x.(\varepsilon + a\ ;\ x)]\!])) \otimes [\![b]\!]$ -- by ta2

$\Rightarrow [\![\mu x.(\varepsilon + a\ ;\ x)]\!] \otimes [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes [\![\mu x.(\varepsilon + a\ ;\ x)]\!]))) \otimes [\![b]\!]$ -- by ts1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))) \otimes [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))))$

$\otimes [\![b]\!]$ -- by tr4

$\Rightarrow (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\ \{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))))) \otimes [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))))) \otimes [\![b]\!]$

-- by tr6

B.5.2.2 Congruence in r.2 with the selection operator

If $\forall a \in Activity \bullet [\![\mu x.(\varepsilon + a\ ;\ x)]\!] \equiv [\![(\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x))]\!]$, **then**

$\forall a, b \in Activity \bullet [\![\mu x.(\varepsilon + a\ ;\ x) + b]\!] \equiv [\![(\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x)) + b]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![\mu x.(\varepsilon + a\ ;\ x)]\!] \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ([\![\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x)]\!] \cup [\![b]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes ([\![\mu x.(\varepsilon + a\ ;\ x)]\!] \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![a;\ \mu x.(\varepsilon + a\ ;\ x)]\!])) \cup [\![b]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes ([\![\mu x.(\varepsilon + a\ ;\ x)]\!] \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes [\![\mu x.(\varepsilon + a\ ;\ x)]\!])\ ))\ \cup [\![b]\!])$

-- by ts1

$\Rightarrow \{<\downarrow>\} \otimes (\mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))\ \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes$

$\mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ ))\ \cup [\![b]\!])$ -- by tr4

$\Rightarrow \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes$

$\mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ ))\ \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes$

$\mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ ))\ \cup [\![b]\!])$ -- by tr6

B.5.2.3 Congruence in r.2 with the parallel composition operator

If $\forall a \in \mathit{Activity} \bullet [\![\mu x.(\varepsilon + a\ ;\ x)]\!] \equiv [\![(\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x))]\!]$, **then**

$\forall a, b \in \mathit{Activity} \bullet [\![\mu x.(\varepsilon + a\ ;\ x) \parallel b]\!] \equiv [\![(\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x)) \parallel b]\!]$

$\Rightarrow [\![\mu x.(\varepsilon + a\ ;\ x)]\!] /\!/ [\![b]\!] \equiv [\![\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x)]\!] /\!/ [\![b]\!]$ -- by tp1

$\Rightarrow [\![\mu x.(\varepsilon + a\ ;\ x)]\!] /\!/ [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![a;\ \mu x.(\varepsilon + a\ ;\ x)]\!])) /\!/ [\![b]\!]$ -- by ta2

$\Rightarrow [\![\mu x.(\varepsilon + a\ ;\ x)]\!] /\!/ [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes [\![\mu x.(\varepsilon + a\ ;\ x)]\!])\ ))\ /\!/ [\![b]\!]$ -- by ts1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))\ /\!/ [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ ))$

$/\!/ [\![b]\!]$ -- by tr4

$\Rightarrow (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\ \{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ ))\ /\!/ [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ ))\ /\!/ [\![b]\!]$

-- by tr6

B.5.2.4 Congruence in r.2 with the until-loop

If $\forall a \in Activity \bullet [\![\mu x.(\varepsilon + a\ ;\ x)]\!] \equiv [\![\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x)]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.(\mu x.(\varepsilon + a\ ;\ x);\ \varepsilon + x)]\!] \equiv [\![\mu x.(\ (\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x));\ \varepsilon + x)]\!]$

$\Rightarrow \mu t.([\![\mu x.(\varepsilon + a\ ;\ x)]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x)]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.([\![\mu x.(\varepsilon + a\ ;\ x)]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![a;\ \mu x.(\varepsilon + a\ ;\ x)]\!])) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

-- by ta2

$\Rightarrow \mu t.([\![\mu x.(\varepsilon + a\ ;\ x)]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes [\![\mu x.(\varepsilon + a\ ;\ x)]\!])\ ))$

$\otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by ts1

$\Rightarrow \mu t.(\mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ ))$

$\otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\ (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\ \{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ ))$

$\otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ ))$

$\otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr6

B.5.2.5 Congruence in r.2 with the while-loop

If $\forall a \in Activity \bullet [\![\mu x.(\varepsilon + a\ ;\ x)]\!] \equiv [\![\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x)]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.(\varepsilon + \mu x.(\varepsilon + a\ ;\ x);\ x)]\!] \equiv [\![\mu x.(\varepsilon + (\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x));\ x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\mu x.(\varepsilon + a\ ;\ x)]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x)]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\mu x.(\varepsilon + a\ ;\ x)]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![a;\ \mu x.(\varepsilon + a\ ;\ x)]\!])) \otimes t)))$

-- by ta2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\mu x.(\varepsilon + a\ ;\ x)]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes [\![\mu x.(\varepsilon + a\ ;\ x)]\!])\ ))$

$\otimes t)))$ -- by ts1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\}$

$\cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ )) \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\ \{<\downarrow>\}$

$\cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ )) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\}$

$\cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ )) \otimes t)))$ -- by tr6

B.5.2.6 Congruence in r.2 with the encapsulation

If $\forall a \in \mathit{Activity} \bullet [\![\mu x.(\varepsilon + a\ ;\ x)]\!] \equiv [\![\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x)]\!]$, **then**

$\forall a \in \mathit{Activity} \bullet [\![\{\mu x.(\varepsilon + a\ ;\ x)\}_T]\!] \equiv [\![\{\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x)\}_T]\!]$

$\Rightarrow \mathrm{unpack}([\![\mu x.(\varepsilon + a\ ;\ x)]\!]) \equiv \mathrm{unpack}([\![\varepsilon + a;\ \mu x.(\varepsilon + a\ ;\ x)]\!])$ -- by tu1

$\Rightarrow \mathrm{unpack}([\![\mu x.(\varepsilon + a\ ;\ x)]\!]\ )$

$\equiv \mathrm{unpack}(\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![a;\ \mu x.(\varepsilon + a\ ;\ x)]\!]))$ -- by ta2

$\Rightarrow \mathrm{unpack}([\![\mu x.(\varepsilon + a\ ;\ x)]\!]\ )$

$\equiv \mathrm{unpack}(\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes [\![\mu x.(\varepsilon + a\ ;\ x)]\!])\ ))$ -- by ts1

$\Rightarrow \mathrm{unpack}(\mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))\ )$

$\equiv \mathrm{unpack}(\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ ))$

-- by tr4

$\Rightarrow \mathrm{unpack}(\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\ \{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ ))$

$\equiv \mathrm{unpack}(\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup ([\![a]\!] \otimes \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t))))\ ))$

-- by tr6

B.6 Showing congruence for encapsulation

This section shows the last group of axioms for the task algebra. Encapsulation is formed by the axioms of vacuous subtask, coincident exit, and vacuous selection. Every axiom is represented in combination with one of the basic operators defined for the task algebra.

B.6.1 Showing congruence for basic operators in the vacuous subtask axiom

The vacuous subtask axiom (e.1) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.6.1.1 Congruence in e.1 with the sequence operator

If $[\![\{\sigma\}_T]\!] \equiv [\![\varepsilon]\!] \equiv [\![\{\varepsilon\}_T]\!]$, **then**

$\forall a \in Activity \bullet [\![\{\sigma\}_T; a]\!] \equiv [\![\varepsilon; a]\!] \equiv [\![\{\varepsilon\}_T; a]\!]$

$\Rightarrow [\![\{\sigma\}_T]\!] \otimes [\![a]\!] \equiv [\![\varepsilon]\!] \otimes [\![a]\!] \equiv [\![\{\varepsilon\}_T]\!] \otimes [\![a]\!]$ -- by ts1

$\Rightarrow \text{unpack}([\![\sigma]\!]) \otimes [\![a]\!] \equiv [\![\varepsilon]\!] \otimes [\![a]\!] \equiv \text{unpack}([\![\varepsilon]\!]) \otimes [\![a]\!]$ -- by tu1

$\Rightarrow \text{unpack}([\![\sigma]\!]) \otimes [\![a]\!] \equiv \{<>\} \otimes [\![a]\!] \equiv \text{unpack}(\{<>\}) \otimes [\![a]\!]$ -- by tb1

$\Rightarrow \text{unpack}(\{<\sigma>\}) \otimes [\![a]\!] \equiv \{<>\} \otimes [\![a]\!] \equiv \text{unpack}(\{<>\}) \otimes [\![a]\!]$ -- by tb2

$\Rightarrow \{<>\} \otimes [\![a]\!] \equiv \{<>\} \otimes [\![a]\!] \equiv \{<>\} \otimes [\![a]\!]$ -- by up1

B.6.1.2 Congruence in e.1 with the selection operator

If $[\![\{\sigma\}_T]\!] \equiv [\![\varepsilon]\!] \equiv [\![\{\varepsilon\}_T]\!]$, **then**

$\forall a \in Activity \bullet [\![\{\sigma\}_T + a]\!] \equiv [\![\varepsilon + a]\!] \equiv [\![\{\varepsilon\}_T + a]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![\{\sigma\}_T]\!] \cup [\![a]\!]) \equiv \{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![a]\!])$

$\equiv \{<\downarrow>\} \otimes ([\![\{\varepsilon\}_T]\!] \cup [\![a]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}([\![\sigma]\!]) \cup [\![a]\!]) \equiv \{<\downarrow>\} \otimes ([\![\varepsilon]\!] \cup [\![a]\!])$

$\equiv \{<\downarrow>\} \otimes (\text{unpack}([\![\varepsilon]\!]) \cup [\![a]\!])$ -- by tu1

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}([\![\sigma]\!]) \cup [\![a]\!]) \equiv \{<\downarrow>\} \otimes (\{<>\} \cup [\![a]\!])$

$\equiv \{<\downarrow>\} \otimes (\text{unpack}(\{<>\}) \cup [\![a]\!])$ -- by tb1

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}(\{<\sigma>\}) \cup [\![a]\!]) \equiv \{<\downarrow>\} \otimes (\{<>\} \cup [\![a]\!])$

$\equiv \{<\downarrow>\} \otimes (\text{unpack}(\{<>\}) \cup [\![a]\!])$ -- by tb2

$\Rightarrow \{<\downarrow>\} \otimes (\{<>\} \cup [\![a]\!]) \equiv \{<\downarrow>\} \otimes (\{<>\} \cup [\![a]\!])$

$\equiv \{<\downarrow>\} \otimes (\{<>\}) \cup [\![a]\!])$ -- by up1

B.6.1.3 Congruence in e.1 with the parallel composition operator

If $[\![\{\sigma\}_T]\!] \equiv [\![\varepsilon]\!] \equiv [\![\{\varepsilon\}_T]\!]$, **then**

$\forall a \in \mathit{Activity} \bullet [\![\{\sigma\}_T \parallel a]\!] \equiv [\![\varepsilon \parallel a]\!] \equiv [\![\{\varepsilon\}_T \parallel a]\!]$

$\Rightarrow [\![\{\sigma\}_T]\!] // [\![a]\!] \equiv [\![\varepsilon]\!] // [\![a]\!] \equiv [\![\{\varepsilon\}_T]\!] // [\![a]\!]$ -- by tp1

$\Rightarrow \text{unpack}([\![\sigma]\!]) // [\![a]\!] \equiv [\![\varepsilon]\!] // [\![a]\!] = \text{unpack}([\![\varepsilon]\!]) // [\![a]\!]$ -- by tu1

$\Rightarrow \text{unpack}([\![\sigma]\!]) // [\![a]\!] \equiv \{<>\} // [\![a]\!] \equiv \text{unpack}(\{<>\}) // [\![a]\!]$ -- by tb1

$\Rightarrow \text{unpack}(\{<\sigma>\}) // [\![a]\!] \equiv \{<>\} // [\![a]\!] \equiv \text{unpack}(\{<>\}) // [\![a]\!]$ -- by tb2

$\Rightarrow \{<>\} // [\![a]\!] \equiv \{<>\} // [\![a]\!] \equiv \{<>\} // [\![a]\!]$ -- by up1

B.6.1.4 Congruence in e.1 with the until-loop

If $[\![\{\sigma\}_T]\!] \equiv [\![\varepsilon]\!] \equiv [\![\{\varepsilon\}_T]\!]$, **then**

$[\![\mu x.(\{\sigma\}_T; \varepsilon + x)]\!] \equiv [\![\mu x.(\varepsilon; \varepsilon + x)]\!] \equiv [\![\mu x.(\{\varepsilon\}_T; \varepsilon + x)]\!]$

$\Rightarrow \mu t.([\![\{\sigma\}_T]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![\varepsilon]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![\{\varepsilon\}_T]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\text{unpack}([\![\sigma]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![\varepsilon]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\text{unpack}([\![\varepsilon]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tu1

$\Rightarrow \mu t.(\text{unpack}([\![\sigma]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\text{unpack}(\{<>\}) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tb1

$\Rightarrow \mu t.(\text{unpack}(\{<\sigma>\}) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(unpack\ (\{<>\}) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(\{<>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\{<>\} \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by up1

B.6.1.5 Congruence in e.1 with the while-loop

If $[\![\{\sigma\}_T]\!] \equiv [\![\varepsilon]\!] \equiv [\![\{\varepsilon\}_T]\!]$, **then**

$[\![\mu x.(\varepsilon + \{\sigma\}_T; x)]\!] \equiv [\![\mu x.(\varepsilon + \varepsilon; x)]\!] \equiv [\![\mu x.(\varepsilon + \{\varepsilon\}_T; x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\{\sigma\}_T]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\{\varepsilon\}_T]\!] \otimes t)))$ -- by tr2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack([\![\sigma]\!]) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\varepsilon]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack\ ([\![\varepsilon]\!]) \otimes t)))$ -- by tu1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack([\![\sigma]\!]) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<>\} \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack\ (\{<>\}) \otimes t)))$ -- by tb1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack(\{<\sigma>\}) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<>\} \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack\ (\{<>\}) \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<>\} \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<>\} \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\{<>\} \otimes t)))$ -- by up1

B.6.1.6 Congruence in e.1 with the encapsulation

If $[\![\{\sigma\}_T]\!] \equiv [\![\varepsilon]\!] \equiv [\![\{\varepsilon\}_T]\!]$, **then**

$[\![\{\{\sigma\}_T\}_T]\!] \equiv [\![\{\varepsilon\}_T]\!] \equiv [\![\{\{\varepsilon\}_T\}_T]\!]$

$\Rightarrow$ unpack($[\![\{\sigma\}_T]\!]$) $\equiv$ unpack($[\![\varepsilon]\!]$) $\equiv$ unpack($[\![\{\varepsilon\}_T]\!]$) -- by tu1

$\Rightarrow$ unpack(unpack($[\![\sigma]\!]$)) $\equiv$ unpack($[\![\varepsilon]\!]$) $\equiv$ unpack(unpack($[\![\varepsilon]\!]$)) -- by tu1

$\Rightarrow$ unpack(unpack($[\![\sigma]\!]$)) $\equiv$ unpack($\{<>\}$) $\equiv$ unpack(unpack($\{<>\}$) -- by tb1

$\Rightarrow$ unpack(unpack($\{<\sigma>\}$)) $\equiv$ unpack($\{<>\}$)

$\equiv$ unpack(unpack($\{<>\}$) -- by tb2

$\Rightarrow$ unpack($\{<>\}$) $\equiv$ unpack($\{<>\}$) $\equiv$ unpack($\{<>\}$) -- by up1

B.6.2 Showing congruence for basic operators in the coincident exit axiom

The coincident exit axiom (e.2) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.6.2.1 Congruence in e.2 with the sequence operator

If $\forall a \in Activity \bullet [\![\{a; \sigma\}_T]\!] \equiv [\![\{a\}_T]\!]$, **then**

$\forall a, b \in Activity \bullet [\![\{a; \sigma\}_T; b]\!] \equiv [\![\{a\}_T; b]\!]$

$\Rightarrow [\![\{a; \sigma\}_T]\!] \otimes [\![b]\!] \equiv [\![\{a\}_T]\!] \otimes [\![b]\!]$ -- by ts1

$\Rightarrow$ unpack($[\![a; \sigma]\!]$) $\otimes$ $[\![b]\!]$ $\equiv$ unpack($[\![a]\!]$) $\otimes$ $[\![b]\!]$ -- by tu1

$\Rightarrow$ unpack($[\![a]\!] \otimes [\![\sigma]\!]$) $\otimes$ $[\![b]\!]$ $\equiv$ unpack($[\![a]\!]$) $\otimes$ $[\![b]\!]$ -- by ts1

$\Rightarrow$ unpack($[\![a]\!] \otimes \{<\sigma>\}$) $\otimes$ $[\![b]\!]$

$\equiv$ unpack($[\![a]\!]$) $\otimes$ $[\![b]\!]$ -- by tb2

$\Rightarrow$ unpack($\{t_1, t_2, \ldots, t_n\} \otimes \{<\sigma>\}$) $\otimes$ $[\![b]\!]$

$\equiv$ unpack($[\![a]\!]$) $\otimes$ $[\![b]\!]$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow [\![a]\!] \otimes [\![b]\!] \equiv$ unpack($[\![a]\!]$) $\otimes$ $[\![b]\!]$ $\bigcup_{i=1}^{n}$ $\{lift\ (t_i \# <\sigma>)\}$

$\Rightarrow [\![a]\!] \otimes [\![b]\!] \equiv$ unpack($\{t_1, t_2, \ldots, t_n\}$) $\otimes$ $[\![b]\!]$

Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$ in unpack($[\![a]\!]$})

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow [\![a]\!] \otimes [\![b]\!] \equiv [\![a]\!] \otimes [\![b]\!]$ $\bigcup_{i=1}^{n}$ $\{lift\ t_i\}$

B.6.2.2 Congruence in e.2 with the selection operator

If $\forall a \in Activity \bullet [\![\{a; \sigma\}_T]\!] \equiv [\![\{a\}_T]\!]$, **then**

$\forall a, b \in Activity \bullet [\![\{a; \sigma\}_T + b]\!] \equiv [\![\{a\}_T + b]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![\{a; \sigma\}_T]\!] \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes ([\![\{a\}_T]\!] \cup [\![b]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}([\![a; \sigma]\!]) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup [\![b]\!])$ -- by tu1

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}([\![a]\!] \otimes [\![\sigma]\!]) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup [\![b]\!])$ -- by ts1

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}([\![a]\!] \otimes \{<\sigma>\}) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup [\![b]\!])$ -- by tb2

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}(\{t_1, t_2, \ldots, t_n\} \otimes \{<\sigma>\}) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup [\![b]\!])$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup [\![b]\!])$ $\bigcup_{i=1}^{n}$ $\{lift\ (t_i \# <\sigma>)\}$

$\Rightarrow \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes (\text{unpack}(\{t_1, t_2, \ldots, t_n\}) \cup [\![b]\!])$

Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$ in unpack($[\![a]\!]$})

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!]) \equiv \{<\downarrow>\} \otimes ([\![a]\!] \cup [\![b]\!])$ $\bigcup_{i=1}^{n}$ $\{lift\ t_i\}$

B.6.2.3 Congruence in e.2 with the parallel composition operator

If $\forall a \in Activity \bullet [\![\{a; \sigma\}_T]\!] \equiv [\![\{a\}_T]\!]$, **then**

$\forall a, b \in Activity \bullet [\![\{a; \sigma\}_T \parallel b]\!] \equiv [\![\{a\}_T \parallel b]\!]$

$\Rightarrow [\![\{a; \sigma\}_T]\!] // [\![b]\!] \equiv [\![\{a\}_T]\!] // [\![b]\!]$ -- by tp1

$\Rightarrow \text{unpack}([\![a; \sigma]\!]) // [\![b]\!] \equiv \text{unpack}([\![a]\!]) // [\![b]\!]$ -- by tu1

$\Rightarrow \text{unpack}([\![a]\!] \otimes [\![\sigma]\!]) // [\![b]\!] \equiv \text{unpack}([\![a]\!]) // [\![b]\!]$ -- by ts1

$\Rightarrow \text{unpack}([\![a]\!] \otimes \{<\sigma>\}) // [\![b]\!]$

$\equiv \text{unpack}([\![a]\!]) // [\![b]\!]$ -- by tb2

$\Rightarrow \text{unpack}(\{t_1, t_2, \ldots, t_n\} \otimes \{<\sigma>\}) // [\![b]\!]$

$\equiv \text{unpack}([\![a]\!]) // [\![b]\!]$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow [\![a]\!] // [\![b]\!] \equiv \text{unpack}([\![a]\!]) // [\![b]\!]$ $\bigcup_{i=1}^{n} \{lift\ (t_i \# <\sigma>)\}$

$\Rightarrow [\![a]\!] // [\![b]\!] \equiv \text{unpack}(\{t_1, t_2, \ldots, t_n\}) // [\![b]\!]$

Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$ in unpack($[\![a]\!]$})

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow [\![a]\!] // [\![b]\!] \equiv [\![a]\!] // [\![b]\!]$ $\bigcup_{i=1}^{n} \{lift\ t_i\}$

B.6.2.4 Congruence in e.2 with the until-loop

If $\forall a \in Activity \bullet [\![\{a; \sigma\}_T]\!] \equiv [\![\{a\}_T]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.(\{a; \sigma\}_T; \varepsilon + x)]\!] \equiv [\![\mu x.(\{a\}_T; \varepsilon + x)]\!]$

$\Rightarrow \mu t.([\![\{a; \sigma\}_T]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![\{a\}_T]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.(\text{unpack}([\![a; \sigma]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\text{unpack}([\![a]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tu1

$\Rightarrow \mu t.(\text{unpack}([\![a]\!] \otimes [\![\sigma]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\text{unpack}([\![a]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by ts1

$\Rightarrow \mu t.(\text{unpack}([\![a]\!] \otimes \{<\sigma>\}) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(unpack([\![a]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(unpack(\{t_1, t_2, \ldots, t_n\} \otimes \{<\sigma>\}) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(unpack([\![a]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(unpack([\![a]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ $\bigcup_{i=1}^{n}$ $\{lift\ (t_i \# <\sigma>)\}$

$\Rightarrow \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(unpack(\{t_1, t_2, \ldots, t_n\}) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$ in $unpack([\![a]\!]\})$

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![a]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ $\bigcup_{i=1}^{n}$ $\{lift\ t_i\}$

B.6.2.5 Congruence in e.2 with the while-loop

If $\forall a \in Activity \bullet [\![\{a; \sigma\}_T]\!] \equiv [\![\{a\}_T]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.(\varepsilon + \{a; \sigma\}_T; x)]\!] \equiv [\![\mu x.(\varepsilon + \{a\}_T; x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\{a; \sigma\}_T]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\{a\}_T]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack([\![a; \sigma]\!]) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack([\![a]\!]) \otimes t)))$ -- by tu1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack([\![a]\!] \otimes [\![\sigma]\!]) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack([\![a]\!]) \otimes t)))$ -- by ts1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack([\![a]\!] \otimes \{<\sigma>\}) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack([\![a]\!]) \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack(\{t_1, t_2, \ldots, t_n\} \otimes \{<\sigma>\}) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack([\![a]\!]) \otimes t)))$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack([\![a]\!]) \otimes t)))$ $\bigcup_{i=1}^{n} \{lift\ (t_i \# <\sigma>)\}$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack(\{t_1, t_2, \ldots, t_n\}) \otimes t)))$

Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$ in $unpack([\![a]\!]\})$

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![a]\!] \otimes t)))$ $\bigcup_{i=1}^{n} \{lift\ t_i\}$

B.6.2.6 Congruence in e.2 with the encapsulation

If $\forall a \in Activity \bullet [\![\{a; \sigma\}_T]\!] \equiv [\![\{a\}_T]\!]$, **then**

$\forall a \in Activity \bullet [\![\{\{a; \sigma\}_T\}_T]\!] \equiv [\![\{\{a\}_T\}_T]\!]$

$\Rightarrow unpack([\![\{a; \sigma\}_T]\!]) \equiv unpack([\![\{a\}_T]\!])$ -- by tu1

$\Rightarrow unpack(unpack([\![a; \sigma]\!])) \equiv unpack(unpack([\![a]\!]))$ -- by tu1

$\Rightarrow unpack(unpack([\![a]\!] \otimes [\![\sigma]\!])) \equiv unpack(unpack([\![a]\!]))$ -- by ts1

$\Rightarrow unpack(unpack([\![a]\!] \otimes \{<\sigma>\})) \equiv unpack(unpack([\![a]\!]))$ -- by tb2

$\Rightarrow unpack(unpack(\{t_1, t_2, \ldots, t_n\} \otimes \{<\sigma>\}))$

$\equiv unpack(unpack([\![a]\!]))$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow unpack([\![a]\!]) \equiv unpack(unpack([\![a]\!]))$ $\bigcup_{i=1}^{n} \{lift\ (t_i \# <\sigma>)\}$

$\Rightarrow unpack([\![a]\!]) \equiv unpack(unpack(\{t_1, t_2, \ldots, t_n\}))$

Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$ in $unpack([\![a]\!]\})$

where $t_1 \neq <\sigma>$, $t_2 \neq <\sigma>$, …, $t_n \neq <\sigma>$

$\Rightarrow$ unpack(⟦a⟧) ≡ unpack(⟦a⟧) $\bigcup_{i=1}^{n} \{lift\ t_i\}$

B.6.3 Showing congruence for basic operators in the vacuous selection axiom

The vacuous selection axiom (e.3) is demonstrated in this section for the binary operators of sequence, selection, and parallel composition; as well as for the repetition structures (while- and until-loop) and the encapsulation.

B.6.3.1 Congruence in e.3 with the sequence operator

If $\forall a \in Activity \bullet$ ⟦{a + σ}$_T$⟧ ≡ ⟦{a}$_T$ + ε⟧, **then**

$\forall a, b \in Activity \bullet$ ⟦{a + σ}$_T$; b⟧ ≡ ⟦({a}$_T$ + ε); b⟧

$\Rightarrow$ ⟦{a + σ}$_T$⟧ ⊗ ⟦b⟧ ≡ ⟦{a}$_T$ + ε⟧ ⊗ ⟦b⟧ -- by ts1

$\Rightarrow$ unpack(⟦a + σ⟧) ⊗ ⟦b⟧ ≡ ⟦{a}$_T$ + ε⟧ ⊗ ⟦b⟧ -- by tu1

$\Rightarrow$ unpack({<↓>} ⊗ (⟦a⟧ ∪ ⟦σ⟧)) ⊗ ⟦b⟧

≡ ({<↓>} ⊗ (⟦{a}$_T$⟧ ∪ ⟦ε⟧)) ⊗ ⟦b⟧ -- by ta2

$\Rightarrow$ unpack({<↓>} ⊗ (⟦a⟧ ∪ ⟦σ⟧)) ⊗ ⟦b⟧

≡ ({<↓>} ⊗ (unpack(⟦a⟧) ∪ ⟦ε⟧)) ⊗ ⟦b⟧ -- by tu1

$\Rightarrow$ unpack({<↓>} ⊗ (⟦a⟧ ∪ ⟦σ⟧)) ⊗ ⟦b⟧

≡ ({<↓>} ⊗ (unpack(⟦a⟧) ∪ {<>})) ⊗ ⟦b⟧ -- by tb1

$\Rightarrow$ unpack({<↓>} ⊗ (⟦a⟧ ∪ {<σ>})) ⊗ ⟦b⟧

≡ ({<↓>} ⊗ (unpack(⟦a⟧) ∪ {<>})) ⊗ ⟦b⟧ -- by tb2

$\Rightarrow$ unpack(({<↓>} ⊗ ⟦a⟧) ∪ ({<↓>} ⊗ {<σ>})) ⊗ ⟦b⟧

≡ ({<↓>} ⊗ (unpack(⟦a⟧) ∪ {<>})) ⊗ ⟦b⟧

-- by distribution of ⊗ over union

$\Rightarrow$ unpack(({<↓>} ⊗ ⟦a⟧) ∪ {<↓, σ>}) ⊗ ⟦b⟧

≡ ({<↓>} ⊗ (unpack(⟦a⟧) ∪ {<>})) ⊗ ⟦b⟧ -- by cp1

$\Rightarrow$ (unpack ({<↓>} ⊗ ⟦a⟧) ∪ unpack({<↓, σ>})) ⊗ ⟦b⟧

$\equiv \{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\}) \otimes [\![b]\!]$

-- by distribution of *unpack* over union

$\Rightarrow (\text{unpack}(\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup \text{unpack}(\{<\downarrow, \sigma>\})) \otimes [\![b]\!]$

$\equiv \{<\downarrow>\} \otimes (\text{unpack}(\{t_1, t_2, \ldots, t_n\}) \cup \{<>\}) \otimes [\![b]\!]$

Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow (\text{unpack}(\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup \{<\downarrow>\}) \otimes [\![b]\!]$

$\equiv \{<\downarrow>\} \otimes (\{t_1, t_2, \ldots, t_n\} \cup \{<>\}) \otimes [\![b]\!]$ -- by up1

$\Rightarrow (\text{unpack}(\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup \{<\downarrow>\}) \otimes [\![b]\!]$

$\equiv ((\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup (\{<\downarrow>\} \otimes \{<>\})) \otimes [\![b]\!]$

-- by distribution of $\otimes$ over union

$\Rightarrow (\text{unpack}(\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\}) \cup \{<\downarrow>\}) \otimes [\![b]\!]$

$\equiv (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\}) \otimes [\![b]\!]$ -- by cp1

$\Rightarrow (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\}) \otimes [\![b]\!]$

$\equiv (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\}) \otimes [\![b]\!]$

$\bigcup_{i=1}^{n} \{lift <\downarrow t_i>\}$

B.6.3.2 Congruence in e.3 with the selection operator

If $\forall a \in Activity \bullet [\![\{a + \sigma\}_T]\!] \equiv [\![\{a\}_T + \varepsilon]\!]$, **then**

$\forall a, b \in Activity \bullet [\![\{a + \sigma\}_T + b]\!] \equiv [\![(\{a\}_T + \varepsilon) + b]\!]$

$\Rightarrow \{<\downarrow>\} \otimes ([\![\{a + \sigma\}_T]\!] \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ([\![\{a\}_T + \varepsilon]\!] \cup [\![b]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}([\![a + \sigma]\!]) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ([\![\{a\}_T + \varepsilon]\!] \cup [\![b]\!])$ -- by tu1

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes ([\![\{a\}_T]\!] \cup [\![\varepsilon]\!])) \cup [\![b]\!])$ -- by ta2

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup [\![\varepsilon]\!])) \cup [\![b]\!])$ -- by tu1

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\})) \cup [\![b]\!])$ -- by tb1

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup \{<\sigma>\})) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\})) \cup [\![b]\!])$ -- by tb2

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}((\{<\downarrow>\} \otimes [\![a]\!]) \cup (\{<\downarrow>\} \otimes \{<\sigma>\})) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\})) \cup [\![b]\!])$

-- by distribution of $\otimes$ over union

$\Rightarrow \{<\downarrow>\} \otimes (\text{unpack}((\{<\downarrow>\} \otimes [\![a]\!]) \cup \{<\downarrow, \sigma>\}) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\})) \cup [\![b]\!])$ -- by cp1

$\Rightarrow \{<\downarrow>\} \otimes ((\text{unpack} (\{<\downarrow>\} \otimes [\![a]\!]) \cup \text{unpack}(\{<\downarrow, \sigma>\})) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\}) \cup [\![b]\!])$

-- by distribution of *unpack* over union

$\Rightarrow \{<\downarrow>\} \otimes ((\text{unpack} (\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup \text{unpack}(\{<\downarrow, \sigma>\})) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes (\text{unpack}(\{t_1, t_2, \ldots, t_n\}) \cup \{<>\}) \cup [\![b]\!])$

Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow \{<\downarrow>\} \otimes ((\text{unpack} (\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup \{<\downarrow>\}) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (\{<\downarrow>\} \otimes (\{t_1, t_2, \ldots, t_n\} \cup \{<>\}) \cup [\![b]\!])$ -- by up1

$\Rightarrow \{<\downarrow>\} \otimes ((\text{unpack} (\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup \{<\downarrow>\}) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (((\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup (\{<\downarrow>\} \otimes \{<>\})) \cup [\![b]\!])$

-- by distribution of $\otimes$ over union

$\Rightarrow \{<\downarrow>\} \otimes (\ (\text{unpack}\ (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\}) \cup \{<\downarrow>\}\ ) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (\ (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\})\ \cup [\![b]\!])$ -- by cp1

$\Rightarrow \{<\downarrow>\} \otimes (\ (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\}\ ) \cup [\![b]\!])$

$\equiv \{<\downarrow>\} \otimes (\ (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\})\ \cup [\![b]\!])$

$\bigcup_{i=1}^{n} \{lift <\downarrow t_i>\}$

B.6.3.3 Congruence in e.3 with the parallel composition operator

If $\forall a \in Activity \bullet [\![\{a + \sigma\}_T]\!] \equiv [\![\{a\}_T + \varepsilon]\!]$, **then**

$\forall a, b \in Activity \bullet [\![\{a + \sigma\}_T \,||\, b]\!] \equiv [\![(\{a\}_T + \varepsilon) \,||\, b]\!]$

$\Rightarrow [\![\{a + \sigma\}_T]\!] \,//\, [\![b]\!] \equiv [\![\{a\}_T + \varepsilon]\!] \,//\, [\![b]\!]$ -- by tp1

$\Rightarrow \text{unpack}([\![a + \sigma]\!]) \,//\, [\![b]\!] \equiv [\![\{a\}_T + \varepsilon]\!] \,//\, [\![b]\!]$ -- by tu1

$\Rightarrow \text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])) \,//\, [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes ([\![\{a\}_T]\!] \cup [\![\varepsilon]\!])) \,//\, [\![b]\!]$ -- by ta2

$\Rightarrow \text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])) \,//\, [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup [\![\varepsilon]\!])) \,//\, [\![b]\!]$ -- by tu1

$\Rightarrow \text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])) \,//\, [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\})) \,//\, [\![b]\!]$ -- by tb1

$\Rightarrow \text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup \{<\sigma>\})) \,//\, [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\})) \,//\, [\![b]\!]$ -- by tb2

$\Rightarrow \text{unpack}((\{<\downarrow>\} \otimes [\![a]\!]) \cup (\{<\downarrow>\} \otimes \{<\sigma>\})) \,//\, [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\})) \,//\, [\![b]\!]$

-- by distribution of $\otimes$ over union

$\Rightarrow \text{unpack}((\{<\downarrow>\} \otimes [\![a]\!]) \cup \{<\downarrow, \sigma>\}) \,//\, [\![b]\!]$

$\equiv (\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\})) \,//\, [\![b]\!]$ -- by cp1

$\Rightarrow (\text{unpack}\ (\{<\downarrow>\} \otimes [\![a]\!]) \cup \text{unpack}(\{<\downarrow, \sigma>\}\ )) \,//\, [\![b]\!]$

$\equiv \{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\})$ // $[\![b]\!]$

-- by distribution of *unpack* over union

$\Rightarrow (\text{unpack}(\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup \text{unpack}(\{<\downarrow, \sigma>\})$ // $[\![b]\!]$

$\equiv \{<\downarrow>\} \otimes (\text{unpack}(\{t_1, t_2, \ldots, t_n\}) \cup \{<>\})$ // $[\![b]\!]$

Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow (\text{unpack}(\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup \{<\downarrow>\})$ // $[\![b]\!]$

$\equiv \{<\downarrow>\} \otimes (\{t_1, t_2, \ldots, t_n\} \cup \{<>\})$ // $[\![b]\!]$ -- by up1

$\Rightarrow (\text{unpack}(\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup \{<\downarrow>\})$ // $[\![b]\!]$

$\equiv ((\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup (\{<\downarrow>\} \otimes \{<>\}))$ // $[\![b]\!]$

-- by distribution of $\otimes$ over union

$\Rightarrow (\text{unpack}(\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\}) \cup \{<\downarrow>\})$ // $[\![b]\!]$

$\equiv (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\})$ // $[\![b]\!]$ -- by cp1

$\Rightarrow (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\})$ // $[\![b]\!]$

$\equiv (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\})$ // $[\![b]\!]$

$$\bigcup_{i=1}^{n} \{lift <\downarrow t_i>\}$$

B.6.3.4 Congruence in e.3 with the until-loop

If $\forall a \in Activity \bullet [\![\{a + \sigma\}_T]\!] \equiv [\![\{a\}_T + \varepsilon]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.(\{a + \sigma\}_T; \varepsilon + x)]\!] \equiv [\![\mu x.((\{a\}_T + \varepsilon); \varepsilon + x)]\!]$

$\Rightarrow \mu t.([\![\{a + \sigma\}_T]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![\{a\}_T + \varepsilon]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tr2

$\Rightarrow \mu t.(\text{unpack}([\![a + \sigma]\!]) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.([\![\{a\}_T + \varepsilon]\!] \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by tu1

$\Rightarrow \mu t.(\text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow>\} \otimes ([\![\{a\}_T]\!] \cup [\![\varepsilon]\!]))\ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t))$ -- by ta2

$\Rightarrow \mu t.(\text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!]))\ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup [\![\varepsilon]\!])) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

-- by tu1

$\Rightarrow \mu t.(\text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\}))\ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

-- by tb1

$\Rightarrow \mu t.(\text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup \{<\sigma>\}))\ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\})) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

-- by tb2

$\Rightarrow \mu t.(\text{unpack}((\{<\downarrow>\} \otimes [\![a]\!]) \cup (\{<\downarrow>\} \otimes \{<\sigma>\}))\ \otimes (\{<\downarrow>\}$

$\cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\}))\ \otimes (\{<\downarrow>\}$

$\cup (\{<\downarrow>\} \otimes t)))$ -- by distribution of $\otimes$ over union

$\Rightarrow \mu t.(\text{unpack}((\{<\downarrow>\} \otimes [\![a]\!]) \cup \{<\downarrow, \sigma>\})\ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\}))\ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

-- by cp1

$\Rightarrow \mu t.(\ (\text{unpack}\ (\{<\downarrow>\} \otimes [\![a]\!]) \cup \text{unpack}(\{<\downarrow, \sigma>\}\ )) \otimes (\{<\downarrow>\}$

$\cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup \{<>\}))\ \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

-- by distribution of *unpack* over union

$\Rightarrow \mu t.(\ (\text{unpack}\ (\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup \text{unpack}(\{<\downarrow, \sigma>\}\ ))$

$\otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.((\{<\downarrow>\} \otimes (\text{unpack}(\{t_1, t_2, \ldots, t_n\}) \cup \{<>\})\ )$

$\otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

where $t_1 \neq <\sigma>$, $t_2 \neq <\sigma>$, …, $t_n \neq <\sigma>$

$\Rightarrow \mu t.(\ (\text{unpack}\ (\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup \{<\downarrow>\}\ )$

$\otimes\ (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.((\{<\downarrow>\} \otimes (\{t_1, t_2, \ldots, t_n\} \cup \{<>\}))\ \otimes (\{<\downarrow>\}$

$\cup\ (\{<\downarrow>\} \otimes t)))$ -- by up1

$\Rightarrow \mu t.(\ (\text{unpack}\ (\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup \{<\downarrow>\}\ )$

$\otimes\ (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ ((\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup (\{<\downarrow>\} \otimes \{<>\}))$

$\otimes\ (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by distribution of $\otimes$ over union

$\Rightarrow \mu t.(\ (\text{unpack}\ (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\}) \cup \{<\downarrow>\}\ )$

$\otimes\ (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\})$

$\otimes\ (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$ -- by cp1

$\Rightarrow \mu t.(\ (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\}\ ) \otimes (\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes t)))$

$\equiv \mu t.(\ (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\})\ \otimes (\{<\downarrow>\}$

$\cup\ (\{<\downarrow>\} \otimes t)))$ $\qquad \bigcup_{i=1}^{n}\ \{lift <\downarrow t_i>\}$

B.6.3.5 Congruence in e.3 with the while-loop

If $\forall a \in Activity \bullet [\![\{a + \sigma\}_T]\!] \equiv [\![\{a\}_T + \varepsilon]\!]$, **then**

$\forall a \in Activity \bullet [\![\mu x.(\varepsilon + \{a + \sigma\}_T; x)]\!] \equiv [\![\mu x.(\varepsilon + (\{a\}_T + \varepsilon); x)]\!]$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\{a + \sigma\}_T]\!] \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\{a\}_T + \varepsilon]\!] \otimes t)))$ -- by tr4

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\text{unpack}([\![a + \sigma]\!])\ \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ([\![\{a\}_T + \varepsilon]\!] \otimes t)))$ -- by tu1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (\ (\{<\downarrow>\} \otimes ([\![\{a\}_T]\!] \cup [\![\varepsilon]\!])) \otimes t)))$ -- by ta2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (unpack([\![a]\!])$

$\cup [\![\varepsilon]\!])) \otimes t)))$ -- by tu1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes$

$(unpack([\![a]\!]) \cup \{<>\})) \otimes t)))$ -- by tb1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack(\{<\downarrow>\} \otimes ([\![a]\!] \cup \{<\sigma>\})) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (unpack([\![a]\!])$

$\cup \{<>\})) \otimes t)))$ -- by tb2

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack((\{<\downarrow>\} \otimes [\![a]\!])$

$\cup (\{<\downarrow>\} \otimes \{<\sigma>\})) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (unpack([\![a]\!]) \cup \{<>\})) \otimes t)))$

-- by distribution of $\otimes$ over union

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (unpack((\{<\downarrow>\} \otimes [\![a]\!]) \cup \{<\downarrow, \sigma>\}) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (unpack([\![a]\!]) \cup \{<>\})) \otimes t)))$

-- by cp1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((unpack (\{<\downarrow>\} \otimes [\![a]\!])$

$\cup unpack(\{<\downarrow, \sigma>\})) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (unpack([\![a]\!]) \cup \{<>\})) \otimes t)))$

-- by distribution of *unpack* over union

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((unpack (\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\})$

$\cup unpack(\{<\downarrow, \sigma>\})) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (unpack(\{t_1, t_2, \ldots, t_n\})$

$\cup \{<>\})) \otimes t)))$ Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\text{unpack } (\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\})$

$\cup \{<\downarrow>\}) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{<\downarrow>\} \otimes (\{t_1, t_2, \ldots, t_n\}$

$\cup \{<>\})) \otimes t)))$ -- by up1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\text{unpack } (\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\})$

$\cup \{<\downarrow>\}) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes (((\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}) \cup (\{<\downarrow>\}$

$\otimes \{<>\})) \otimes t)))$ -- by distribution of $\otimes$ over union

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\text{unpack } (\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\})$

$\cup \{<\downarrow>\}) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\}$

$\cup \{<\downarrow>\}) \otimes t)))$ -- by cp1

$\Rightarrow \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\}) \otimes t)))$

$\equiv \mu t.(\{<\downarrow>\} \cup (\{<\downarrow>\} \otimes ((\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\}$

$\cup \{<\downarrow>\}) \otimes t)))$ $\bigcup_{i=1}^{n} \{lift <\downarrow t_i>\}$

B.6.3.6 Congruence in e.3 with the encapsulation

If $\forall a \in Activity \bullet [\![\{a + \sigma\}_T]\!] \equiv [\![\{a\}_T + \varepsilon]\!]$, **then**

$\forall a \in Activity \bullet [\![\{\{a + \sigma\}_T\}_T]\!] \equiv [\![\{\{a\}_T + \varepsilon\}_T]\!]$

$\Rightarrow \text{unpack}([\![\{a + \sigma\}_T]\!]) \equiv \text{unpack}([\![\{a\}_T + \varepsilon]\!])$ -- by tu1

$\Rightarrow \text{unpack}(\text{unpack}([\![a + \sigma]\!])) \equiv \text{unpack}([\![\{a\}_T + \varepsilon]\!])$ -- by tu1

$\Rightarrow \text{unpack}(\text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])))$

$\equiv \text{unpack}(\{<\downarrow>\} \otimes ([\![\{a\}_T]\!] \cup [\![\varepsilon]\!]))$ -- by ta2

$\Rightarrow \text{unpack}(\text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])))$

$\equiv \text{unpack}(\{<\downarrow>\} \otimes (\text{unpack}([\![a]\!]) \cup [\![\varepsilon]\!]))$ -- by tu1

$\Rightarrow \text{unpack}(\text{unpack}(\{<\downarrow>\} \otimes ([\![a]\!] \cup [\![\sigma]\!])))$

$\equiv$ unpack($\{<\downarrow>\} \otimes$ (unpack($[\![a]\!]$) $\cup$ $\{<>\}$)) -- by tb1

$\Rightarrow$ unpack(unpack($\{<\downarrow>\} \otimes ([\![a]\!] \cup \{<\sigma>\})$))

$\equiv$ unpack($\{<\downarrow>\} \otimes$ (unpack($[\![a]\!]$) $\cup$ $\{<>\}$)) -- by tb2

$\Rightarrow$ unpack(unpack(($\{<\downarrow>\} \otimes [\![a]\!]$) $\cup$ ($\{<\downarrow>\} \otimes \{<\sigma>\}$)))

$\equiv$ unpack($\{<\downarrow>\} \otimes$ (unpack($[\![a]\!]$) $\cup$ $\{<>\}$))

-- by distribution of $\otimes$ over union

$\Rightarrow$ unpack(unpack(($\{<\downarrow>\} \otimes [\![a]\!]$) $\cup$ $\{<\downarrow, \sigma>\}$))

$\equiv$ unpack($\{<\downarrow>\} \otimes$ (unpack($[\![a]\!]$) $\cup$ $\{<>\}$)) -- by cp1

$\Rightarrow$ unpack(unpack ($\{<\downarrow>\} \otimes [\![a]\!]$) $\cup$ unpack($\{<\downarrow, \sigma>\}$))

$\equiv$ unpack($\{<\downarrow>\} \otimes$ (unpack($[\![a]\!]$) $\cup$ $\{<>\}$))

-- by distribution of *unpack* over union

$\Rightarrow$ unpack(unpack ($\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}$) $\cup$ unpack($\{<\downarrow, \sigma>\}$))

$\equiv$ unpack($\{<\downarrow>\} \otimes$ (unpack($\{t_1, t_2, \ldots, t_n\}$) $\cup$ $\{<>\}$))

Let $[\![a]\!] = \{t_1, t_2, \ldots, t_n\}$

where $t_1 \neq <\sigma>, t_2 \neq <\sigma>, \ldots, t_n \neq <\sigma>$

$\Rightarrow$ unpack(unpack ($\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}$) $\cup$ $\{<\downarrow>\}$)

$\equiv$ unpack($\{<\downarrow>\} \otimes$ ($\{t_1, t_2, \ldots, t_n\} \cup \{<>\}$)) -- by up1

$\Rightarrow$ unpack(unpack ($\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}$) $\cup$ $\{<\downarrow>\}$)

$\equiv$ unpack(($\{<\downarrow>\} \otimes \{t_1, t_2, \ldots, t_n\}$) $\cup$ ($\{<\downarrow>\} \otimes \{<>\}$))

-- by distribution of $\otimes$ over union

$\Rightarrow$ unpack(unpack ($\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\}$) $\cup$ $\{<\downarrow>\}$)

$\equiv$ unpack($\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\}$) -- by cp1

$\Rightarrow$ unpack($\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\}$)

$\equiv$ unpack($\{<\downarrow t_1>, <\downarrow t_2>, \ldots, <\downarrow t_n>\} \cup \{<\downarrow>\}$) $\quad \bigcup_{i=1}^{n} \quad \{lift <\downarrow t_i>\}$

B.7 Summary

The previous chapter described the soundness of the axioms for the task algebra illustrated in the chapter 4. The trace semantics from the chapter 5 and basic properties explained in the Appendix A were used to prove the soundness of the axioms. In the present chapter, the congruence properties were demonstrated for axioms of the algebra, by combining the equivalences with each basic operator defined in the task algebra. An implementation of the task algebra will be presented in the next chapter.

Appendix C:
Source Code

The Haskell source code of the software mentioned in Chapter 8 is presented in this appendix. Initially, the code of the Task Algebra is depicted. Section2 shows the code for the LTL implementation. Finally, section 3 presents the code for the CTL implementation.

C.1 Task Algebra

The parser for the Task Algebra expressions was written with the Happy program. Happy is a parser generator software for Haskell [154]. The lexical analyser was hand-written in Haskell and included in the source code of Happy.

```
{
-- Simple task abstract syntax version 1

module MainTaskAlgebra where

import Char
import System (getArgs)
-- Data types to represent the parsed expression are in
Traces.hs
import Traces
import Data.Set as Set

}

%name taskAlgebraParser
%tokentype { Token }

%token
      simple            { TokenSimple $$ }
      taskName          { TokenTaskName $$ }
      'let'             { TokenLet }
      'Epsilon'         { TokenEpsilon }
      'Phi'             { TokenFail }
      'Sigma'           { TokenSucceed }
      'Mu'              { TokenMu }
      '+'               { TokenSelection }
      ';'               { TokenSequence }
```

```
      '||'              { TokenParallel }
      '('               { TokenOB }
      ')'               { TokenCB }
      '{'               { TokenOEnc }
      '}'               { TokenCEnc }
      '.'               { TokenDot }
      '='               { TokenNewTask }

%right ';' '+' '||'
%%

Model : Activity            { $1 }
      | CompoundTask Model  { Model $1 $2 }

CompoundTask :
      'let' taskName '=' Encapsulation { CompoundTask $2 $4 }

Encapsulation:
      '{' Activity '}'      { Task (Encapsulation $2) }

Activity :
      Activity ';' Activity { Sequence $1 $3 }
      | Activity '+' Activity    { Selection $1 $3 }
      | Activity '||' Activity   { Parallel $1 $3 }
      --  Until-loop
      | 'Mu' '.' simple '(' Activity ';'
      'Epsilon' '+' simple ')'   { UntilLoop $5 (Simple $3)
(Simple $9) }
      -- While-loop
      |'Mu' '.' simple '(' 'Epsilon' '+' Activity';' simple ')'
                 { WhileLoop $7 (Simple $3) (Simple $9) }
      | '(' Activity ')'    { Task (Brackets $2) }
      | Encapsulation       { $1 }
      | 'Epsilon'           { Epsilon }
      | 'Phi'               { Fail }
      | 'Sigma'             { Succeed }
      | simple              { Task (Simple $1) }
      | taskName            { Task (Compound $1) }

{
happyError :: [Token] -> a
happyError _ = error "Parse error"

-- Token definition
data Token
      = TokenSimple String
      | TokenTaskName String
      | TokenSelection
      | TokenSequence
      | TokenParallel
      | TokenEpsilon
      | TokenMu
      | TokenFail
      | TokenSucceed
      | TokenOEnc
      | TokenCEnc
```

```
      | TokenOB
      | TokenCB
      | TokenDot
      | TokenNewTask
      | TokenLet
 deriving Show

-- a simple lexer that returns this data structure.

lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
      | isSpace  c = lexer cs
      | isUpper  c = lexTask (c:cs)
      | isLower  c = lexTaskAt (c:cs)
lexer (';':cs) = TokenSequence : lexer cs
lexer ('+':cs) = TokenSelection : lexer cs
lexer ('{':cs) = TokenOEnc : lexer cs
lexer ('}':cs) = TokenCEnc : lexer cs
lexer ('(':cs) = TokenOB : lexer cs
lexer (')':cs) = TokenCB : lexer cs
lexer ('.':cs) = TokenDot : lexer cs
lexer ('=':cs) = TokenNewTask : lexer cs
lexer ('|':cs) = lexCon cs

lexCon ('|':cs) = TokenParallel : lexer cs

-- Compound Tasks, Epsilon, Exit and Mu begin with uppercase
lexTask cs =
      case span isAlphaNum cs of
            ("Epsilon",rest) ->   TokenEpsilon : lexer rest
            ("Phi",rest)     ->   TokenFail : lexer rest
            ("Sigma",rest)   ->   TokenSucceed : lexer rest
            ("Mu",rest)      ->   TokenMu : lexer rest
            (taskName,rest)  ->   TokenTaskName taskName : lexer
rest

-- the name of a simple task begins with a lowercase
lexTaskAt cs =
      case span isAlphaNum cs of
            ("let",rest)     ->   TokenLet : lexer rest
            (simple,rest)    ->   TokenSimple simple : lexer
rest

-- general trace function
tr :: String -> SetOfTraces
tr [] = empty
tr s = trace (taskAlgebraParser (lexer s)) []

}
-- end of happy code :(
```

```
-- task semantics version 1: Traces module
module Traces where

import Data.Set as Set

-------------------------------------------------------------

-- Activity
data Activity
      = Epsilon
      | Fail
      | Succeed
      | Task Task
      | Sequence Activity Activity
      | Selection Activity Activity
      | Parallel Activity Activity
      | UntilLoop Activity Task Task
      | WhileLoop Activity Task Task
      | CompoundTask String Activity
      | Model Activity Activity
   deriving (Eq, Ord)

instance Show Activity where
      show Epsilon          = show (trace Epsilon [])
      show Fail             = show (trace Fail [])
      show Succeed          = show (trace Succeed [])
      show (Task t)         = show t
      show (Sequence a1 a2) = show (trace (Sequence a1 a2) [])
      show (Selection a1 a2)= show (trace (Selection a1 a2) [])
      show (Parallel a1 a2) = show (trace (Parallel a1 a2) [])
      show (UntilLoop act at1 at2)
                        = show (trace (UntilLoop act at1 at2) [])
      show (WhileLoop act at1 at2)
                        = show (trace (WhileLoop act at1 at2) [])
      show (CompoundTask s act)
                  = show (addToDictio (CompoundTask s act) [])
      show (Model ct act)   = show (trace (Model ct act) [])

-- Task
data Task
      = Simple String
      | Brackets Activity
      | Encapsulation Activity
      | Compound String
   deriving (Eq, Ord)

instance Show Task where
      show (Simple s)  = show (trace (Task (Simple s)) [])
      show (Brackets a)= show (trace (Task (Brackets a)) [])
      show (Encapsulation a)
                  = show (trace (Task (Encapsulation a)) [])
      show (Compound a)= show (trace (Task (Compound a)) [])

-- trace functions
```

```
trace :: Activity -> DataDictionary -> SetOfTraces
-- Simple Traces
trace Epsilon _                    = singleton epsilon
trace Fail _                       = singleton [Phi]
trace Succeed _                    = singleton [Sigma]
trace (Task (Simple s)) _   = singleton [Ident s]

-- Sequence composition
trace (Sequence a b) dict   = trace a dict #* trace b dict

-- Selection
trace (Selection a b) dict = if (trace a dict)== (trace b
dict) then trace a dict
                                                 else singleton
[Commit] #* union (trace a dict) (trace b dict)

-- Parallel composition
trace (Parallel a b) dict   = trace a dict // trace b dict

-- Brackets (parenthesis)
trace (Task (Brackets a)) dict         = trace a dict

-- Encapsulation
trace (Task (Encapsulation a)) dict    = unpack (trace a dict)

-- Until-loop repetition
trace (UntilLoop act atom1 atom2) dict= if atom1==atom2
                                                                {-
generating just the first two traces of the until-loop we can
derive

      Mu.x(act;E+x) as  act;(E+(act;E+E)) => act;E+act
      -}
                                                                then
trace (finiteU (UntilLoop act atom1 atom2) 2) dict
                                                                else
error ("Until-loop structure error. "++

      show (findMin (trace (Task atom1) dict)) ++" and "

      ++show (findMin (trace (Task atom2) dict))++" have to use
an unique name.")

-- While-loop repetition
trace (WhileLoop act atom1 atom2) dict= if atom1==atom2
                                                                {-
generating just the first two traces of the while-loop we can
derive

      Mu.x(E+act;x) as E+(act;(E+(act;E))) => E+(act;E+act)
           -}
                                                                then
trace (finiteW (WhileLoop act atom1 atom2) 2) dict
                                                                else
error ("While-loop structure error. "++
```

```
      show (findMin (trace (Task atom1) dict)) ++" and "

      ++show (findMin (trace (Task atom2) dict))++" have to use
an unique name.")


-- traceModel myTaskDict encap = trace encap
trace (Model ct act) dict    =  trace act (addToDictio ct dict)

-- When found a compound task, it have to trace its relationed
"code"
trace (Task (Compound a)) dict = trace (findCTask a dict) dict


-------------------------------------------------------------
-- trace low level definitions

-- elements of the traces, as subclasses of Eq
data Event = Ident String | Phi | Sigma | Commit
  deriving (Eq, Ord)

-- defining specific instances of Show
instance Show Event where
  show (Ident c) = c
  show Phi = "Phi"
  show Sigma = "Sigma"
  show Commit = "!"

-- a trace is a list of events
type Trace            = [Event]
type SetOfTraces= Set Trace

epsilon :: Trace
epsilon = []

-- Trace Concatenation
(#) :: Trace -> Trace -> Trace
[Sigma] # (item:rest)            = [Sigma] # rest
[Phi] # (item:rest)              = [Phi] # rest
[Commit] # trace@(item:rest)
      | item == Commit           = trace
      | otherwise                = Commit : trace
(item:rest) # trace              = item : (rest # trace)
epsilon#trace                    = trace

-- concatenated product
(#*) :: SetOfTraces -> SetOfTraces -> SetOfTraces
setA #* setB
      | setA == empty  = empty
      | setB == empty = empty
      | otherwise           = union (insert (findMin setA #
findMin setB)
            (singleton (findMin setA) #* (difference setB
(singleton (findMin setB)))))
            ((difference setA (singleton (findMin setA))) #*
setB )
```

```
-- Trace interleaving
(~~) :: Trace -> Trace -> SetOfTraces
[] ~~ trace                         = singleton trace
trace ~~ []                         = singleton trace
traceA@(a:as) ~~ traceB@(b:bs)
      | traceA == [Sigma]   = singleton [Sigma]
      | traceB == [Sigma]   = singleton [Sigma]
      | traceA == [Phi]     = singleton [Phi]
      | traceB == [Phi]     = singleton [Phi]
      | a   == Commit       = (singleton [Commit]) #* (as ~~
traceB)
      | b == Commit         = (singleton [Commit]) #* (bs ~~
traceA)
      | otherwise                 = union (singleton [a] #* (as
~~ traceB))
           (singleton [b] #* (bs ~~ (traceA)))

-- distributed union
(//) :: SetOfTraces -> SetOfTraces -> SetOfTraces
setA // setB
      | setA == empty = empty
      | setB == empty = empty
      | otherwise           = union (union (findMin setA ~~
findMin setB) (singleton (findMin setA) // (difference setB
(singleton (findMin setB)))))
           ((difference setA (singleton (findMin setA))) //
setB )

-- unpacking
unpack :: SetOfTraces -> SetOfTraces
unpack set
      | set == empty   = empty
      | otherwise           = union (singleton (lift (findMin
set))) (unpack (difference set (singleton (findMin set))))

-- lift
lift :: Trace -> Trace
lift [] = []
lift [Sigma]= []
lift (a:as) = a: (lift as)


-- Additional functions

-- dealing compound Tasks and the data dictionary
type DataDictionary   = [(String, Activity)]

-- return the Activity of a particular compound task
findCTask :: String -> DataDictionary -> Activity
findCTask [] _ = Epsilon
findCTask _ [] = Epsilon
findCTask a (ele:rest)
      | a == fst ele   = snd ele
```

```
      | a < fst ele    = Epsilon
      | otherwise           = findCTask a rest

-- add a compound task to the data dictionary
addToDictio :: Activity -> DataDictionary -> DataDictionary
addToDictio (CompoundTask s act) [] = [(s,act)]
addToDictio (CompoundTask s act) dict@(ele:rest)
                                  = if s<fst ele then
(s,act):dict
                                        else  ele: addToDictio
(CompoundTask s act) rest

-- return a finite expression for a Until-loop (till n).
finiteU :: Activity -> Int -> Activity
finiteU (UntilLoop act atom1 atom2) 0 = Epsilon
finiteU (UntilLoop act atom1 atom2) n = Sequence act
(Selection Epsilon (finiteU (UntilLoop act atom1 atom2) (n-
1)))
finiteU act n = act

-- return a finite expression for a While-loop (till n).
-- E+(act;E+act)
finiteW :: Activity -> Int -> Activity
finiteW (WhileLoop act atom1 atom2) 0 = Epsilon
finiteW (WhileLoop act atom1 atom2) n = Selection Epsilon
(Sequence act (finiteW (WhileLoop act atom1 atom2) (n-1)))
finiteW act n = act
```

C.2 LTL

```
-- Linear temporal logic functions
module Main where
import System (getArgs)

import MainTaskAlgebra  -- task algebra parser
import Data.Set as Set
import Traces

-- LTL syntax
data Phi
    = Bool Bool
    | Pr String
    | Not Phi
    | And Phi Phi
    | Or Phi Phi
    | Impl Phi Phi
    | X Phi                -- Next phi
    | G Phi                -- All future states (Globally)
    | F Phi                -- Eventually (some Future state)
    | U Phi Phi            -- Until (U p q  -- p holds until
q, (when q holds p doesn't hold anymore)
    | W Phi Phi            -- Weak-until
    | R Phi Phi            -- Release
```

```
      deriving (Eq, Ord, Show)


-- returns true if the atomic proposition is true for the
first state
-- of the trace
-- expr - expression defining the traces (or set of paths)
check :: String -> Phi -> (Bool, Trace)
check expr phi  = evalAllTraces (toList (tr expr)) phi

-- evaluates every trace
evalAllTraces :: [Trace] -> Phi  -> (Bool, Trace)
evalAllTraces []                 _          = (True, [])
evalAllTraces traces@(t:ts) phi  = if eval t phi
           then evalAllTraces ts phi -- (i+1)
           else (False, t)

-- returning boolean result
eval :: Trace -> Phi -> Bool
eval trace (Bool b)         = b
eval trace (Pr p)      = pr trace p
eval trace (Not p)     = not (eval trace p)
eval trace (And p q)   = if eval trace p then
                            if eval trace q then True
                                       else False
                              else False

eval trace (Or p q)    = if eval trace p then True
                            else if eval trace q then True
                                          else False
eval trace (Impl p q)  =
           if eval trace p && not (eval trace q) then False
                                       else True
eval trace (X p)       = x trace p
eval trace (G p)       = g trace p
eval trace (F p)       = f trace p
eval trace (U p q)     = u trace p q
eval trace (W p q)     = w trace p q
eval trace (R p q)     = r trace p q


-- resolves p using the set of traces
pr :: Trace -> String  -> Bool
pr [] _           = False          -- error "invalid state"
pr (t:ts) atm
                | t == Commit         = pr ts atm
                | Ident atm == t = True
                | otherwise                = False

-- X phi - neXt phi states that the formula phi should hold
for the rest
-- of the execution without the first state
x :: Trace -> Phi -> Bool
x [] _     = False
x trace phi      = eval (dropHead trace) phi
```

```
-- G (Globally) phi holds for all the future states
g :: Trace -> Phi -> Bool
g [] _           = True
g trace phi = if eval trace phi then g (dropHead trace) phi

                            else False

-- F Phi -- Eventually (some Future state)
-- evaluate the tression for every atom in each trace if there
is no null trace
f :: Trace -> Phi -> Bool
f [] _           = False
f trace phi = if eval trace phi then True
                     else f (dropHead trace) phi


-- U p q -- p until q (for p != q)
-- p has to be true at least the first state and after q has
to be true
-- but p is not required to hold
-- u function verifies if p is true and if it is then pass the
control to u2
u :: Trace -> Phi -> Phi -> Bool
u []   _ _ = False
u trace p q
                | eval trace q   = True
                | eval trace p   = u2 (dropHead trace) p q
                | otherwise           = False

-- u2 verifies if p is true and when it's false verifies if q
is true
u2 :: Trace -> Phi -> Phi -> Bool
u2 trace p q
                | trace == []    = False
                | eval trace q   = True          -- if q then
try the next trace
                | eval trace p   = u2 (dropHead trace) p q
                | otherwise = False

-- W p q -- p until q but q is not required to be satisfied
w :: Trace -> Phi -> Phi -> Bool
w []   _ _ = False
w trace p q
                | eval trace q   = True  -- if q then try the
next trace
                | eval trace p   = w2 (dropHead trace) p q
                | otherwise           = False

-- w2 verifies if p still holds or if q holds
w2 :: Trace -> Phi -> Phi -> Bool
w2 trace p q
                | trace == []    = True -- try next trace
                | eval trace q   = True -- if q then try the
next trace
                | eval trace p   = w2 (dropHead trace) p q
```

```
                | otherwise      = True -- try next trace

-- p R q. p releases q if q is true until the first position
in which p is true
-- (or forever if such a position does not exist).
-- p R q = Not (Not p U Not q)
r :: Trace -> Phi -> Phi -> Bool
r trace p q = eval trace (Not (U (Not p) (Not q) ))

-- drops the first element of the list
dropHead :: Trace -> Trace
dropHead []      = []
dropHead (t:ts)
     -- drops the commit element before droping the next
relevant element
                      | t == Commit    = dropHead ts
                      | otherwise      = ts
```

C.3 CTL

```
-- Computation Tree Logic (CTL)  functions
module Main where

import Traces
import MainTaskAlgebra
import Data.List
import qualified Data.Set as Set

-- CTL syntax
data Phi    -- Path and State Operators
     -- operands and logical operators
     = Pr String
     | Bool Bool
     | Not Phi
     | And Phi Phi
     | Or Phi Phi
     | Impl Phi Phi
-- A ? - All: ? has to hold on all paths starting from the
current state.
     | AX Phi                    -- Next phi
     | AG Phi                    -- All future states
(Globally)
     | AF Phi                    -- Eventually (some Future
state)
     | AU Phi Phi                -- Until (U p q  -- p holds
until q, (when q holds p doesn't hold anymore)
-- E ? - Exists: there exists at least one path starting from
the current state where ? holds.
     | EX Phi                    -- Next phi
     | EG Phi                    -- All future states
(Globally)
     | EF Phi                    -- Eventually (some Future
```

```
state)
      | EU Phi Phi                -- Until (U p q  -- p holds
until q, (when q holds p doesn't hold anymore)
   deriving (Eq, Ord, Show)


-- create tree from list of traces
-- Trace == [Event] (defined in Traces.hs)
type ListOfTraces     = [Trace]

-- Returns true if the atomic proposition is true for the
first state
-- of the trace
-- expr - expression defining the traces (or set of paths)
-- phi  - CTL expression
check :: String -> Phi ->  ([ [Integer] ], Node)
check expr phi = (sort (sat (tree (Set.toList(tr expr))) phi),
tree (Set.toList(tr expr)) )


-- SAT function. It takes a CTL formula s input and returns
the set of states
-- satisfying the formula.
-- It calls the functions satEx, SatEu and SatAf,
respectively, if EX, EU, or AF
sat :: Node -> Phi -> [ [Integer] ]
sat tr (Bool True)                = s tr
sat _ (Bool False)                =     []    -- phi is False
sat tr (Pr str)                         -- phi is atomic
      | tr == Empty               =     []
      | atomic tr str == [] =     []
      | otherwise                 = atomic tr str
sat tr (Not p)                    = (getAllStateNumbers
(getAllStates [tr])) \\ (sat tr p)
sat tr (And p q)            = (sat tr p) `intersect` (sat tr q)
sat tr (Or p q)                   = (sat tr p) `union` (sat tr
q)
sat tr (Impl p q)                 = sat tr (Not p `Or` q)
sat tr (AX p)                     = satAx    tr p
sat tr (EX p)                     = satEx tr p
sat tr (AU p q)                   = if snd (satAu tr p q) ==
True then fst (satAu tr p q)
           else  []
sat tr (EU p q)                   = satEu tr p q
sat tr (EF p)                     = sat tr (EU (Bool True) (p))
sat tr (EG p)                     = if snd (satEg tr p) == True
then fst (satEg tr p)
           else  []
sat tr (AF p)                     = if snd (satAf tr p) == True
then fst (satAf tr p)
           else  []

sat tr (AG p)                     = if snd (satAg tr p) == True
then fst (satAg tr p)
           else  []
```

```
-- S  here represents the set of states that each element of
the diagram can have
s :: Node -> [ [Integer] ]
s tr@( Node (nodeNumber, evt) (subnodes) )
      | tr == Empty                =     []
      | otherwise                  =  [nodeNumber]


satAx :: Node -> Phi -> [[Integer]]
satAx Empty _    = []
satAx tr p = if length (sat tr (Not (EX (Not p)))) < length
(getSubNodes tr)
                       then [] else  [getNodeNumber tr] `union`
sat tr (Not (EX (Not p)))

-- satEx finds the states satisfying EX phi, looking FORWARD
-- along the subnodes
satEx :: Node -> Phi -> [[Integer]]
satEx Empty _ = []
satEx (Node (nodeNumber, evt) (sn:snds) ) p
      | sn == Empty    = []
      | otherwise      = if ((sat sn p)  `union`  (if snds/=[]
then (satEx (Node (nodeNumber, evt) (snds) ) p) else [])) /=
[]
                                    then [nodeNumber] `union`
((sat sn p)  `union`  (if snds/=[] then (satEx (Node
(nodeNumber, evt) (snds) ) p) else []))
                                    else []

-- satAg tr p
-- It determines the set of states satisfying AG p
satAg:: Node -> Phi -> ([[Integer]], Bool)
satAg (Node (_, _) [] ) _   = ([], True)
satAg Empty       _                    = ([], True)
satAg nd@(Node (nodeNumber, evt) (sn:snds) ) p
      -- if it is found, add to the list and continue with the
next branch
      | sat nd p /= []      =  ( [nodeNumber] `union`((sat nd
p) ++ (concat  [fst (satAg x p) | x<-(sn:snds)] )  ),
                                    (and  [snd (satAg x p) | x<-
(sn:snds)] )   )
      | otherwise                    = ([], False)



-- satEg tr p
-- It determines the set of states satisfying EG p
satEg:: Node -> Phi -> ([[Integer]], Bool)
satEg (Node (_, _) [] ) _   = ([], True)
satEg Empty       _                    = ([], True)
satEg nd@(Node (nodeNumber, evt) (sn:snds) ) p
      -- if it is found, add to the list and continue with the
next branch
```

```
      | sat nd p /= []      =  ( [nodeNumber] `union`((sat nd
p) ++ (concat  [fst (satEg x p) | x<-(sn:snds)] )  ),
                                  (or  [snd (satEg x p) | x<-
(sn:snds)] )   )
      | otherwise                       = ([], False)


-- satAf tr p
-- It determines the set of states satisfying AF p
satAf:: Node -> Phi -> ([[Integer]], Bool)
satAf (Node (_, _) [] ) _   = ([], True)
satAf (Node (nodeNumber, evt) (sn:snds) ) p
      -- if it is found it, add to the list and continue with
the next branch
      | sat sn p /= []      =  ( [nodeNumber] `union`((sat sn
p) ++ fst ( satAf (Node (nodeNumber, evt) (snds) ) p) ),
                                                 {-True &&-} snd (
satAf (Node (nodeNumber, evt) (snds) ) p) )
      -- otherwise if there are not subnodes then false
      | getSubNodes sn == [Empty]= ([], False)
      -- otherwise try the branch till you find it or till the
end
      | fst (satAf (Node (nodeNumber, evt) (getSubNodes sn) )
p) /= []
                                      = ( [nodeNumber] `union`
( ((getNodeNumber sn):fst (satAf (Node (nodeNumber, evt)
(getSubNodes sn) ) p)  )
                                                ++ fst ( satAf
(Node (nodeNumber, evt) (snds) ) p) ),
                                                snd (satAf (Node
(nodeNumber, evt) (getSubNodes sn) ) p) &&
                                                snd ( satAf (Node
(nodeNumber, evt) (snds) ) p) )
      | otherwise                       = ([], False)


-- satAu determines the set of states satisfying A [ p U q]
-- It computes the states satisfying p by calling sat. Then,
it accumulates states
-- satisfying A [ p U q] in the manner described in the
labelling algorithm
satAu:: Node -> Phi -> Phi -> ([[Integer]], Bool)
satAu Empty _ _  = ([], True)
satAu tr@(Node (nodeNumber, evt) snds ) p q
      | sat tr q /= []      = (sat tr q, True)
      | sat tr p /= [] = exploreAu tr p q
      | otherwise                 = ([], False)

-- explore to see if p es true and then q
exploreAu :: Node -> Phi -> Phi -> ([[Integer]], Bool)
exploreAu tr@(Node (nodeNumber, evt) snds ) p q
      | snds == [Empty]                                       =
([], False)
      | snd (applyToSnAu snds p q) == True  = ((sat tr p)
`union` (fst (applyToSnAu snds p q)), True)
      | otherwise
```

```
      = ([], False)

-- look into subnodes for p and q
applyToSnAu:: SubTree -> Phi -> Phi -> ([[Integer]], Bool)
applyToSnAu []           _ _          = ([], True)
applyToSnAu  (sn:snds) p  q
      | sn == Empty    = ([], False)
      | sat sn q /= []      = ((sat sn q) `union` (fst
(applyToSnAu snds p q)), snd (applyToSnAu snds p q)) -- True)
      | sat sn p /= [] = ( concatMap fst ( (exploreAu sn p
q):[applyToSnAu snds p q] ) , (and (map snd ( (exploreAu sn p
q):[applyToSnAu snds p q] ) )) )
      | otherwise           = ([], False)


-- satEu determines the set of states satisfying E [ p U q]
-- It computes the states satisfying p by calling sat. Then,
it accumulates states
-- satisfying E [ p U q] in the manner described in the
labelling algorithm
satEu:: Node -> Phi -> Phi -> [[Integer]]
satEu Empty _ _  = []
satEu tr@(Node (nodeNumber, evt) snds ) p q
      | sat tr q /= []      = sat tr q
      | sat tr p /= [] = explore tr p q
      | otherwise                = []

-- explore to see if p es true and then q
explore :: Node -> Phi -> Phi -> [[Integer]]
explore tr@(Node (nodeNumber, evt) snds ) p q
      | snds == [Empty]                       = []
      | (applyToSn snds p q) /= []     = (sat tr p) `union`
(applyToSn snds p q)
      | otherwise                                 = []

-- look into subnodes for p and q
applyToSn:: SubTree -> Phi -> Phi -> [[Integer]]
applyToSn []           _ _          = []
applyToSn  (sn:snds) p  q
      | sn == Empty    = []
      | sat sn q /= []      = (sat sn q) `union` (applyToSn
snds p q)
      | sat sn p /= [] = (explore sn p q) `union` (applyToSn
snds p q)
      | otherwise           = []


-- atomic function
--  Phi is atomic. Returns {s in S | phi in L(s) }
atomic :: Node -> String -> [[ Integer]]
atomic Empty _   = []
atomic (Node (nodeNumber, evt) (subnodes) ) str
      | evt == Ident str    = [nodeNumber]
      | otherwise                = []
```

```
-- getAllStateNumbers scans the tree and return a list with
the number of the states
getAllStateNumbers :: [Node] -> [ [Integer] ]
getAllStateNumbers [] = []
getAllStateNumbers (nd@(Node (n, evt) (_)):nds)
      | nd==Empty           = []
      | otherwise           = n:getAllStateNumbers nds


-- getAllStates return all the nodes as a single list
getAllStates :: SubTree -> [Node]
getAllStates [] = []
getAllStates [Empty] = []
getAllStates (nd@(Node (n, evt) (subnodes)) : sbns)
      | nd == Empty    = []
      | otherwise           = ((Node (n, evt) []):getAllStates
(subnodes)) ++ getAllStates sbns



-----------------------------------------------------------------
---------
data Node =      Empty
      |  Node ([Integer], Event) (SubTree)
      deriving (Eq, Ord, Show)
type Empty =[]
type SubTree = [Node]


--getSubNodes gets the list of sub-nodes from a current node
getSubNodes :: Node -> SubTree
getSubNodes Empty           = [Empty]
getSubNodes (Node (s, evt) st) = st

-- return tree from traces
tree :: ListOfTraces  -> Node
tree []  = Empty
tree traces  =  Node ([0], Ident "null") (subNodes traces [0]
[])

-- creates the subnodes from the list of traces for a tree
{- Parameters:
                ListOfTraces : Traces to introduce to the
nodes
                Integer: node number
                SubTree: subnodes to be passed to the function
                SubTree: final result of the nodes  -}
subNodes :: ListOfTraces -> [Integer] -> SubTree-> SubTree
subNodes [] _ sbnds   = sbnds
subNodes (t:ts) n sbnds
           | t == []  = subNodes ts n sbnds
           | sbnds == []    = subNodes ts (n) ((setFirstTrace t
n):sbnds)
           | otherwise = subNodes ts (n) (setNTrace t sbnds n)
```

```
-- setFirstTrace creates a branch based  on the first trace
setFirstTrace :: Trace -> [Integer] -> Node
setFirstTrace [] _                = Empty
setFirstTrace (evt:tr) n
           | evt == Commit        = setFirstTrace tr n
           | otherwise = Node  ((n++[1]), evt) ([setFirstTrace
tr (n++[1])])

-- setNTrace adds a trace to the tree without repeating states
already created
-- Parameters:  Trace - Trace to add to the tree,
--                    SubTree - subnodes
--                    Integer - number of last node added
--                    SubTree - modified subnodes
setNTrace :: Trace -> SubTree -> [Integer] -> SubTree
setNTrace [] _ _             = [Empty]
setNTrace (evt:tr) subnodes n
           | evt == Commit        = setNTrace tr subnodes n
           -- the event is in a subnode. Go to the next
sublevel
           | (cmpEvtInSubNodes evt subnodes) /= Empty
                       =  addToSubNodes (cmpEvtInSubNodes evt
subnodes)
                                (setNTrace tr (getSubNodes
(cmpEvtInSubNodes evt subnodes)) ( getNodeNumber
(cmpEvtInSubNodes evt subnodes) ))
                                 subnodes
           -- the event is not in a subnode. Add the node with
the event
           | otherwise            = (Node (getNextNumber
subnodes n, evt) ([setFirstTrace tr (getNextNumber subnodes
n)])):subnodes        -- newNode evt n

-- It gets the number of a node
getNodeNumber :: Node -> [Integer]
-- getNodeNumber [] = Empty
getNodeNumber (Node (n, _) _) =  n

--It gets the next number on a list of nodes for a particula
level
getNextNumber :: SubTree -> [Integer] -> [Integer]
getNextNumber subnodes n = n ++ [(fromIntegral (length
subnodes))+1]

-- addToSubNodes adds the result of setNTrace to the subnodes
of the existent node
--  Parameters: Node - The existent node. To its subnodes the
result of setNTrace will be added
--                          SubTree - set of subnodes resulted
from setNTrace
--                          SubTree - original subnodes where
Node is part of.
--                          SubTree - subnodes at the level of
the node parameter
addToSubNodes :: Node -> SubTree -> SubTree -> SubTree
addToSubNodes Empty _ _     = []
```

```
addToSubNodes _ []     _      = []
addToSubNodes (Node (n, evt) subnodes)  moreSubnodes  osbnds=
replaceNode (Node (n, evt) (moreSubnodes)) osbnds


-- replaceNode  replaces Node in the list of nodes
-- Parameters:  Node - The node to replace in the list
--                    SubTree - The list of nodes where the
node is being replaced
--                    SubTree - new list with the node replaced
replaceNode :: Node -> SubTree -> SubTree
replaceNode Empty _ = []
replaceNode (Node (n, evt) sbnds) ( (Node (n2, evt2)
sbnds2):ms)
      | n == n2        = (Node (n, evt) sbnds) : ms
      | otherwise      = (Node (n2, evt2) sbnds2) : (replaceNode
(Node (n, evt) (sbnds)) ms)


-- cmpEvtInSubNodes returns the node if this has the same
event
-- to compare or empty in other case
--          Event - Event to compare with the nodes
--          SubTree  -  the node with the subnodes to compare
cmpEvtInSubNodes :: Event -> SubTree ->Node
cmpEvtInSubNodes _ [] = Empty
cmpEvtInSubNodes _ [Empty] = Empty
cmpEvtInSubNodes evt (Node (n, evn) stn:nds)
            | evt==evn = Node (n, evn) stn
            | otherwise = cmpEvtInSubNodes evt    nds

-------------------------------------------------------------------
---------
-- Displays a branch of the trace list
displayBranch :: ListOfTraces -> ListOfTraces
displayBranch [] = []
displayBranch (trc:restOfTraces)
      | restOfTraces == []  = [trc]
      | head trc == head (head restOfTraces) = trc:
displayBranch(restOfTraces)
      | otherwise      =  [trc]

-- find the next trace list
nextTraceList :: Trace -> ListOfTraces ->ListOfTraces
nextTraceList [] listTrc    = listTrc
nextTraceList trc []        = []
nextTraceList trc (t:ts)
      | head trc == head t  = nextTraceList trc ts
      |otherwise                        = t:ts

-- Displays initial branchs from a trace list
displayBranchs :: ListOfTraces -> [ListOfTraces]
displayBranchs [] = []
displayBranchs (trc:restOfTraces)
      | restOfTraces == []  = [trc]:[]
      | head trc == head (head restOfTraces)
```

```
           = (displayBranch (trc:restOfTraces)):(displayBranchs
(nextTraceList trc (tail restOfTraces)))
      | otherwise = [trc]:displayBranchs (restOfTraces)

-----------------------------------------------------------------
---------

-- main function
main = do
        args<- getArgs
           if length args ==2 then print ( check (head args)
( ctlCompiler (args!!1)) ) else print "Sintax:
ctlModelChecking <Task-Algebra-Expr> <CTL-Expr>"
```