THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (Toulouse INP)

Discipline ou spécialité :

Informatique et Télécommunication

Présentée et soutenue par :

M. MATHIEU MONTIN

le lundi 14 septembre 2020

Titre :

A formal framework for heterogeneous systems semantics

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (IRIT)

Directeur(s) de Thèse :

M. YAMINE AIT AMEUR
M. MARC PANTEL

Rapporteurs :

M. FRÉDÉRIC MALLET, INRIA SOPHIA ANTIPOLIS
Mme CATHERINE DUBOIS, ENSIIE

Membre(s) du jury :

M. JEAN-PAUL BODEVEIX, , Président
M. BENOIT COMBEMALE, UNIVERSITE RENNES 1, Membre
M. FREDERIC BOULANGER, SUPELEC, Membre
M. MARC PANTEL, TOULOUSE INP, Membre

Contents

Preamble **ix**
P.1 Acknowledgements ix
P.2 Abstract xi
P.2.1 English abstract xi
P.2.2 French abstract xii
P.3 Content of the manuscript xiii
P.3.1 Detailed outline of the chapters xiii
P.3.2 Dependencies between chapters xv
P.4 Lecture grid xvi
P.4.1 On the layout of this document xvi
P.4.2 On the AGDA code in this document xvi

1 Introduction **1**
1.1 Modelling heterogeneous systems: a challenge 1
1.1.1 The need of verifying critical software 1
1.1.2 Separations of concerns 2
1.1.3 Horizontal separation: components everywhere 4
1.1.4 Transversal separation: aspect oriented design 5
1.1.5 Vertical separation: refinement on the rescue 6
1.1.6 Challenges in heterogeneous modelling 7
1.2 Our contributions 8
1.2.1 Our goal: mixing separations over behavioural properties . 8
1.2.2 A formal execution of event-based systems 8
1.2.3 A formal depiction of trace refinement 9
1.2.4 A denotational semantics for CCSL 10
1.2.5 An investigation on the impact of refinement over CCSL . 11
1.2.6 Subsidiary contribution: AGDA methodology 11

2 From formal methods to AGDA **13**
2.1 Formal methods: the banner of computer scientists 14
2.1.1 Model checking: exploring state spaces 14
2.1.2 Static analysis and abstract interpretation 15
2.1.3 Correct-by-construction approaches: the use of refinement 16

2.1.4 Automated theorem provers 17
2.1.5 Proof assistants 18
2.1.6 Combining verification methods together 20
2.2 Dependent types: types or properties ? 20
2.2.1 Logics and classical logics 21
2.2.2 Intuitionistic logic 22
2.2.3 Curry-Howard isomorphism 22
2.2.4 Dependent types 23
2.3 AGDA: a programming language and much more 24
2.3.1 Presentation of AGDA 24
2.3.2 The specificities of the AGDA language 25
2.3.3 The related tools around AGDA 29
2.3.4 AGDA: a short tutorial 35
2.3.5 AGDA vs COQ: a matter of taste 43

3 Globally unique lists: an AGDA library 53
3.1 Methodology 54
3.2 Unique membership in a list 57
3.2.1 The Any Unique predicate 57
3.2.2 Properties of Any Unique 60
3.2.3 Decidability of Any Unique 62
3.2.4 Parametrized membership 63
3.3 Global unicity 66
3.3.1 Definition 66
3.3.2 Globally unique lists on different kind of relations 71
3.4 Commands and requests over globally unique lists 73
3.4.1 Commands 73
3.4.2 Queries 79
3.4.3 Examples 81
3.4.4 Comparison 83
3.5 Two relevant instances of globally unique lists 83

4 Toward the mechanization of event-based systems in AGDA 85
4.1 Presentation of the approach 86
4.1.1 Objective of the approach 86
4.1.2 A structural representation of the states 86
4.1.3 A relational representation of the transitions 87
4.1.4 A constrained evolution of the system 87
4.2 Application to PETRI NET 88
4.2.1 Presentation of the PETRI NET language 88
4.2.2 Our approach applied to PETRI NET 92
4.3 Application to SIMPLEPDL 100
4.3.1 Presentation of the SIMPLEPDL language 100
4.3.2 Our approach applied to SIMPLEPDL 102

5 Refining instants in asynchronous systems execution 111
5.1 Handling of time in asynchronous systems 112
5.1.1 Introduction to traced-based semantics 112
5.1.2 Instants 113
5.1.3 Strict partial orders 115
5.2 A formal definition of instant refinement 118
5.2.1 On refining instants 118
5.2.2 Our proposal: relating strict partial orders 119
5.3 Mechanization of the refinement relation 121
5.3.1 Transformation between levels of abstraction 121
5.3.2 Proof of partial ordering 122
5.4 An example of instant refinement 124
5.4.1 Presentation of the example 124
5.4.2 Verification of the example 128

6 A mechanized denotational semantics of CCSL 129
6.1 CCSL: A language to abstract event occurrences 130
6.1.1 Domain Specific Modelling Languages 130
6.1.2 Presentation of CCSL 135
6.1.3 TIMESQUARE: an operational semantics to CCSL 138
6.1.4 A paper version of the denotational semantics of CCSL . . 140
6.2 A mechanized semantics of CCSL 140
6.2.1 Time-related notions 141
6.2.2 Clocks 145
6.2.3 Relations 150
6.2.4 Expressions 168

7 An extension of CCSL with refinement 175
7.1 Introduction 176
7.1.1 Stakes of the approach 176
7.1.2 Formal context 177
7.1.3 Useful naming convention and operators 177
7.2 1-N clock refinement 178
7.2.1 Definition of 1-N refinement 178
7.2.2 1-N refinement and coincidence 179
7.2.3 1-N refinement and precedence 181
7.3 1-1 clock refinement 183
7.3.1 Motivation 183
7.3.2 Definition of 1-1 refinement 184
7.3.3 Consequences of the definition 185
7.3.4 1-1 refinement and coincidence 186
7.3.5 1-1 refinement and precedence 186

8 Conclusion **189**
8.1 Assessments 189
8.2 Limitations and Perspectives 192
8.3 Metrics 196
8.4 Publications and Seminaries 198

Appendices **215**
A.1 On conformity of lists 215
A.1.1 Decidability of the membership relation 215
A.1.2 Decidability of the none or one relation 215
A.1.3 Lemmas about membership of characters 216
A.1.4 Lemmas about global unicity and equivalence 216
A.1.5 Lemmas about assignment and membership 217
A.1.6 Comparison between globally unique lists 218
A.1.7 Trimming a non-empty globally unique list 222
A.2 On Petri nets and SimplePDL models 223
A.2.1 Lemmas to add an arc to a net 223
A.2.2 Exporting nets to TINA 224
A.2.3 Decidability from a list of candidates 225
A.2.4 A detailed building of the firing predicates 226
A.2.5 A detailed building of the decidability predicates 226
A.2.6 Decidability of the equality between worksequences 227
A.2.7 An alternate definition of SimplePDL 227
A.2.8 Adding a WorkSequence to a process model 228
A.2.9 Decidability of the predicate of compliance 229
A.3 On instant refinement 231
A.3.1 Definition of instants 231
A.3.2 Equivalence between pairs of relations 234
A.3.3 Partial ordering between pairs of relations 235
A.3.4 Verification of the example 235
A.4 On CCSL 240
A.4.1 Image of a function in a setoid 240
A.4.2 The binding function of the precedence is bijective 241
A.4.3 Compliance between precedence and equality 241
A.4.4 A clock can strictly precede itself on integers 242
A.4.5 Two clocks can precede each other and not be equal 244
A.4.6 Antisymmetry of the strict precedence towards the equality 246
A.4.7 Non empty decidable clocks on natural numbers 247
A.4.8 Lattice from union and intersection 249
A.5 On refinement and CCSL 250
A.5.1 A binding operator to ease refinement proofs 250
A.5.2 From unique existence to existence 250
A.5.3 Embodiment of strict precedence 251
A.5.4 Abstraction of precedence 252

List of Figures

1 Dependencies between chapters xv
2.1 Examples of inference rules 21
2.2 The type/formula correspondence 23
2.3 Example of unicode characters 25
2.4 Features around operators 26
2.5 A mixfix operator with three operands 27
2.6 A convenient way for defining lists 28
2.7 Currying a mixfix operator 28
2.8 Identifiers with relevant names 29
2.9 The AGDA EMACS mode 30
2.10 Limitations of AGSY 34
2.11 Inference rules for the Even predicate 38
2.12 deduction tree proving that 6 is even 39
2.13 Inference rules for the x ≡_ predicate 41
2.14 Comparison of addition commutativity proofs 45
2.15 A lambda term displayed by COQ 45
2.16 Comparison of addition commutativity proofs with automation . . 46
2.17 AGDA's automation 46
2.18 Comparison of equality proofs 48
2.19 An example of computationally irrelevant argument 50
2.20 AGDA does not evaluate irrelevant elements 51
2.21 AGDA evaluates all relevant elements 51
3.1 The Any data type 58
3.2 The Any! data type 58
3.3 Satisfaction of Any! with the lower than 3 predicate 59
3.4 Satisfaction of Any! with a given size of strings 60
4.1 PETRI NET meta-model 89
4.2 The seasons PETRI NET 90
4.3 The deadlock PETRI NET 91
4.4 A snapshot of TINA's usage 91
4.5 The generated nets 96
4.6 SPEM simplified meta-model 101
4.7 The development process 102
5.1 Both possible behaviours 115

5.2 The underlying partial order 116
5.3 The strict partial order dependent record 116
5.4 The five relations binding instants together 117
5.5 A trivial example of property 118
5.6 A more complex example of property 118
5.7 A simple system 124
5.8 A trace on a single timeline 124
5.9 One timeline per event 124
5.10 The system pilots a light 125
5.11 The trace of the system with the addition of the variable x 125
5.12 The refined system 126
5.13 Both levels of observation 126
5.14 Both annotated levels of observation 127
6.1 The V cycle in a system development 131
6.2 A simplified V cycle with the use of models 132
6.3 The V cycles in a product development 133
6.4 The place of CCSL inside the UML world 135
6.5 UML as an instance of MOF 136
6.6 A simplified CCSL meta-model 137
6.7 A simple CCSL specification in TIMESQUARE 139
6.8 A possible trace associated to the specification 139
6.9 An example of a clock c 145
6.10 Some instants are constrained 150
6.11 c_1 is a subclock of c_2 151
6.12 c_1 is equal to c_2 153
6.13 c_1 is in exclusion with c_2 155
6.14 An incorrect strict precedence example 156
6.15 A standard strict precedence example 157
6.16 A specific strict precedence example 157
6.17 An example of non-strict precedence 157
6.18 Transitivity of `dense` 160
6.19 Proof that the equivalence classes are respected 160
6.20 This clock strictly precedes itself 162
6.21 Two non-equal clocks with both precedences 164
6.22 A case of precedence without alternation 167
6.23 c_1 alternates with c_2 167
6.24 An example of intersection 168
6.25 An example of union 171
7.1 The binding operator 178
7.2 Clock precedence and abstraction 181
7.3 Clock precedence and embodiment 182
7.4 Clock precedence is not always preserved 183
7.5 An example of non-preservation of alternation 187

Preamble

P.1 Acknowledgements

This PhD has been supervised by associate professor Marc Pantel from IRIT, who I thank for his advice and suggestions regarding this work. I am very grateful to Pr. Pantel for providing me with a deep and interesting subject while believing in me for the entirety of this PhD, despite the doubts and difficulties that have arisen. Pr. Pantel has been very patient and very positive towards me which has been of great use for the success of this PhD. In addition, Pr. Pantel had done a very thorough and complete proof-reading of this document which helped to substantially improve the quality of both the work it depicts and the way it depicts it.

This work has been conducted in the ACADIE team at IRIT, whose researchers and PhD students have all been both helpful and supportive during my stay. It was a pleasure to work with such a pleasant team.

Among these researchers and PhD students, there were some with whom I created a special bound, for they shared my office for several months or even years. These people, Kahina, Guillaume and Florent made my stay at IRIT especially enjoyable and have been both of great help and comfort when it was needed. The time we shared together shall not be forgotten, and I wish them all the best for their future careers and personal lives.

This work could not have been successfully conducted without the constant help from the laboratory secretaries. They handled all the required procedures with celerity and devotion. These ladies, and especially Miss Sansus, have been of great support in conducting this work towards its completion and it has been a pleasure to meet and share time with them.

During this work I have had the opportunity to supervise the internship of Marie Bouette, who actively participated in this work. During two months, she has done a substantial work on exploring a branch which I would not have had time to investigate without her. I am grateful for her patience and her dedication on understanding as quickly as possible the work she had to participate in. Her work has produced fruitful blossoms, the fruits of which are depicted in Section 4.3.2.

I would like to specifically thank Pr. Catherine Dubois from *ENSIIE* for believing in my abilities to conduct this work and for providing me with very fruitful opportunities to share my work with fellow researchers and teams. She has been both very supportive in that regard as well as honest when part of my work in which we both believed happened not to bear the fruits we expected. I thank her deeply for that, it helped me realize once more that we learn even more from failures than we do from successes.

This document has received a thorough and conscientious proofreading from an old friend of mine, Vinay Damani, which saved me a lot of precious time while also improving the overall quality of my english, which is not my native language. I thank him deeply for that, and especially for agreeing to do so in such a short notice.

I am very grateful to both Pr. Catherine Dubois and Pr. Frédéric Mallet for agreeing to be the reviewers of this document. Their reports have been of great interest and provided me with much needed outside perspectives on my work.

I thank Pr. Marc Pantel, Pr. Benoît Combemale, Pr. Jean-Paul Bodeveix, Pr. Frédéric Boulanger, Pr. Catherine Dubois and Pr. Frédéric Mallet for agreeing to be part of the jury for the defense of this PhD, especially considering the sanitary conditions which surrounded the event. Their presence was very much appreciated, as well as their questions, remarks and interesting feedbacks regarding my work.

On a more personal note, I would like to thank my close relatives, my mother, father and sister, for their constant support during these years. Even though they do not have any academic background, they have always believed in me and they have allowed me to follow my path all the way to this PhD, often putting my interest and well-being before theirs. I am happy and thankful to have such a strong and loving family.

During this PhD, I have also received a very strong support from my family in law. They have constantly believed in my abilities to conduct this work, and meeting these expectations has been a healthy and powerful motivation tool. I also thank them deeply for they occasional concrete help, and even more so for their constant support and benevolence towards me.

Finally, I would like to thank Jessica Hornik, who has been my companion for most of this PhD. Against all odds she has stood next to me, and her help has been nothing but priceless. Several times I lost my way, and she led me back on track that many times. This work would simply never have been completed if it wasn't for her. Thank you Jessica, with all my heart.

P.2 Abstract

P.2.1 English abstract

Cyber physical systems (CPS) are usually complex systems which are often critical, meaning their failure can have significant negative impacts on human lives. A key point in their development is the verification and validation (V & V) activities which are used to assess their correctness towards user requirements and the associated specifications. This process aims at avoiding failure cases, thus preventing any incident or accident. In order to conduct these V & V steps on such complex systems, separations of concerns of various nature are used. In that purpose, the system is modelled using heterogeneous models that have to be combined together. The nature of these separations of concerns can be as follows: horizontal, which corresponds to a structural decomposition of the system; vertical, which corresponds to the different steps leading from the abstract specification to the concrete implementation; and transversal, which consists in gathering together the parts that are thematically identical (function, performance, security, safety, ...). These parts are usually expressed using domain specific modelling languages (DSML), while the V & V activities are historically conducted using testing and proofreading, and more and more often, using formal methods, which is advocated in our approach.

In all these cases, the V & V activities must take into account these separations in order to provide confidence in the global system from the confidence of its sub-parts bound to the separation in question. In other words, to ensure the correctness of the system, a behavioural semantics is needed which has to rely on the ad-hoc semantics of the subsystems. In order to define it, these semantics must be successfully combined in a single formalism. This thesis stems from the GEMOC project: a workbench that allows the definition of various languages along with their coordination properties, and target the formal modelling of the GEMOC core through the association of trace semantics to each preoccupation and the expression of constraints between them to encode the correct behaviour of the system.

This thesis follows several other works conducted under the TOPCASED, OPEES, QUARTEFT, P and GEMOC projects, and provides four contributions in that global context: the first one proposes a methodology to give an operational semantics to executable models illustrated through two case studies: PETRI NET and SIMPLEPDL. The second one proposes a formal context on which refinement can be expressed to tackle vertical separation. The third one gives a denotational semantics to CCSL, which is the language that is currently used in the GEMOC projects to express behavioural properties between events from one or several models, possibly heterogeneous. Finally, the fourth one proposes an investigation on how to extend CCSL with the notion of refinement we proposed. All these contribution are mechanized in the AGDA proof assistant, and thus have been modelled and proven in a formal manner.

P.2.2 French abstract

Les systèmes cyber-physiques sont des systèmes habituellement complexes et souvent critiques, dans le sens où leur défaillance peut avoir des impacts négatifs significatifs sur des vies humaines. Lors de leur développement, il convient donc de mettre l'accent sur les phases de validation et vérification (V & V) afin de prouver que le système satisfait sa spécification et les exigences de l'utilisateur et que les cas d'erreur pouvant conduire à des accidents ne se produiront pas. Dans la mesure où ils sont souvent très volumineux et complexes, leur développement repose habituellement sur des procédés dits de séparation des préoccupations. Cela consiste à modéliser le système de manière hétérogène, avec différents modèles qui doivent ensuite être combinés pour rendre compte du système dans son ensemble. Ces séparations des préoccupations peuvent être de différentes natures : horizontale, ce qui revient à séparer le système de manière structurelle en sous-systèmes ; verticale, ce qui revient à séparer le développement d'une partie du système en plusieurs étapes allant de la spécification abstraite à l'implémentation concrète ; et enfin transversale, ce qui consiste à regrouper ensemble les différents aspects du système qui participent de la même thématique (fonction, performance, sécurité, sûreté, ...). Usuellement, les différentes parties du système sont modélisées avec des langages métier dédiés tandis que les activités de V & V sont effectuées soit par tests et relectures, soit par l'approche que nous utilisons : les méthodes formelles.

Dans tous ces cas, les activités de V & V doivent prendre en compte ces séparations afin de fournir une confiance dans le système complet se basant sur la confiance en ses constituants. En d'autres termes, afin de prouver la conformité du système global, il faut lui définir une sémantique comportementale qui doit prendre en compte les sémantiques ad-hoc des constituants du système. Pour définir cette sémantique, il faut parvenir à regrouper toutes ces sémantiques intermédiaires dans un même formalisme. Cette thèse se place dans le cadre de GEMOC, un environnement permettant de développer des langages ainsi que leurs propriétés de coordination, et propose de modéliser formellement le cœur de GEMOC en associant à chaque préoccupation une sémantique de traces tout en exprimant les contraintes liées à leur composition afin de représenter le comportement global du système.

Cette thèse se place dans la continuité de travaux conduits dans les projets TOPCASED, OPEES, QUARTEFT, P et GEMOC, contexte dans lequel elle propose quatre contributions : la première propose une méthodologie permettant d'attribuer une sémantique opérationnelle aux parties exécutables du système, en traitant deux cas d'étude : PETRI NET et SIMPLEPDL. La deuxième propose un cadre formel pour exprimer des propriétés de raffinement afin d'exprimer les liens tissés par la séparation verticale des préoccupations. La troisième consiste à donner une sémantique dénotationnelle à CCSL, qui est le langage utilisé dans le projet GEMOC pour exprimer les propriétés comportementales liant des événements associés à une ou plusieurs préoccupations. Enfin, la quatrième propose d'ajouter à notre modèle formel de CCSL notre notion de raffinement afin d'en analyser l'impact. Toutes ces contributions ont été mécanisées et vérifiées dans l'environnement formel AGDA.

P.3 Content of the manuscript

The work depicted in this PhD document follows several paths that all contribute to the global purpose, which was presented in the abstract in Section P.2. Each of these paths answers part of the original issue in different, yet intricate ways. The outline follows the different contributions while also adding some background information about the tools and theory we used.

P.3.1 Detailed outline of the chapters

Introduction Chapter 1 focuses on the main global aspects justifying this work. It introduces several fundamental elements on which this thesis is grounded, including the notion of heterogeneous systems, vertical and horizontal separations of concern in their design while also placing the purpose of this work in that context. It aims at introducing the different notions inherent to this work that will be used throughout this manuscript. While this global context is wider than what this work tackles, it is fundamental in the understanding of the challenges at stake. Chapter 1 is split into two different sections. Section 1.1 gives the global context in which our work takes place, while section 1.2 places our different contributions in that context.

Formal methods, dependent types and AGDA Chapter 2 focuses on the core technical aspects of our methodology, formal methods. The work we conducted is rooted in the use of formal methods. On the one hand, formal methods have dictated our procedures on how to handle some categories of problems. On the other hand, our issues have led to precise and deep use of formal methods, thus allowing new users to be introduced to the field. We explain what formal methods are according to us (a very strict definition can never be formulated because it cannot be consensual, apart from the use of mathematics to model and prove the correctness of systems). We then narrow down the spectrum of these methods towards theorem provers as well as more precisely dependently typed languages in order to present the proof assistant used in our development: AGDA. Both the AGDA language and tools are depicted through examples, a short tutorial along with a detailed comparison to its cousin COQ. Since this document contains technical aspects related to AGDA, this section is fundamental for a thorough understanding of our work.

Library on globally unique lists Chapter 3 presents a library that was developed at the beginning of this thesis, and which is used in Chapter 4 to model the structural aspects of the event-based languages. This library proposes a notion of unique membership in a list which leads to the notion of globally unique lists, which are lists that only contain at most a single value that satisfy a given predicate. These globally unique lists allow us to create two specific instantiations: the maps and the sets (named bags for technical reasons). This chapter uses the depiction of this library to emphasize the methodology that is advocated in our use of formal methods: we define constructs which are directly verified using conformity properties.

Transition systems Chapter 4 introduces our first contribution in this thesis: the mechanization of transition systems in a proof assistant. Among these transition systems, we mostly focus on event-based systems although our approach is generic. This work relies on different kinds of semantics to define and compare languages. This section treats the modelling of transition systems in an operational manner, providing a structural depiction of these systems and functions which allow their temporal evolution. The goal here was to model both the structural part of executable systems and the proof that they can evolve in the same framework. This approach is first summarized and motivated in the context of our work and illustrated and validated by applying it to two executable languages: PETRI NET and SIMPLEPDL. Both languages will be presented along with a description of their mechanization with the focus being on the common denominators between them as well as the methodology used to build them. This contribution has been presented in a French event, FAC (Formal methods for Concurrent Activities), in 2015 [116].

Time and instant refinement Chapter 5 introduces the handling of time in concurrent systems. The notion of instants is detailed – with both a philosophical and a more concrete approach – as well as the notion of strict partial orders to bind these instants together. After introducing these elements and their mechanization, we present the second contribution of this thesis: the modelling of refinement in the context of the execution of different events in different systems. It gives a precise definition of instant refinement in asynchronous systems. This contribution starts with the global description of refinement in this context and progresses towards the definition of a refinement relation applicable in such cases. This definition is both presented conceptually (through the reasoning that led to its definition) and formally in AGDA. Some properties to ensure the correctness of this definition are described, proved and discussed. This contribution has led to two publications: a first one in October 2017 as a student paper in the junior workshop of RTNS (JRWRTC) [115] and a second one in the REFINE workshop under FLOC in July 2018 [119].

Mechanization of CCSL Chapter 6 focuses on the third contribution of this thesis: the formal modelling of the CCSL language whose goal is to describe and model interactions between events in the asynchronous execution of one or several systems. This language is described and its relevance towards our goal is discussed. Its existing denotational semantics is presented and discussed while our formal denotational semantics in AGDA is presented as well. The differences between these semantics is highlighted and explained by the use of formal methods in our work. Our mechanization of CCSL establishes some interesting properties which are fundamental in the process of assessing its correctness regarding the commonly accepted semantics of CCSL. This contribution has been published in the proceedings of MEDI 2018 that took place in October 2018 [118]. It had previously been prevented in a French national event, FAC, in 2017 [117].

CCSL and refinement Chapter 7 focuses on the last contribution of this thesis: the addition of a concept of refinement in the denotational semantics of CCSL. This chapter describes how our work on refinement has been applied to that language and how the CCSL operators properties have been proven with respect to our refinement relation. This binds together the two main notions of this thesis by combining an abstract representation of refinement (theoretically applicable to any language similar to CCSL) to an existing language in the same formal environment. This chapter proposes two different notions of refinement that are added to CCSL: 1-N refinement and 1-1 refinement. Both these refinements are motivated, modelled and their impact on the preservation of CCSL constructs is detailed and established.

Conclusion and appendix Chapter 8 brings a conclusion with assessments, limitations and perspectives for our work. Followed by references and commented appendices containing additional AGDA code that was either irrelevant, too verbose, too complex or too trivial to be part of the main matter of this document.

P.3.2 Dependencies between chapters

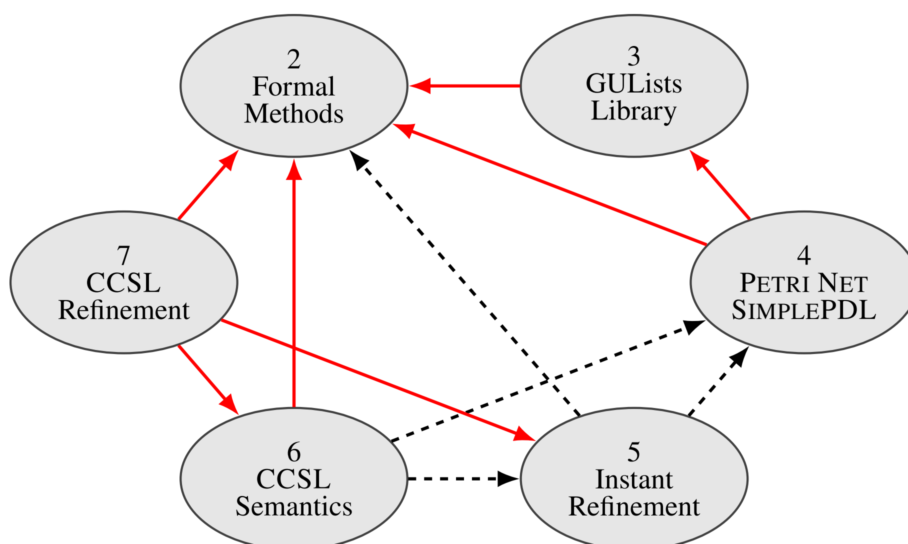


Figure 1: Dependencies between chapters

The chapters of this document are related in a way that is depicted in Figure 1. Dashed black arrows depicted a weak dependency: it is advised to read the chapter the arrow points at before the one the arrow comes from. Red arrows depict strong dependencies: it is mandatory to read the chapter the arrow points at before the one the arrow comes from. Note that reading this document in a linear order is a solution that satisfies all the requirements of this graph. The introduction and conclusion are absent from this graph because they are by nature bond to all the other chapters.

P.4 Lecture grid

There are a certain number of elements that are important to the reading of this document which should be pondered and read thoroughly to understand how this work was conducted as well as how the results are displayed and explained. These elements are mostly focused around the AGDA code present in this document, but also embed information on the choices made in the layout of this document.

P.4.1 On the layout of this document

Assessments This document contains a conclusion which provides assessments, limitations and perspectives for this work. Each chapter also contains a small conclusion which brings up assessments over the chapter itself, while summarizing the most important aspects and results that were discussed. Each chapter also comes with a small outline which gives the reader a foretaste of what to expect of the current chapter. The assessments of the chapters are the counterparts of these outlines, both of them wrapping up the content of the chapter.

State-of-the-art While this document contains an introduction which provides a context for this work as well as the contributions that were made inside this context, and a complete introduction on formal methods, the state-of-the-art aspects of this work are purposely not limited to these introductions. Rather, they are disseminated throughout the manuscript. This was intended as a way of placing precisely each contribution in their context, as well as a way of not boring the reader with very long and sometimes hard to read state-of-the-art chapters. For instance, the notion of model-driven engineering is mentioned and roughly explained in the introduction, but is described more thoroughly when talking about CCSL.

Chronological order This document does not follow the chronological order of the works it depicts, and is rather structured in a thematic way. Historically, the first work that was addressed revolved around PETRI NET and maps, which were then generalized with globally unique lists and SIMPLEPDL, after which the work on CCSL started. Then, the notion of instant refinement was introduced and eventually added to our mechanization of CCSL. Finally, when writing this document, a lot of additional aspects have been developed and added to all of these contributions, which are yet to be published.

P.4.2 On the AGDA code in this document

AGDA's ubiquity in this work The reader will notice that the concrete parts of the work have been implemented using the proof assistant AGDA. While projects that are based heavily on a specific tool might want to hide the technical aspects related to this tool, I chose to display and discuss most of it because it was a substantial part of my work. This manuscript contains a lot of AGDA code, references to AGDA as

well as didactic elements about this tool and language. While AGDA's community is growing at a decent pace, it is a lot harder to find information about it than other well-known similar tools such as COQ and ISABELLE/HOL. This means that this work is based on a significant, ever-lasting effort around learning and mastering AGDA. Thus, this manuscript can be of help to any reader who would like to make the same journey I did. While I cannot ensure that all that has been done in AGDA was implemented in an optimal manner especially my earlier work on transition systems, I have made a substantial amount of research in order to perfect my mastery of the tool as well as the theory on which it is founded. I have slowly made my way through the various updates the tool and the language have received. Through the complex, intricate and ever-growing standard library provided by AGDA's developers as well as through the alternate confusion and amazement which comes when using such a tool. This journey has been complex and time consuming making it natural that AGDA takes an important part in this manuscript. However, this is not the only justification for the overall presence of AGDA in this thesis.

Formalization in AGDA While mathematicians like to present their results in the mathematical language, us computer scientists find it relevant to present our results in our version of mathematics. Most of the results presented in this work (properties, definitions, axioms, examples...) are presented in the AGDA language. This is not merely a choice, but rather the expression of a deeper truth: they are not usual mathematical results, which would implicitly be anchored in set theory and only displayed by AGDA. Rather, they are AGDA's results, in the sense that they are defined and given a meaning by AGDA. AGDA is the formal foundation which allowed these results to be either defined or proved using both the internal core of AGDA or other notions that have been defined using AGDA. AGDA not only provides a formalism to define mathematical entities, it also provides a convenient language to express these definitions in a way that should not disorient the reader used to common mathematical definitions. While I intend no criticism to mathematicians whose work is the very reason why we are able to develop and use tools like AGDA today, defining and proving notions using AGDA has the immense upside of granting a greater correctness than the one provided by a pencil and a sheet of paper. Granted, this correctness is relative to AGDA's correctness itself, which leads to the inevitable infinite loop of correctness, which makes any correctness relative to the correctness of some other entity, whose correctness is also dependent on something else's correctness and so on. All in all, providing AGDA definitions rather than usual mathematical definitions is advocated and applied in this work.

AGDA's signatures and bodies While a lot of code snippets are present in this manuscript, to emphasize a definition or a property, their complete understanding is not mandatory to the overall comprehension of this work. AGDA is a functional language, as such, the AGDA snippets will mostly be functions which usually provide a proof term which means they will contain a signature and a body. The signa-

tures are always relevant in the context of this work as explained in Section 2.2, because they usually represent properties that have been proven and thus should not be skipped. Although they will usually be paraphrased in natural language, the body, which represents the proof terms can however usually be skipped when the reader is not interested in the process of building proofs in AGDA, which is also emphasized in this document. Since the signatures are syntactically close to the usual mathematical language, their understanding should be easily possible. On the contrary, understanding AGDA's bodies is often both very complicated – even for the mere people that wrote it using the help of the AGDA environment of development – and also moderately relevant in terms of understanding the overall theory that is being developed. All that matters is that said bodies have been type-checked accordingly to the signatures they are bound to.

Missing parts of code These bodies, as explained, are left in a didactic purpose in the code snippets to emphasize different ways of building terms or using AGDA overall. However, when they are too long or simply too irrelevant they have been cut out of the main part of this document, but they can still be found in the appendix. Every time this happens, a reference to the related section in the appendix is made to ensure that the curious reader can find the missing piece of code. However, there are some parts of the code that cannot be found anywhere in this document, which are the imports of the modules. The relevant import have usually been shown especially when they feature renamed imports but most of them have been hidden because of their poor overall relevance. As for the bodies that have been shown, they have usually been reworked and factorized to be as concise and elegant as possible, using for instance function composition and anonymous functions. These terms have usually been built differently before being refactored: it is illusory to think that such elegant factorized terms can be provided in one go by the programmer.

Colouration The snippets of AGDA code are colourized using the following conventions: orange for AGDA keywords, blue for types (either simple, dependent and / or polymorphic), green for data type constructors, black for parameters and indices (basically values, or types used as values), purple for atomic values, such as natural numbers coming from the standard library, and module names, brown for comments, strings and chars, and finally pink for record fields.

Literate AGDA AGDA provides a very convenient feature which allows us to mix AGDA code and LaTeX code in a single file. The AGDA compiler can then be called on said file to create a LaTeX file where all the AGDA code has been transformed into LaTeX, while the LaTeX code remained unchanged. All parts of this document that contain AGDA code – expects when it appears as figures – have been written using literate AGDA. The main consequence is that the code in this document is the actual AGDA code that has been lexically and syntactically checked as well as typed-checked. As a consequence, all proofs we provide are proof-checked.

Chapter 1

Introduction

1.1 Modelling heterogeneous systems: a challenge

1.1.1 The need of verifying critical software

History of software Software engineering as a discipline has been developed since the middle of the 20th century. Most computer scientists agree that the 1960s, and especially the 1968 convention held in Marktoberdorf, Germany [34] paved the way to where software engineering stands now: present in every single field and company. The fact that such an explosion has occurred in only 60 years is purely extraordinary, which is why some entomologists even consider that the birth of software engineering marks the beginning of a new era called the information era. This era is characterized by the ubiquity of software not just for companies but also in our everyday lives. The impact of software's evolution in our lives has been studied and described in several works, such as [153]. The rate of this evolution continues to grow as shown with trendy notions such as deep, machine learning as well as the trendier quantum computers. While we believe that the future induced by this evolution will be bright, it can only be so provided we have the ability to validate and verify the software being made, in other words, to assess that it satisfies its user requirements. Since formal methods intend to provide a mathematical account of software, this makes them an important part of current software evolution, even though they might be less trendy and less known to the general public.

Ubiquity of software As seen before, software is now ubiquitous and constantly involved in various interactions with other entities: software through logical interfaces, human beings through user interfaces and the outside world through both sensors, which provides the software with data, and actuators, which control various physical mechanisms. The software from the last category are called cyber physical systems (CPS) [95]. The actuators in these systems have their output constantly adjusted using the data received from their sensors. Such software covers an extremely wide area of domains used by a large number of companies. These

ones, and sometimes public entities, may require some strong guarantees regarding the provided software, especially when its failure can have a strong negative impact, and possibly cause the loss of human lives. These systems are attributed the name "critical" and are especially present in fields such as aeronautics, cars, railway, space exploration, energy transformation, healthcare and factory automation.

Formal methods and critical software While failures are sadly assumed to be a regular occurrence in software such as the ones developed for personal computers or entertainment systems, this is not the case for critical systems that must satisfy a considerable amount of requirements to avoid such failures. Should these requirements not be satisfied, tragedies could happen, as many have happened before: the famous crash of the Ariane 5 rocket in 1996, due to a simple integer overflow [91], the crashes of the Boeing 737 Max in 2018 [145], the accidents with the Therac-25 radiation therapy machine from 1985 to 1987 [100] or the failure of Toyota's ETCS system in 2009 [88]. The use of verification techniques to avoid failure is very costly both in terms of money, time and manpower, which explains why "bugs" [34] are so common in our everyday life, since this process is usually not applied thoroughly in the development of these software. However, it cannot be avoided when developing critical software. For the past half-century, the approach that was advocated for that purpose was systematic testing and proofreading. The critical systems were heavily reviewed and tested, and were considered fault-proof when the test outcomes were all correct and when all reviews agreed that the development was appropriate. While this process indeed builds a certain level of confidence in the developed software, it cannot be enough to ensure that it fully respects the requirements in each and every case that can possibly be encountered in real life. Indeed, as Dijkstra himself noted in 1969 with his famous quote, "Testing shows the presence not the absence of bugs" [34], test cases are limited to what the developer can think of and are in finite numbers. Even if this number is large, nothing proves that hypothetical future tests would pass. This is one of the main reasons why formal methods have been developed and are slowly gaining momentum in safety critical industrial uses [144]. These methods aim at proving that the modelled version of critical software always satisfy their requirements. This is a very difficult and tedious process because real-life software is complex and often much more so than what can reasonably be modelled and verified in a single step, which is why various separation approaches are usually used in their development.

1.1.2 Separations of concerns

Complexity of software Complexity is a major issue in the development and especially in the verification of any software, and even more for critical software. Complexity is ubiquitous in computer science, even in the simplest algorithms. We can distinguish two different kinds of complexity: the complexity during the execution of a given program, and the complexity of the problem which has to be modelled and implemented. The first kind of complexity is studied in a field usu-

ally called complexity [150], the aim of which is to assess how much time and memory space will be taken by the execution of a given algorithm depending on the size of the data it processes. People with little knowledge of computer science usually have the intuition that a program which takes more than a few minutes, let alone a few days, to complete its task is abnormal. However, such duration is actually neglectful compared to algorithms which would take more than the supposed age of the universe (13.7 billions of years) to do so. Similarly, there is a physical limitation on how much data can be stored in a certain volume [28], which would easily be overcome by certain combinatorial algorithms of exponential memory usage. The second kind of complexity from the theory of complex systems is wider and encompasses a large number of possible cases. It was called algorithmic complexity [101] by its inventors Kolmogorov and Chaitin and is related to the size of the simplest algorithm in a given language that can implement the system. The system in question can be huge in size, and provide a sophisticated service ; it can be composed of numerous small size elements which are required to interact with one another in a correct manner ; finally, it can simply be a software of arbitrary size that displays a very complex behaviour which translates into a deep conceptual challenge for the developers. This last kind of complexity, reflecting the complexity of the problem to solve, is very concrete, especially in CPS where it can often be found in its various incarnations. Indeed, CPS can be as big as a rocket, or even bigger such as the railway system for an entire city or even country.

Separating concerns While developing and assessing such complex systems, the notion of separation of concerns becomes mandatory, as first introduced by Dijsktra himself in [61]. A concern is a specific element of a system that must be handled during its development. As for separation of concerns, while the name being somewhat self-explanatory, it hides a higher level of complexity depending on the nature of the concerns that are separated from others. Indeed, these concerns can be of various nature: functional, physical, logical, abstraction, human, sociological, ergonomic, psychological, economical, ethical, . . . and can refer to various macroscopic elements: provided service, provided quality of service, . . . including the process, methods, tools for the development itself. Whatever concerns are separated, doing so is a key asset in the development of these ever more complex systems, because it allows us to handle parts of the global system separately rather than the whole system in one go. Usually, concerns are intertwined and every effort to isolate, separate them and express their relations in the simplest way possible has been shown to ease the design. There are three main kinds of separation of concerns that we will target, that are usually called horizontal, transversal and vertical. Horizontal separation of concerns is the more natural and common of the three, since it consists in applying a divide and conquer strategy that splits a problem in subproblems recursively until reaching problems that are small enough to be solved efficiently (so-called top-down development strategy). This separation is usually called Component Based System. The result of this separation at a given step of

its development, is the system's architecture that describes the system as an assembly of components. The complexity of the components is reduced as their size is reduced however they usually mix different issues to be solved: provided services, provided quality of service, etc. The transversal separation isolates each of these issues to handle them separatly with the most appropriate tools relying on so called domain engineering. This separation is usually called Aspect Oriented Engineering. The last one, vertical separation, is less natural but nonetheless mandatory: it consists in separating a given development into different steps from an abstract specification to a concrete implementation. Separation of concerns has been widely studied in the literature and applied to several domains which emphasizes the universal aspect of this methodology. It is strongly related to the so called Product Life Cycle Engineering. Examples of works using separation of concerns explicitly are as follows: verification of Railway signalling rules [94], software architecture [110], epidemiology [8], multimedia systems [120] and concurrent programs [154].

Preservation of properties Since separation of concerns is such an important aspect in the development of CPS, it is mandatory to assess how these separations can be handled through verification and validation activities. Indeed, splitting the systems has the important upside of allowing us to describe each concern separately from the others, but it also implies that these parts must at some point be composed together. This means that there is a concrete need of compositional verification, which means to assess how this composition impacts the properties that were verified by the sub-parts of the global system. This need is different when considering separations from different natures, which are described in the following sections.

1.1.3 Horizontal separation: components everywhere

Nature of horizontal separation The first commonly used separation of concerns is called horizontal. When doing such separation, the system designers split the problem that will be solved by the system under design into different parts, each of which serves a different purpose. These parts can then possibly be split again if needed, in order to have a set of small and manageable pieces. For instance, a global product can be split into almost autonomous yet collaborative systems and the whole architecture is called a system of systems. Each of these systems can then be split into sub-systems that are less and less autonomous one from the other, until reaching atomic equipments that will become physical entities at the implementation level, and so on as the equipments can again be split as components. These various parts can usually be seen as black boxes with a well specified interface allowing us to connect them with the other boxes contained in the whole system's design. Usually, the various parts of the horizontal separation are all expressed using the same languages at all stages of their design, except for the last ones. Indeed, these last stages are usually handled through the use of languages specific to the physical technology of the given component of the system. Thus, the final product is the result of the integration of many components that rely on many different tech-

nologies modelled using different languages. These kinds of integrations are called heterogeneous.

Benefits of horizontal separation Horizontal separation is widely used since it provides a very concrete benefit: It allows the system engineers to develop their parts of the system almost independently one from the other and then to assemble the various parts to build the full system. They can rely on their own languages, methods and tools (i.e the most adapted to their goal), which are usually called Domain Specific Modelling Languages (DSML) [69] and which allows them to describe specific concrete parts of the system that correspond to a specific goal inside its global function. These DSML can be very technical and require a certain level of expertise to be manipulated, which leads to the main advantage of this approach: horizontal separation helps the system designers focus on what matters in their work since they do not need to always have the global system in mind. As this global system can be very complicated, it is unreasonable to assume that a single human being would have the technical knowledge to specify and define each constituent of this system. This is why separating the concerns horizontally is so important, and even mandatory. Since each designer focuses on their area of expertise, their only constraint is to provide the rest of them with a precise interface for the component that they are designing.

1.1.4 Transversal separation: aspect oriented design

Transversal separation Horizontal separation, as presented in the previous section, is purely structural, in the sense that each part usually corresponds to a structural element in the global product (systems, equipments, components, ...). However, it is mandatory to take into account the behavioural aspects of these parts. Usually, this is done as follows: the system is split into different elements as explained, and then each of these elements is split once more into its behavioural and structural aspects. This is particularly relevant because the structural aspects are usually stable through composition using the interfaces of its different constituents, but the behavioural aspects usually need additional properties to be transferred into the behaviour of the global system. This is why separating these two aspects is very important. This additional separation is called transversal. It is called this way because it crosses the others in order to unite parts that are thematically identical, such as the behaviour in this case. This leads to models that are heterogeneous in three different ways. Such an heterogeneous modelling (different DSML for each part of the systems, whether it is structural or behavioural) has been integrated in various development environments, such as the Ptolemy toolset proposed by Lee et al. [33], the ModHel'X toolset proposed by Boulanger et al. [78] and the GEMOC studio proposed by Combemale et al. [43].

Benefits of transversal separation Transversal separation has become widely used since the introduction of Software engineering and more precisely MDE

(Model Driven Engineering) as it provides a very concrete benefit: It allows the designers to handle each aspect of the system with the DSML that is the most appropriate. As previously stated, these DSML can be very technical and require a certain level of expertise to be manipulated, which leads to the main advantage of this approach: transversal separation, as horizontal separation, helps the system designers focus on what matters in their work since they do not need to always have the global system in mind. More on such separation is discussed in Chapter 6. The nature of the parts usually depends on the nature of the system as well as the designers' will. This means that the purpose of each part also depends on these factors. What matters is that each of these parts corresponds to a well identified conceptual notion in the designer's mind.

1.1.5 Vertical separation: refinement on the rescue

Nature of vertical separation Vertical separation is a lot harder to define and grasp, although it is definitely as useful as its horizontal and transversal counterparts, which we name planar separations. While planar separations are handled during the design process, and precedes implementation, vertical separation takes place during the implementation process. More precisely, it builds a bridge that goes from one to the other. Should the system be split into several black boxes through planar separations, and should these boxes be defined by their interfaces – their specifications – then vertical separation consists in splitting the development of a given box into different steps going from said specification to the concrete implementation of this concern. This means that vertical separation usually enforces a refinement relation between the different models of the same part of the system in order to ensure their consistency regarding the requirements of the current concern. The notion of refinement, described in [15], consists in developing a system step by step, each of which should preserve correctness towards the expected specification.

Correct-by-construction refinement As explained, vertical separation can be seen as a succession of refinement steps from the abstract specification to the concrete implementation, as first introduced by C. Morgan in [121], and as advocated by the B [2] and Event-B methods [3] proposed by J.R Abrial. In order to prove the preservation of the properties throughout these steps, these methods use invariants that are defined at each level of refinement. The proof that the invariant at a given level implies the invariant at the next level has to be provided to ensure the consistency of the process. This approach is very operational, in the sense that it considers vertical separation as a process of construction. Indeed, an operational semantics gives a meaning to a system by building its output and stating that the semantics of the system is the output it produces. For instance, one can give an operational semantics to an automaton by computing the language it accepts.

Assessing a correct vertical separation Another way of seeing vertical separation consists in assessing that different systems can indeed be seen as two stages of

a same development. This approach is more denotational – or relational – and any operational semantics, such as the ones proposed in the B and Event-B methods should provide models that satisfy such relations. Indeed, a denotational semantics proposes to give a semantics of a system by specifying relations on the elements it provides. For instance, giving a denotational semantics to an automaton consists in giving the regular expression which represents the language accepted by the automata. More generally, any trace provided by the operational semantics of a transition system should comply with this denotational semantics. This denotational view of vertical separation is usually related to a notion of simulation (either simple simulation, bisimulation or weak bisimulation). For instance, weak bisimulation, which is a bisimulation with the addition of invisible transitions called τ transitions, is very akin to represent refinement because these invisible transitions are perfect candidates to be refined, as advocated in our approach in Chapter 5.

More on refinement Vertical separation, in other words refinement, has been widely studied [134, 135]. Synchronous refinement has been studied in the case of synchronous models of computation (MoC) first as oversampling for data-flow languages [113] and then as time refinement for reactive languages [72, 108]. Polychronous time models have been used to assess the vertical refinement during system design [142]. Refinement has also been widely implemented for many different modelling and programming concerns like data [55] and algorithms (sequential [17], concurrent [16], distributed, etc). Time can be represented with a single global reference clock that binds all clocks in the system together [106, 32]. However, since building these global clocks is usually tricky, time is more often abstracted as a partial order relation [133, 105]. Refinement [1] then relies on simulation [79, 80] or bisimulation relations between the semantics of the more abstract and concrete system models.

1.1.6 Challenges in heterogeneous modelling

This context provides numerous issues in the process of modelling and verifying heterogeneous systems. These issues are often related to the handling of the composition between the parts of a given system, but not limited to them. Indeed, this heterogeneous modelling requires the development and existence of numerous modelling languages, the depiction of various behavioural coordination properties between these languages, the handling of the structural heterogeneity of the models as well as the handling of the refinement between languages and models. All of these require the definition of tools and environment in which such definitions and handling can be made, such as GEMOC and PTOLEMY. In this context, we propose several contributions, which are depicted in the following sections.

1.2 Our contributions

1.2.1 Our goal: mixing separations over behavioural properties

This thesis targets a proof based modelling and verification approach to prove properties over languages and models in such tool-sets. We intend to tackle the verification of the heterogeneous aspects of such complex systems. Our work takes place in the context defined by GEMOC that mixes the three main separations of concerns. Indeed, GEMOC allows us to define the DSML used to model the various parts in a CPS in each phase of their development. Thus, DSML are combined both in a horizontal, vertical and transversal manner. GEMOC first defines atomic events as methods on the meta classes defining the abstract syntax of the language, which updates the state of the system, and then relies on the UML MARTE CCSL (Clock Constraint Specific Language) to model both the MoC (Model of Computation) [138] for the various DSML [44, 57, 93] and the coordination between DSML using the Behavioural Coordination Language (BCOoL) [92].

In the context of a tool-set which combines all the different aspects required to model complex and heterogeneous systems, we focus on the behavioural aspects of these models, which correspond to elements that evolve with time, and for which we intend to give a formal context. Behavioural aspects are an element of transversal separation, and are themselves horizontally separated one from another. We intend to target both this horizontal separation of concerns by giving a semantics to CCSL, which provides such horizontal heterogeneity naturally, and vertical separation by assessing the relations between the various time concerns in the models of the same system part. Ultimately, we intend to mix these two separations of concern by providing a framework on which time constraints can be expressed both horizontally and vertically in the behavioural transversal concern. In this context, we would like to assess how constraints expressed at a given vertical level can be transferred to other levels. This would contribute to answer a fundamental question in system engineering, which was introduced in Section 1.1: how time constraints over different horizontal and vertical parts of a complex systems can be merged together successfully, such that the global behaviour of the system can be assessed correctly ? In order to provide such an answer, we conducted several investigations, some more fruitful than others, but all contributing in a way to this global purpose. These contributions are depicted and introduced in the following sections.

1.2.2 A formal execution of event-based systems

A first question that was addressed was the following: how can one formally model the execution of models whose semantics is a transition system ? That is, a model composed of a state which can evolve through the execution of transitions. This corresponds to the atomic events in GEMOC introduced previously. To our knowledge, there are no such attempts in the literature, even though some similar works can be found, such as an attempt at verifying such systems using both tests

and formal methods [146] and an attempt at modelling and verifying concurrent systems using CCS process algebra [111]. This initial contribution had a double objective. The first one was to get familiar with the stakes of this thesis, the formal modelling of languages as defined in the GEMOC tool, as well as the formal tool AGDA, on which more information can be found in [125], [107] and [30], and which is based on the intuitionistic type theory by Martin-Löf [109].

The second one was to provide a concrete methodology with which instances of a given language can be defined and verified in a formal environment, provided that this language can be given a semantics as an event-based system. This verification consists of two aspects: a structural aspect, which guarantees that only models that are correct towards their meta-model and their static semantics can be built, through a correct-by-construction approach ; and a behavioural aspect, which consists in assessing the properties that need to be satisfied in order to execute a given transition. In other words, this consists in assessing if the guard of the transition holds, which would allow the temporal evolution of the system.

A concrete evolution of the system can then be emulated in a step by step manner, giving a finite trace for the system. In other words, this step-by-step approach provides a correct-by-construction operational semantics for models which can be defined as transition systems, by computing at each step the possible evolutions for the system, and possibly executing one of them. This approach has been applied and validated on two languages which exhibit this behaviour: the PETRI NET and SIMPLEPDL. This is the only contribution which relies on an operational semantics, because we think it is always fruitful to start by studying some operational aspects before trying to describe denotational semantics, since they are far more abstract, and when dealing with abstract entities, it is usually mandatory to have in mind which concrete elements they are related to.

1.2.3 A formal depiction of trace refinement

The second contribution we propose answers the following question: is it possible to give a formal counterpart to system refinement based on their trace semantics ? This question arises from two different observations: first, there exists a specific relation between the two languages that were used as support for the previous contribution: SIMPLEPDL and PETRI NET. Indeed, these two languages are bound by a relation of weak-bisimulation, as established in [41], which can be seen as a relation of refinement, as mentioned in Section 1.1.5. Secondly, the language which provides a way of expressing synchronization constraints between languages in GEMOC, CCSL, lacks a notion of refinement.

Refinement is often seen as a partial order between systems, such as in [52] or [82]. In most publications around the subject however, it is not quite clear exactly what are the formal notions being refined. In other words, it is not always easy to understand what exactly are these entities that are being partially ordered. In addition, this partial order is often considered true by default, with no accent being made on the axioms of partial ordering being verified by said relation. All in all,

trace refinement lacks a proper formal definition along with a formal proof that it is indeed a partial order. This contribution aims at closing this gap: we try to assess what are the entities that should be bound by partial ordering when considering a refinement between the traces of systems. We give a depiction of a relation of refinement, which is meant to correspond to the usual notion of trace refinement which has not yet, to our knowledge, been formally expressed. This definition of refinement is then proved to be a partial order.

This contribution provides a formal refinement relation mechanized in the AGDA proof assistant. This relation of refinement is based on a formal notion of time that has been modelled using the same tool. Similar formal mechanization of time models has already been done using other formal methods, for example [76] uses Higher Order Logic in Isabelle/HOL; [71] and [130] use the Calculus of Inductive Constructions in COQ, a tool described in [24].

1.2.4 A denotational semantics for CCSL

The third contribution of this thesis focuses on CCSL, while answering the following questions: is it possible to give a formal denotational semantics of a language that allows us to express behavioural constraints between languages ? If such, what fruits would such a semantics bear ? And what would the differences with the paper version of the semantics of such language be ? Indeed, CCSL already has a denotational semantics which was done on paper, and which can be found here [56]. As for CCSL itself, it was first presented in [7]. CCSL allows the expression of constraints between clocks, which are abstract entities that track the occurrences of events of the same nature in a system. This contribution also answers the following question: Are there properties between CCSL constraints which can be deduced from a given specification ? Are CCSL constructs part of some underlying mathematical structure ? Other attempts at giving semantics to languages like CCSL have been developed, such as a promising approach to give an operational semantics to TESL using ISABELLE/HOL that can be found in [148], or an encoding of CCSL in first order logic to assess properties with SMT solvers [157].

We provide a denotational semantics of CCSL, which differs from the paper version through the mechanization process. CCSL handling of time is based on the notion of TimeStructure [152] which was modelled accordingly with the notion of time that is used in the contribution around refinement presented in 1.2.3. This mechanization is meant to provide a bridge between the commonly accepted semantics of CCSL and the actual current implementation of the language. This bridge goes both ways: the properties that are established reinforce the confidence we have in our modelling, and this modelling gives us additional information on the semantics of CCSL as well as the properties that should hold for the requirements to be verified. This is a concrete example of the merits of such formal approaches: the modelling comes from an existing notion, and then enriches it, provided it is conforming to the basic requirements of the language.

1.2.5 An investigation on the impact of refinement over CCSL

The last contribution of this thesis focuses on the combination of our notion of refinement with our semantics of CCSL. Both mechanizations have been modelled using the same tool, and share the same basic model of time. In this contribution, we answer the following question: is it possible to add a notion of instant refinement to CCSL ? If such, which form would this refinement take, and which fruits would it bear ? CCSL has been designed to handle horizontal separation in the behavioural concern in an elegant manner, because it abstracts all the events into clocks, regardless of which part of the system these clocks come from. Adding a notion of refinement to CCSL would allow the language to handle all main separations of concerns in a single framework, which is provided by this contribution.

Similarly to some of the upsides of the approach depicted in Section 1.2.4, we investigate which constructs provided by CCSL are preserved by refinement. For that purpose, this contribution provides two additional CCSL relations which correspond to two different multiplicities of refinement: 1-N and 1-1. These new relations only make sense in a context where several layers of refinement are present, using our formal depiction of refinement presented in Section 1.2.3. Thus, the addition of refinement to CCSL provides both an enrichment of the current formal context of CCSL which allows us to express refinement, and two concrete refinement relations in CCSL, along with properties of preservations for these relations.

1.2.6 Subsidiary contribution: AGDA methodology

All these contributions have been mechanized in the AGDA proof assistant, and all the results that are presented in this document have been proved in this formal context. While we do not contribute to AGDA itself, this document presents an underlying contribution in terms of understanding and using AGDA. It proposes several ways of using the tools in an as optimal as possible manner, as well as a lot of tools for AGDA beginners to learn the language and for current AGDA users to maybe discover new ideas on how to build proof terms in AGDA. The range of this contribution is hard to assess, because it is difficult to be sure that such or such methodology is unknown to AGDA experts. However, we believe that throughout this document several AGDA proofs could be of use for further AGDA users, which is why we made a strong effort to provide as many explanations as possible along with these proofs. We believe that this document is self-contained in terms of AGDA notions and that a non AGDA expert can find all the elements that are needed to the understanding of the technical aspects of this work. This contribution is less straightforward and not linked to our global context and goal, but exists nonetheless.

Chapter 2

From formal methods to AGDA

Outline

This chapter proposes an overview of formal methods which is slowly narrowed down to the language that was used in this work: AGDA. It uses the following outline:

1. Section 2.1 presents a possible classification of the currently available formal methods. It treats model checking, static analysis, abstract interpretation, correct-by-construction approaches with the use of refinement, automated theorem provers, proof assistants and finally tools which aim at combining several of these methods. Each class of methods is briefly described and given a field of application.

2. Section 2.2 presents the notion of dependent types and explains why and where such a type system finds its place into this categorization. It introduces constructive logics and explains how they are connected to type systems through the Curry-Howard correspondence. It gives a brief history of logic and explains the thought process behind the creation of languages using dependent and polymorphic types.

3. Section 2.3 presents the AGDA language and the tools that revolve around it. AGDA is a dependently typed language which is described thoroughly and compared to its cousin COQ through several angles. A short tutorial is given which, coupled to the rest of the section, should help the reader familiarize with the AGDA language which has been used throughout this work and which holds an important part in this document.

2.1 Formal methods: the banner of computer scientists

Attempts at giving a complete and consensual definition of formal methods always ends in vain. Some researchers consider that only the most advanced tools, usually based on complex mathematical theories, can be considered as formal methods while others take paper proofs as such methods. There is no actual answer on what they actually are, however, there is a consensus on what they try to achieve, thus making their goal the only effective way to successfully describe them. They aim at mathematically modelling and proving properties about programs and systems, thus being an alternative to verification by systematic testing and proofreading that can ensure the correctness and also the exhaustiveness of the verification. They can be categorized in many ways and a lot of writers as well as researchers have tried to elaborate the best categorization for these methods to clearly separate the aims of each of them, such as in [5]. This is a hard task as well because these methods' spectrum usually overlap with each other. This classification also depends on whether one sees formal methods as theories on which tools are built, or as the tools themselves, which makes a significant difference. For a sake of clarity and simplicity, we introduce the kind of formal methods we used through a very simple classification, that does not pretend to be neither exhaustive nor consensual for all these reasons.

2.1.1 Model checking: exploring state spaces

Model-checking approaches, described thoroughly by Clarke et al. in [37] aims at modelling a system as a state transition system [67] (the system is split into its possible states and the transitions that can occur from a given state during its execution. More precisely, a transition system is described by an internal state, which can be modified through the execution of transitions that specify when and how it might evolve) in order to explore the state-space of all possible executions of the abstracted system in regard to a given predicate – usually specified in a temporal logic (LTL, CTL, μ-calculus, CTL*,...) – that supposedly[1] expresses the requirements that must be satisfied by the system. The transitions of the abstracted system can be labelled, – have a name – and / or be guarded – requiring a specific predicate to hold in order to be activated. Such systems will be described and modelled in Chapter 4. Many automated systems that we encounter in our everyday life can be modelled as transition systems, whose transitions are labelled through the action that activates it. For instance, the automaton to retrieve money from a bank first requires the user to enter a code (the writing of which also requires a transition system to be run, which accepts four digits before validation) then to pick an action among the ones that are available, each of which are represented by a specific transition from the current state. After that, depending on the action picked by the user and the transition executed, another screen will appear with additional instructions. After which the user is asked to retrieve their credit card to exit the system, which

[1] the gap between the expected behaviour and the requirement will be discussed later on in chapter 6

then waits for another user. Model checking requires unfolding the execution tree of these systems to check, on each possible branch, the validity of the temporal property. Many formal tools embed a model checker. For instance, this is the case for TLA+, a language introduced by Leslie Lamport [90] allowing us both to define transition systems and prove properties on their temporal execution using the TLC model checker. This is also the case of the B [2] and Event-B [3] methods – more on these methods can be found in Section 2.1.3 – that are respectively implemented in the tools Atelier B [63], and Rodin [4] and that use the Pro B [99] [98] model checker. Interestingly, Pro B can also be used as a model checker for other languages, such as TLA+, which means that the core of the model checking approach is somewhat independent from the tools that are used to model the preoccupations, as long as the modelled system can be translated into a transition system. Another example is Alloy [85] which uses bounded model checking using SAT/SMT solvers as described later on. However, the main drawback of this approach is that it does not handle infinite state spaces which are common nonetheless as an abstraction of repetitive systems. However, there are a lot of works on folding these state spaces to build an equivalent that would be finite – an example of this would be symbolic approaches to handle specific patterns of infinite [140] – but there is no generic way of handling such infinite case. By building the state space, the model checker is a direct generalization of systematic testing since it aims at automating the testing of every single possible case of execution in all its possible states and execution paths.

2.1.2 Static analysis and abstract interpretation

Static analysis aims at proving properties about existing programs or executable models written in languages that do not natively embed verification elements, which is the case for most of the usual programming languages such as C, FORTRAN, Java and so on. It is however possible to retrieve some information on the programs written in such languages, depending on their semantics. Static analysis will exploit this semantics to extract such information. These information can be directly extracted for verification purposes, through methods such as Hoare Logics [81], where each instruction of the program is annotated with a precondition and a postcondition depending on the nature of the instruction. The preconditions must imply the postconditions for the function to be correct. This requires an assessment of the right preconditions (and sometimes variants and invariants) that are needed to ensure the correctness of the program, using for instance backward analysis to find the weakest precondition which would imply the postconditions. These preconditions, postconditions, variants and invariants are the basic bricks of what are called proof obligations. They are extracted from the annotations (usually automatically by adequate tools) and represent the minimal proof effort that has to be done in order to prove the correctness of the program. The tools based on the B and Event-B method also rely on proof obligations to handle the development of the systems.

This information and the associated semantics of the language can also be interpreted in a different mathematical space in which case the process is called abstract

interpretation, as first introduced by P. Cousot in 1977 [49]. The goal here is to be able to retrieve even more information by projecting the program in a more complex – but more resourceful – mathematical space (usually exhibiting a lattice structure) on which mathematical operations can be executed. The common example of abstract interpretation is to abstract the numerical variables into the mathematical set of intervals, which can be manipulated through the union and intersection, forming a lattice in that regard. After injecting the variables of the program in this world (assuming there is some initial information on the intervals of the variables), a reasoning can be conducted throughout the program to keep track of the possible intervals that the variables are constrained to during the execution of the program. This is very useful both for verification purposes (checking if the intervals at the end of the function are indeed acceptable) and to check for dead branches in the program. Such techniques, however, require an additional layer over the program itself. For instance, if a branch has been proven non-reachable through abstract interpretation, – the additional layer – will still appear and be tested in the execution of the original program, because there is no dialogue between the operation world (the program) as well as the verification world (the abstract tools used over it). However, some compilers, – the ones that embed some abstract interpretation – will erase the unreachable branches. In the rest of the section, other approaches will be shown to tackle this issue.

2.1.3 Correct-by-construction approaches: the use of refinement

A third approach in formal methods is to build correct-by-construction software through the use of refinement. Correct-by-construction approaches are not limited to refinement approaches. Proof assistants, described in Section 2.1.5 allow correct-by-construction programs to be built but these are the most representative ones. Correct-by-construction approaches rely on developing programs along with their corresponding proofs of correctness at the same time, which means that completing the development of such programs implies their correctness, whereas in static analysis the proof is built after the program is written and may require additional testing to locate errors when said analysis could not be fully done. This means that they need no further verification once they have been developed. This requires some well-designed tools and underlying theories to allow such a thing to happen. A good example of correct-by-construction tools is B [2] and later Event-B [3], two methods developed by Jean-Raymond Abrial. Event-B provides a framework to build machines coupled to contexts and invariants, through which a system can be built using several refinement steps. The specification is given at the highest level of abstraction and is then embodied more and more through the development of the actual computable model. This approach is highly advocated in the community because the proof effort is mitigated throughout the development rather than being done afterwards – and it can be automated to some extent. Refinement is not only an efficient programming paradigm that allows the development to be split in different steps, but it also tries to embed an accurate and real description of the developed

system. To be more accurate, each layer of refinement supposedly represents an actual view of how the system works, hiding purposely some inner details when the observable behaviour is sufficient to whoever needs to express or understand something about the system, as studied in Chapter 5. This means that developing systems in such a way is also very convenient for system engineers because they do not need to handle technical details in every layer of refinement. And each of the layers can later on be used to enlighten a given aspect of the system behaviour. Every layer introduces new requirements (the invariants of the systems), a preliminary implementation with the related proofs and details over the previous layers with their respective proofs. An example of such development can be found in [14] where the authors model in Event-B a process of on-line shopping through the use of carts after which they substitute it with another one in case of failure while preserving the global behaviour of the system.

The next set of formal methods that are discussed in this document are theorem provers. Roughly, they aim at modelling and / or proving properties through a set of axioms and a set of deduction rules. Among these provers, there exists a substantial amount that are able, in certain circumstances, to build proofs without requiring any user interaction. They are called automated theorem provers (ATP).

2.1.4 Automated theorem provers

ATPs are powerful tools that are based on a given logic. This logic can either be decidable, in which case ATPs will always succeed at their goal but the underlying expressiveness of this logic is quite limited (such as propositional logic). This success is, however, theoretical since there can be some complexity issue in the computation, even for the most basic logics. In propositional logics for instance, the complexity of assessing if a given formula of n variables can be satisfied is 2^n. They can also rely on stronger logics by using heuristics to try and achieve goals while providing no guarantee they will ever find the solution – which may simply not exist [74]. This second approach is the one usually advocated because the logics used in computer science are never fully decidable and usually require incomplete heuristics to tackle even the easiest problems. These logics are somewhat separated in two categories, first order logics and higher order logics (FOL and HOL) that have been widely studied and that do not embed the same expressiveness, as explained in more detail in Section 2.2. This is why these automated theorem provers are rarely used as stand alone programs but rather as parts of higher level proof assistants. A basic example of these automated theorem provers are propositional SATisfiability problem (SAT) solvers. SAT solvers are programs whose goal is to find if a given propositional logic formula is satisfiable under a certain valuation for the variables it contains. For example, considering the formula $P : (A \to \neg B) \wedge A$, P is satisfiable when $A = \top$ and $B = \bot$ which should be returned by the solver. However, considering the formula $P : A \wedge \neg A$, the solver should return that there exists no such valuation to satisfy P. Since propositional logics is decidable (it is always possible to find whether or not a formula is valid simply by building its truth

table) the solver should always succeed, but the computation time or the required storage can be a limitation depending on the heuristic used to solve the problem and the form of the formula. Another example of theorem provers are Satisfiability Modulo Theories (SMT) solvers. These solvers can be seen as the natural extension of SAT solvers with additional data types and associated proof techniques to extend the modelling and proving power of propositional logic. SMT solvers work on formulas where the variables are replaced by statements coming from another theory, such as natural numbers, lists, real numbers, ... These statements contain variables coming from these theories and the solver attempts at giving a valuation to these variables. Considering the formula $P : m \leq n \wedge n \leq 3 \wedge 2 < m$ the solver should return the valuation $m = 3$ and $n = 3$. This is important to note that such formulas can only be determined satisfiable when the underlying SAT problem (here $A \wedge B \wedge C$) is itself satisfiable, but the reciprocal is clearly not true. There exist many SMT solvers, that are based on different heuristics and handle different mathematical theories. The most known among them are CVC4 [21], Z3 [53], Alt-Ergo [47], veriT [29] and Vampire [136].

While SMT solvers represent the highest state-of-the-art level of automation that has been achieved regarding proof generation, there exist even higher level tools in formal methods, the proof assistants.

2.1.5 Proof assistants

Considering the programs that have to be verified through formal methods can not all be reduced to SMT problems – even if it can be the case for many useful real-life systems – and also that the state-of-the-art knowledge in automated provers seems to limit their extent to SMT problems, any tool that aims at verifying such programs has to partly rely on user interaction. This is the case for proof assistants, that assist the developer in its proof effort through two very different – but complementary – aspects.

The first one is the help in building the models and the associated proofs. This step is highly important because it consists in helping the programmer building proofs that are usually too complicated to be solely apprehended by the human mind in their entirety. Usually, the help comes at least from the ability to automatically decompose (and recompose) proofs in different sub-parts (often called sub-goals) that are accessible to the programmers understanding. This step is assisted by the tool but cannot be completely automatized because of the inherent undecidability in the logics these tools handle.

The second step is the automated verification of the provided proof. This step is done automatically by the tools as it relies on a decidable procedure. It takes a fully formed proof (either built by the programmer or taken from an external prover) and proceeds to ensure its correctness. In theorem provers based on type theories for instance, this step is directly equivalent to type checking the proof in regard of a given expected type. This notion of compatibility between the type of the term and the type corresponding to the expected property is not as straightforward

as it sounds. Indeed, there may be some further unification – or a notion of subtyping – required to assess this compatibility rather than just comparing the type for equality. In AGDA for instance, there exists a clear distinction between the entity that assesses the correctness of the proof – the type checker – and the entity that tries to automatically provide such terms – AGSY, a term generator – as explained and detailed in Section 2.3.3.d. In COQ as well, when some tactics fail, this can either be because no term was provided by the tactic – either internally or using external solvers – or because the provided term turned out to be incorrect – incompatible – regarding the specification when proof-checked by COQ.

In other words, proof assistants are programs that aim to guide the human user through his proof effort while verifying that the resulting proof indeed corresponds to the formula it aims at proving. These tools provide various ways of handling a computer assisted proof development. They usually embed, or are interfaced with, automated theorem provers (for example first order decision procedures, or SAT / SMT solvers) by allowing the user to invoke these provers when he sees it fit – when the current sub-problem can be expressed as an automatically solvable problem. While they usually allow the proofs to be divided in several fragments, each of these fragments has to be manually solved by the users, either by explicitly giving the right proof term, by recursively fragmenting the new proof goal or by invoking an external or internal prover. This means that each step of the proof effort is manually done by the user while the tool checks and composes the user inputs. There exist many different kinds of proof assistants, such as COQ, a dependently typed language using tactics to build proof terms, AGDA, the language which was used during this thesis, Isabelle, a meta-language to model logics [131], Isabelle/HOL, the implantation of HOL in Isabelle [123], PVS [10], Lean [54], LCF [139], Matita [11], ACL2 [35], KeYmaera X [70] or Automath [19]. More on COQ and AGDA will be found in Section 2.3.5. Proof assistants usually allow the program and the proof to be built in the same development environment, thus keeping the link between these two worlds at all time. With the emergence of type theories and the Curry-Howard correspondence, this link has further been re-enforced in a way that they actually appear as two faces of the same coin. This will be detailed in Section 2.2. Being able to develop both the program and the proof in the same language also allows the development of correct-by-construction systems, as done in B [2] and Event-B [3] using vertical separation, as detailed earlier. However, this is only a possibility, not an obligation which means that these types of software are a lot more flexible than such approaches, with the drawback that they are also a lot less systematic and mechanical. Properties can be proven during the development effort, or after it, depending on what suits the proof assistant user best. This makes these tools a very powerful and very flexible asset in the development of certified programs, to the point that they are, in my opinion – an opinion shared, for instance, by Xavier Leroy [97] – the most advanced tools we have in that regard – so advanced that they were able to provide proofs for very important postulates that were yet to be solved, both in mathematics (4-colors theorem [9], Kepler theorem [77], Feit-Thompson [75], ...) and in computer science (Compcert [96], Sel4 [87], ...).

2.1.6 Combining verification methods together

This section, by presenting a partial overview of the available verification methods, depicts a complex, dense and ever changing set of tools and methods which all target a specific field or rely on a specific logic. This makes them powerful for a given category of problems, and less so for the others. Attempts at combining the upsides of some of these methods have been made, and mostly follow two different directions. The first direction consists in a framework which, given a program annotated with preconditions, postconditions, variants and invariants, produces a set of proof obligations which are then translated into different possible formats which can be processed by different verification tools. This is for instance the approach advocated by tools like Boogie [20] and Why3 [66]. Why3, for instance, generates proof obligations which can be given directly to SMT solvers such as Alt-Ergo, Z3 or CVC4, or manually processed with Coq. Why3 then ensures that all the proof obligations have been handled by either of these means, in which case it considers the program correct. Why3 can also be used as pivot model between more complex languages like ADA or C and low level proof tools. This approach is very interesting but does not allow different logics to actually talk and understand one another. Indeed, Why3 does not retrieve nor verifies the proofs which have been provided by the provers that were used, especially because some of them do not provide any proof term. Why3 can be seen as an additional layer built on top of different provers, to which it delegates the proof effort after having generated the proof obligations. This leads to the second approach, which is more conceptually pleasing but also more challenging, which consists in embedding several logics in a single framework, which would be able to verify proofs coming from different tools and provers, and expressed in different formalisms. Such a framework is DEDUKTI [13] which proposes to encode different proof theories using a single formalism: $\lambda\Pi$calculus modulo. This is quite challenging because it requires to allow theories which are based on very different assumptions – such as constructive and non-constructive logics – to coexist. This approach has a major upside which is the existence of a proof term in all cases, which is then verified by DEDUKTI itself. The provers or theories are no longer black boxes which should blindly be trusted: they are merely providers of goods which are then verified.

2.2 Dependent types: types or properties ?

Explained in Section 4.1, the work depicted in this thesis relies on the use of dependent types to express definitions and relations between them. Indeed, AGDA is a dependently typed language and it will be described thoroughly in Section 2.3. Dependent types are the core concept of several proof assistants and / or programming languages such as COQ, IDRIS [31] or AGDA. Some of them, like COQ, provide convenient high level ways, – tactics – to build proofs while hiding numerous complex underlying proof terms. Others, like AGDA, have chosen to keep these terms visible while also providing help to the programmer building them, also AGDA is

currently experimenting the addition of tactics using reflection [149]. In both cases, the underlying mathematical theory is the same and needs to be mentioned in order to understand several parts of this thesis. This is the goal of this section.

2.2.1 Logics and classical logics

Classical logic is the most well-known logic that has ever been studied both in mathematics and computer science. While its philosophical creation goes all the way back to the antiquity, its mathematical formalization and improvements have been performed throughout the 19th and 20th century. While classical logic is often referred to as a single logic, it is more a family of logics sharing the same heart. Any logic can be defined as a set of inference rules stating how the proof of some formula can be built from the proof of others. An inference rule is composed of premises and one or possibly several conclusions, that the logic allows us to build when the premises hold. From these inference rules, others can be deduced which are called lemmas or theorems. A set of inference rules is said coherent (or consistent, or sound) when no successive applications of of inference deductions can result in both an affirmation and its negation. Logic differs from one another by their set of inference rules. Axioms are usually considered as inference rules without any premise, although since all inference rules are by definition postulated, they can all be considered as axioms. Figure 2.1a shows an inference rule with a premise while Figure 2.1b shows an inference rule without a premise.

$$\frac{A, B}{A \land B}\ \texttt{introduction} \qquad\qquad \frac{}{A \lor \neg A}\ \texttt{excluded-middle}$$

(a) An inference rule

(b) An axiom

Figure 2.1: Examples of inference rules

These examples of inference rules have not been chosen randomly. The first one, the introduction of the conjunction is a typical inference rule of most logics. It states that if both A and B are proven true, then $A \land B$ as a whole can be proven true as well. This looks trivial but it really isn't since this inference rule also introduces the notion of conjunction by defining the operator $\land$ and by giving it a specific meaning, that is the consensual conjunction of facts or elements. The second one is also very important because it captures the essence of classical logics. Basically, any logic that is classical contains this axiom of excluded middle, or any equivalent axiom in its foundations. This is a very profound axiom because it captures a certain way of perceiving reality. Indeed, in classical logics, any fact is supposedly either true or false regardless of whether anybody has proven one or the other. This seems natural because it matches our common notion of logic: either something happened or it did not, either something is there or it is not, and so on. This corresponds to the well-known (and sometimes controversial) axiom of choice in mathematics.

However, this is not so obvious when considering for instance infinite systems or the notion of witness or computation, as advocated in intuitionistic logic.

2.2.2 Intuitionistic logic

While being very intuitive, classical logic implies a certain version of truth. They imply that something is true or not by essence, as if a "god" or an "oracle" would be, for any given term, able to say whether this term is true or false. Advocating this notion of truth seems, as already stated, natural, but there exist some weird – and extraordinary – results that somewhat shattered its obvious nature. These results, mostly due to a famous logician called Kurt Gödel, refers to the foundation and nature of logic itself. One of the results is Gödel's theorem of incompleteness [74]. This result states that any computable axiomatic system that is able to express the arithmetic of natural numbers is either incomplete or inconsistent, hence if it is consistent then it cannot be complete. This means that such system necessarily contains true theorems that cannot be proven within its own system. This directly questions the notion of truth. What is the truth if it cannot be proven?

Such results that directly question the nature of what we call "truth", coupled with the paradoxes of the set theory (such a the Russel paradox [137]), have slowly given birth to logics that embeds a different notion of truth: something is true if there exists a proof of it. Weirdly enough, this sounds as natural as the definition of truth within the classical logics, even though it is a completely different one. This can even lead to philosophical questions about the notion of truth, which I personally find fascinating. The logic that embeds this definition of truth – and thus rejects the excluded middle – are called intuitionistic or constructive logics, because they solely rely on building proofs to decide the truth of a given term instead of just assuming it has to either be true or false. A direct consequence of this definition is that intuitionistic logic are by nature weaker than classical logic but they aim at mirroring as close as possible the notion of constructivism: is considered true only what can be built, or, in the case of programs, what can be computed.

2.2.3 Curry-Howard isomorphism

What is called the Curry-Howard isomorphism – but is mostly a correspondence – is a direct syntactic link that has been noticed and formulated in the middle of the 20th century by Curry [50] and Howard [83] between logic formula and functional types. The original functional language on which it has been noticed is the typed lambda calculus [18] whose function types look like $A \rightarrow B$ when transforming an element of type A into an element of type B. As noticed by both Curry and Howard, $A \rightarrow B$ can also be interpreted as the logical implication between A and B stating that if A holds, then B holds. This observation is a fundamental result in connecting the logical world to the programming world. While logical implication is the most fundamental and natural correspondence between these two worlds, there exist many more of them depending on the kind of logics one considers. How-

ever, all of these logics are intuitionistic and this is quite easily understandable because computer programs and functions build results through computation. This correspondence leads to the fundamental following result: in a functional language whose type system is equivalent – through the Curry-Howard isomorphism – to a given intuitionistic logic, every term that can be built in that language has a type that corresponds to a true formula in that logic, and vice-versa every program is a constructive proof of its type. Similarly, the typing rules of this language are equivalent to the inference rules of the corresponding logics. This result has given birth to the most advanced tools we have in terms of theorem proving: COQ, AGDA, and their siblings. Figure 2.2 summarizes the basic corresponding elements between a typed functional language and an intuitionistic logic.

Logic	**Symbol**	**Type**
Implication	$\Rightarrow$	Function type
Conjunction	$\wedge$	Product type
Disjunction	$\vee$	Sum type
Universal quantification	$\forall$	Π type
Existential quantification	$\exists$	Σ type
True	$\top$	Unit type
False	$\bot$	Empty type

Figure 2.2: The type/formula correspondence

2.2.4 Dependent types

The Curry-Howard correspondence tells us that finding an inhabitant of a given type – proving that this type is not empty – is equivalent to proving the property represented by this type, – note that the nature (the value) of the inhabitant is irrelevant. This correspondence is as powerful as the logic that is emulated by the target language type system. The more expressive the type system, the more powerful the logic it represents becomes. Since logics have been widely studied for a long period of time – and classified throughout the process – logicians and computer scientists started developing languages with types that would be more and more expressive in order to match these logics. Propositional logics would be for instance emulated by simply typed λ-calculus [36]. Adding polymorphism to this language would result in a type system like the one of CAML or SYSTEM F while adding dependent types and higher order polymorphism would result in type systems like the one of AGDA, which is able to emulate higher order logics. Before explaining what I mean by high level polymorphism, let me have a quick look at levels of universe. As mentioned before, usual set theories have some paradoxes that type theories avoid. The inconsistencies are usually created by allowing sets to contain other sets or even themselves, which is forbidden in type theories by the notion of levels of universe. Each type is given a level of universe. The types that describe values are of level

0 while the types that describe and classify types are of level 1, and so on. This means that any element in such language is associated to a level of universe usually called `a`. A type of level `a` can only describe elements of level `(a - 1)` at most, which means it cannot describe – the type equivalent of "contains" for set theories – itself, which solves the paradoxes. As a direct consequence, there exists no such thing in type theories as the type of all types because there is an infinite chain of levels of universe, and this one would be of infinite level, which is not possible. To get back to higher level polymorphism, it means that types of such languages can be parametrized by – polymorphic over – types of any of these levels. In AGDA, for reasons unknown to me, a type is called a `Set`. This is very odd to me because types are typically not sets. Regardless of this personal opinion, this means that a given type `A` of a given level of universe `a` is called `Set a` while `Set 0` is shortened `Set`. In a language like AGDA, types can literally be parametrized by anything, hence giving almost unlimited expressiveness to the logics they represent. Oddly enough, polymorphic types are older than dependent types, even though dependent types emulate first order logic while polymorphic types are the beginning of higher level logics. This is why languages like AGDA, even though they are fully polymorphic, are called dependently typed languages. This is also easily explained by the difficulty of type-checking dependently typed languages.

2.3 AGDA: a programming language and much more

As explained in the previous section, dependent types are a powerful asset in programming and proving properties about programs. They have been theorized in the past four decades and have been present in formal methods ever since. Some languages have been fully developed around them, such as COQ [24], AGDA [125], IDRIS, EPIGRAM [112] or PIE [64]. While these languages share the same heart, they have different goals and different ways of building programs and proofs. We chose AGDA for the development in this thesis and the current chapter explains why this is relevant. It also provides a detailed description of both the language, through a tutorial, and its development environment.

2.3.1 Presentation of AGDA

AGDA is a dependently typed programming language developed at Chalmers university by Ulf Norell during its PhD thesis [124]. It is a complete rewrite of a language created by Catarina and Thierry Coquand also named AGDA while the new version of the language was originally named AGDA 2. Since AGDA is based on intuitionistic type theory (more precisely the Unified Theory of Dependent Types by Zhaohui Luo [103]) it is comparable to COQ, IDRIS and other dependently typed languages because they all share the same core. AGDA is a fully functional language and has originally been designed to write programs with the help of dependent types rather than to be a proof assistant by nature. However, as the standard library grew and the language evolved, it became clear that AGDA could indeed be used as a proof

assistant in an elegant manner due to its powerful development environment which provides a clever way of inhabiting types through an automation that is invisible in the resulting code but exists nonetheless, as shown in Section 2.3.5.b.

2.3.2 The specificities of the AGDA language

The AGDA language provides several features that makes programming and proving in AGDA intuitive and convenient. More importantly, these features can make the programs and proofs look like mathematical properties which is really convenient for their understanding, even though the proofs themselves are lambda terms and do not need to be understood after the type checker has validated them. This means that the properties are usually understandable for non AGDA users while the proof usually remains complicated to understand even for AGDA users, as explained in Section P.4.2. The features described in this part have a cosmetic impact on AGDA's programs while also easing the development of such programs and proofs. Here is a list of these features, given in an arbitrary order. Note that it only represents the subset of features I think are the most interesting and aesthetically impactful, and is not exhaustive in that regard. More features will be distilled throughout this document when their need arises.

2.3.2.a Unicode characters

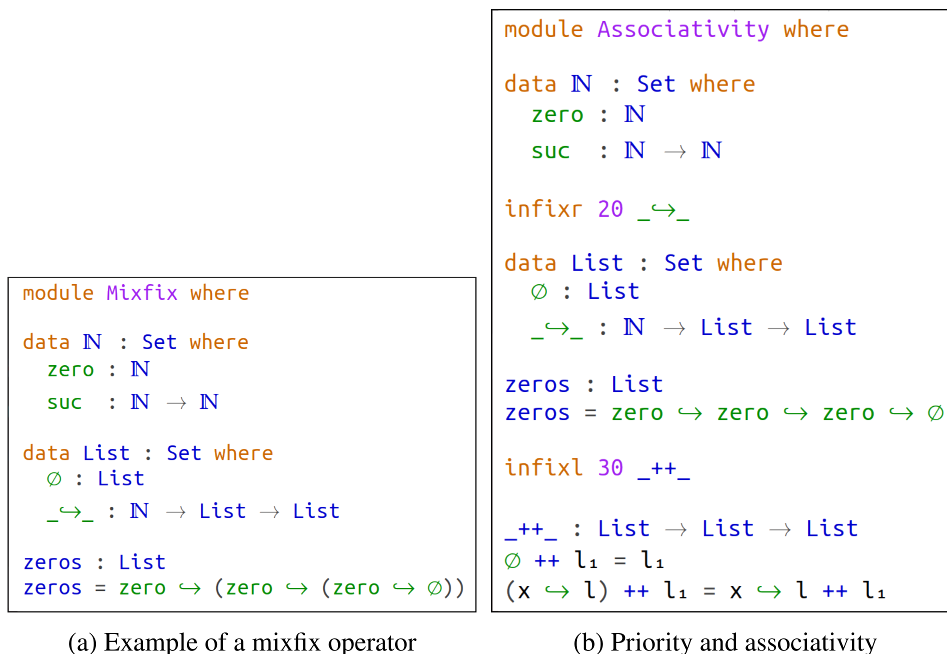
```
module Unicode where

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

data List : Set where
  ∅ : List
  ↪ : ℕ → List → List

zeros : List
zeros = ↪ zero (↪ zero (↪ zero ∅))
```

Figure 2.3: Example of unicode characters

AGDA supports unicode characters and the development environment (that will be described later on) provides a convenient (LaTeX like) way of writing them. This feature is illustrated on Figure 2.3. In this example, we define natural numbers with the symbol ℕ, the empty list with the symbol ∅ and the cons operator with the symbol ↪ [2]. The list we define at the end, named `zeros`, contains three zeros but is rather unreadable since the operator is placed before its operands, and since this definition requires a lot of parentheses. To overcome this limitation, AGDA provides mixfix operators.

[2]These symbols are purposely unorthodox in this example

2.3.2.b Mixfix operators, associativity and priority



(a) Example of a mixfix operator

(b) Priority and associativity

Figure 2.4: Features around operators

Mixfix operators are accessible while writing AGDA programs. They consist in n-ary operators, the names of which include underscores. These underscores specify the location where the n operands shall be placed. An example of the feature is depicted in Figure 2.4a to ease the reading of the list defined in Figure 2.3 by the use of an infix operator (a mixfix operator with one operand on each side. There can be any number of underscores in an identifier, and they can be placed anywhere inside its name, in which case it will be considered by AGDA as a mixfix operator directly. While the number of underscores supposedly matches the number of operands declared for the mixfix operator, it is not a requirement.

When there are more operands than underscores, the remaining ones shall be placed at the end of the identifier. This can be justified by the natural currying that exists in functions defined in most functional languages, AGDA included, in which case the first operands can be seen as part of the definition of a function and the last one as the operands for this function – note that this distinction is purely a view of the mind. When there are less operands than underscores, the operator identifier with underscores included is to be used as if it did not contain any. To my knowledge, there is no reason that would justify such use of underscores in operator identifiers.

In addition, AGDA provides a way to specify priorities between operators as well as an associativity (left or right) for these operators. The priority takes the form of an arbitrary number, the value of which is only relevant when compared to

other operators' priorities. As for the associativity, it allows the writing of even less parentheses by stating how the parser should handle expressions with a succession of the same operator. Parsing mixfix operators is a challenge, and this challenge has been tackled by Danielsson and Norell while developing AGDA, the details on this can be found in [51].

In Figure 2.4b, both features are illustrated. We start by specifying a right associativity for the `_↪_` operator, which allows us to remove parentheses around the list we define, called `zeros`. We give this operator the priority 20, before defining a new operator, the concatenation of two lists, `_++_`, and we give it a higher relative priority, 30. This allows us to write the last line of this AGDA program without any parentheses since the parser knows how to interpret this line.

```
module IfThenElse where

data Boolean : Set where
  ⊤ : Boolean
  ⊥ : Boolean

if_then_else_ : ∀ {a} {A : Set a} → Boolean → A → A → A
if ⊤ then a₁ else _  = a₁
if ⊥ then _  else a₂ = a₂
```

Figure 2.5: A mixfix operator with three operands

As a side example of mixfix operators, we provide an AGDA definition for the "if then else" construct in order to emphasize that, while mixfix operators can be used to model infix operators, they are not limited to them. This example is depicted on Figure 2.5 and is composed of the definition of the booleans, with two constructors, ⊤ and ⊥ as well as the definition for the "if then else" construct. This definition has five operands, two of them – a and A – being implicit, as depicted by the use of braces. The first implicit argument, a, is the level of universe, the second implicit argument, A is a type of that level and the three other arguments are the boolean condition, of type `Boolean`, and the "then" and "else" expression, of type `A`. The underscores are placed in the definition where the three non-implicit parameters shall be dispatched when defining or calling the function. This is used in the two lines following the prototype of the function where a disjunction of cases is made on the boolean value of the condition to return either the "then" or "else" expression. The implicit arguments will be inferred by the type checker when the function is called, while the type of the first argument is inferred by the type checker during the definition of the function, which is allowed by the use of the keyword ∀ in the type signature of the function. Implicit arguments are used in various ways in AGDA which will be discussed and / or mentioned several times throughout this document.

Another example of the use of mixfix operators coupled with priorities is depicted in Figure 2.6. In this example, some very basic operators have been defined and, coupled with the right relative priorities, they allow us to define lists conveniently.

As a side note on mixfix operators in AGDA, they offer the possibility of instan-

```
module ListSugar where

⟦_ : ∀ {a} {A : Set a} → List A → List A
⟦ l = l

_⟧ : ∀ {a} {A : Set a} → A → List A
a ⟧ = a ∷ []

_,_ : ∀ {a} {A : Set a} → A → List A → List A
a , l = a ∷ l

infixr 20 _,_
infix  15 ⟦_
infix  25 _⟧

example₁ : List ℕ
example₁ = ⟦ 2 , 3 , 4 ⟧

example₂ : List String
example₂ = ⟦ "hello" , "world" , "!" ⟧
```

Figure 2.6: A convenient way for defining lists

tiating one of the parameters regardless of their position while leaving the others as arguments. This is a similar behaviour as the currying behaviour with the addition that the instantiated parameters does not have to be the first. This adds to the challenge regarding syntactic analysis of such operators.

```
module Currying where

+3₀ : ℕ → ℕ -- A simple application of _+_
+3₀ x = x + 3

+3₁ : ℕ → ℕ -- Parameters after the whole name
+3₁ x = _+_ x 3

+3₂ : ℕ → ℕ -- Same with currying
+3₂ x = (_+_ x) 3

+3₃ : ℕ → ℕ -- A lambda function
+3₃ = λ x → x + 3

+3₄ : ℕ → ℕ -- Short syntax for the lambda function
+3₄ = \x → x + 3

+3₅ : ℕ → ℕ -- Instantiation of the second parameter
+3₅ = _+ 3
```

Figure 2.7: Currying a mixfix operator

An example of such feature is depicted in Figure 2.7. This example presents the different ways provided by AGDA to define a function. This definition can have various forms all detailed on the figure through AGDA comments using `--`.

2.3.2.c Identifiers

AGDA allows identifiers of the language to be anything as long as they don't include spaces, since they are the separators of the language. This is very convenient because it allows us to give relevant names to proofs by calling it according to their semantics. A example is given in Figure 2.8 where we prove the commutativity of the addition on natural numbers (the definition of which is the usual inductive one reminded in Section 2.3.4.b) through the use of two lemmas, the name of which is sufficient alone to understand what they prove. Since we haven't yet gotten into the semantics of the AGDA language, this is particularly convenient. As much as possible, this approach of using relevant names to characterize properties has been used throughout this work. It is, in my opinion, mandatory because the type signatures of such proofs can be filled with a lot of implicit arguments, in which case a relevant name helps to grasp the semantics of the property in question. This is an approach also used in the standard library.

```
module Commut where

n≡n+0 : ∀ {b} → b ≡ b + 0
n≡n+0 {zero } = refl
n≡n+0 {suc _} = cong suc n≡n+0

s[b+n]≡b+s[n] : ∀ {b n} → suc (b + n) ≡ b + suc n
s[b+n]≡b+s[n] {zero} = refl
s[b+n]≡b+s[n] {suc n} = cong suc s[b+n]≡b+s[n]

comm : ∀ {a b} → a + b ≡ b + a
comm {zero } = n≡n+0
comm {suc a} = trans (cong suc (comm {a})) s[b+n]≡b+s[n]
```

Figure 2.8: Identifiers with relevant names

2.3.3 The related tools around AGDA

AGDA is a very complete and complex language, both because of the features it provides, that were just mentioned, and also because of its inherent proof assistant nature which needs to be fuelled with lambda terms provided directly by the user. Only the most experienced users of such languages are able to write down lambda terms directly, without external help, and these terms have to be very simple which is often not the case. More often than not, these terms are composite and it is by nature very complicated to provide them directly. For these reasons, it is unreasonable to expect anyone to program in AGDA in a simple text editor. While this is theoretically possible, there exists a much more potent and powerful tool that allows us to program in AGDA even the more complex programs and their associated proofs. This tool takes the form of a mode [48] for the EMACS editor, the usage of which is quite mandatory when programming in AGDA. On a personal note, using this mode was not a problem for me because I have been used to EMACS for quite

a while. However, I feel that the mandatory use of EMACS is somewhat a drawback for programming in AGDA for inexperienced EMACS users since there are no alternative options to my knowledge, and there probably never will be considering how advanced this tool is. As a comparison, there exists another EMACS mode, called Proof General [12], which is a similar tool for developing with other proof assistants such as COQ.

2.3.3.a The framework: an EMACS mode

The AGDA EMACS mode is the entry door to any relevant interaction with AGDA while developing a program – or a proof[3]. Such development requires at least two EMACS buffers. The first one contains the code that is being programmed and the second one displays informations given by AGDA when requested. Figure 2.9 illustrates this layout by showing an overview of the EMACS mode while developing a simple program.

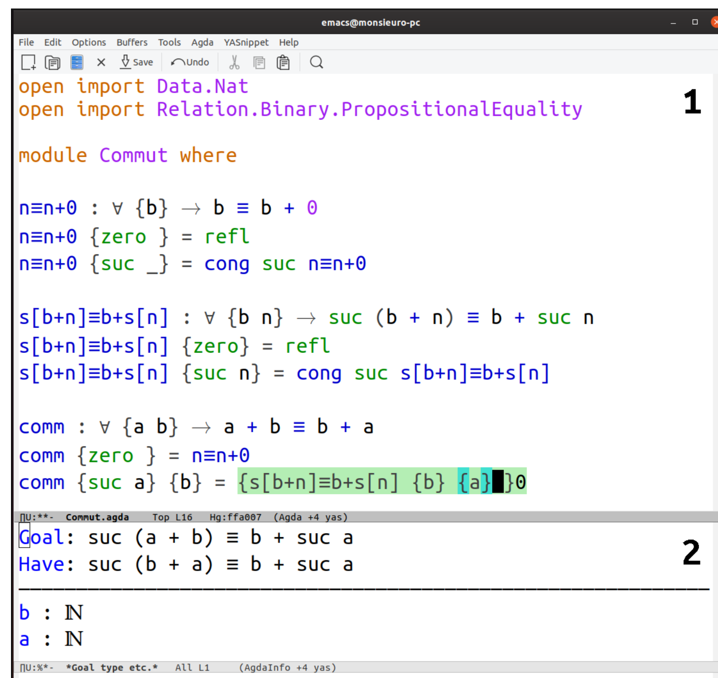


Figure 2.9: The AGDA EMACS mode

Buffer number 1 contains the AGDA code that is being written, with a green area called a hole which is an interaction point with AGDA that needs to be filled with a value of a specific type – with a proof of a specific property. After entering the command ctrl + c , ctrl + . in said hole, AGDA uses buffer number 2 to display the following pieces of information: the type of the hole as well as the type of the

[3]from now on, I will not make the distinction between programs and proofs any more, because the Curry-Howard correspondence tells us that this distinction is irrelevant.

current term that it contains. This allows us to assess quickly what is at our disposal and what is the difference between what is currently provided and what has to be provided in order to achieve the current goal. This view is the main one advocated when programming in AGDA. When several holes are present in the input buffer, one can navigate easily between them and ask AGDA the same information for any of the holes. The holes can also be refined, validated and the user can even ask AGDA to try and solve them itself. In addition, after each type-checking, AGDA uses buffer number 2 to display the types of all the holes in the document. More on these features will follow.

2.3.3.b A type checker

Thanks to the Curry-Howard isomorphism, we know that providing an element of a given type corresponds to proving the property associated to this type. This means that a core aspect of languages that aim to use this correspondence is the type checker. Indeed, the type-checker of such language is basically a proof checker because it assesses the correctness of the given type towards the expected type, – the correctness of the proof towards its theorem. AGDA comes with a type checker that can be interacted with through the EMACS mode, similarly to the interactions between COQ and its development environments, CoqIde or Proof General. Almost every interaction uses the type checker in one way or another because it is such an important piece of the proof assistant puzzle. For instance, Figure 2.9 shows in its second buffer an output that was computed by the type checker. Type checking the buffer and then the given input in the hole allows this interaction. Type-checking the current buffer is asked manually by the user – using ctrl + c, ctrl + l – and is required at most steps of any AGDA development. Type-checking has several positive side-effects that the user can witness and/or request:

Colourization Since type-checking depends on lexical and syntactical analysis, the buffer is colourized through the type-checking process when these steps are successful. Otherwise, errors are displayed.

Hole creation The type-checker transforms any "?" that were present in the buffer – as long as they are placed in a relevant location – into holes. These holes are considered type-correct inside an AGDA file and are sub-goals to be filled by the user. They also happen to be specific points of interaction with the type-checker. However, one cannot import a file that has holes in it because it is incomplete and thus cannot be reused until its completion is achieved.

Case-splitting AGDA functions are written on several lines, each line corresponding to a specific structural case of a given input. This means that case-splitting on an input is an essential aspect of every AGDA function. Thankfully, AGDA embeds a feature that, relying on the type-checker, allows us to automatically case-split on a given input. The type checker searches for the type of the given quantity and

when this type is a data type with several constructors, it expands the definition of the function with one line per possible constructor. Since these constructors can be dependent and can depend on values whose structure can be known at the time of the expansion, AGDA will in the process ignore the constructors that are not valid considering the structural aspect of these values. In COQ, this process is called inversion and corresponds to the creation of several sub-goals that correspond to the possible constructors for the given value.

Inference of implicit arguments On Figure 2.9, it is very interesting to take a closer look at what information is given by AGDA. The goal is completely formulated by AGDA while the currently provided term has a type that is parametrized by unknown quantities indexed by some natural numbers. These quantities are yet to be inferred by AGDA. Both these types are strictly different, one is completely determined while the other is parametrized. However, when asking AGDA to replace the hole by the given quantity – by pressing ctrl + c , ctrl + Space – AGDA accepts the term as type correct. This is particularly interesting because it emphasizes another kind of unification done by the type checker. AGDA, while validating the hole, checked whether or not the type of the goal could be instantiated by the type of the given quantity. To perform such an instantiation, AGDA had to figure out which value to assign to the implicit parameters in order to match the goal type. This process is undecidable, which means that this unification may not be found, in which case the user has to manually give the value of some of the implicit parameters in order for AGDA to accept the term, if such values exist. However, this process is extremely powerful and important while using implicit parameters. Experienced users know how tedious such a language without implicit parameters would be. More often than not, when such a validation is refused by AGDA, the problem comes from the term itself and not from a lack of information given to the type checker. On a personal note, this helped me several times to understand that I was actually trying to prove something that did not hold, usually because of a simple mistake in the function signature.

Hole refinement A hole can be completed with a partial expression (either an expression with additional "?" or a function or constructor) in which case the user can ask AGDA to try and refine[4] the hole toward the given expression. In the case of a function or constructor, this asks AGDA the following question: is there any possible argument that would make such a function call on these arguments type-correct regarding this hole ? If yes, AGDA will validate the function and create new holes for the user to input the arguments. In the case of an incomplete expression, the question is mostly the same: is there any possible quantities that, when placed at the unfilled spots, would make the expression type-correct regarding this hole ? If yes, AGDA will create new holes where the unfilled spots were specified and will validate the rest of the expression. This feature is extremely convenient and used

[4]note that this notion of refinement is unrelated to the one described in Section 5.2

very often while building proofs because it allows us to build them step by step, sub-goal by sub-goal, without having to know the structure of the proof beforehand.

Use of unknown identifiers AGDA allows the user to create new quantities on-the-fly throughout the proof effort. When reaching a certain point in a proof, if the user feels like using a lemma or a function that was not defined beforehand, he is allowed to use an undefined identifier somewhere in the current hole and ask AGDA the following question: would it be possible for such an identifier to exist in such an expression and, if so, what should its type be? If AGDA succeeds in finding such a type, the signature of the new identifier is copied and can be pasted elsewhere in the buffer in order to write its definition. This is also extremely convenient to build proofs step by step as well as ease their subsequent understanding.

2.3.3.c A termination checker

The type checker is coupled with a termination checker. When explaining that the type checker was in fact a proof checker, I deliberately omitted that it needs to be associated with a termination checker in order to be categorized as such. A non terminating program would prove false and allow us to prove basically anything. To accept a definition, AGDA has to ensure that this definition terminates, – in addition to checking that it is indeed type-correct. The termination checker is very basic in the sense that it only accepts structural recursion as termination proofs, – in addition to non-recursive definitions. However, there is nothing more it can do because termination checking is undecidable for any Turing-complete language as proved by Turing himself in 1936 [147]. While a lot of recursive definitions can be formulated with structural recursion, there exist several ways to transform non-structural recursion to structural recursion, hence allowing the termination checker to validate them. The state-of-the-art methods to prove the termination of a non-structural recursive function `f` all revolve around the same idea: define a function `f'` which is structurally recursive and from which the result of `f` can be deduced. There are mostly four ways of defining `f'`:

- Representing the data on which `f` works in a structural manner, and make `f'` structurally decrease on this new data type.
- Making `f'` work on an equivalent data which is typed with sized type that structurally decreases as the function unfolds, a technique which was concurrently discovered in [73] and [84].
- Adding an additional parameter to `f'` that represents the accessibility toward a well-founded relation of the parameter of `f` [59].
- Using semantics labelling, which proposes to label the components of `f` in `f'` with some additional semantics elements as introduced in [156].

These methods have been compared with one another, for instance in [26] where the authors try to build a bridge between sized types and semantics labelling. On another note, there also exist some works whose purpose is to automatically generate termination proof, as for instance the CoLoR library for COQ [25].

2.3.3.d A term generator

The last feature of the AGDA framework discussed in this section is AGSY [102], the AGDA automated term generator. AGSY is a general purpose search algorithm for dependently typed languages that is directly embedded in AGDA. When asked – using ctrl + c , ctrl + a – AGDA will try to fill automatically the current hole by invoking AGSY which will try and build a term of the goal type using the elements in the context. By default, this engine will not use other definitions in the module but it is possible to provide it with "hints" that consist of names of the definitions it should try and use in its search process. This happens to be surprisingly convenient because it is often known which quantities should be used to solve a given goal, for instance a specific lemma that was created for that purpose. When successful, invoking AGSY will replace the current hole with the expression it provided. AGSY is also able, when specified, to try and case split on the variables of the context in which case it will try to provide several terms for each individual case.

```
module Commut where

n≡n+0 : ∀ {b} → b ≡ b + 0
n≡n+0 {zero } = refl
n≡n+0 {suc _} = cong suc n≡n+0

s[b+n]≡b+s[n] : ∀ {b n} → suc (b + n) ≡ b + suc n
s[b+n]≡b+s[n] {zero} = refl
s[b+n]≡b+s[n] {suc n} = cong suc s[b+n]≡b+s[n]

comm : ∀ {a b} → a + b ≡ b + a
comm = {-c -m cong trans }0
```

(a) Invoking AGSY

```
module Commut where

n≡n+0 : ∀ {b} → b ≡ b + 0
n≡n+0 {zero } = refl
n≡n+0 {suc _} = cong suc n≡n+0

s[b+n]≡b+s[n] : ∀ {b n} → suc (b + n) ≡ b + suc n
s[b+n]≡b+s[n] {zero} = refl
s[b+n]≡b+s[n] {suc n} = cong suc s[b+n]≡b+s[n]

comm : ∀ {a b} → a + b ≡ b + a
comm {zero} = trans (cong (λ z → z) n≡n+0) refl
comm {suc n} = trans (cong suc comm) s[b+n]≡b+s[n]
```

(b) Result of this invocation

Figure 2.10: Limitations of AGSY

While this feature is indeed convenient, I have witnessed some weird cases where AGSY builds a correctly-typed term that somehow is rejected by further type-checking of the buffer due to a lack of information regarding implicit arguments. This means that, AGSY was able to unify some values together while subsequent type-checking by AGDA cannot, which is very surprising. This would indicate that AGSY uses a different heuristic than AGDA in terms of type constraints solving (at least regarding the instantiation of implicit arguments) which, in itself, is not that surprising considering that AGSY is naturally separated from AGDA– see Section 2.1.5. However, this raises several questions: is AGSY's heuristic more efficient than the type checker's one in terms of implicit arguments instantiation ? In this case, why doesn't the type checker use the same heuristic? Overall, are both heuris-

tics different but equally powerful? In that case, couldn't they be used together for better results?

Another downside of this automation is that AGSY seems to try and use as many of the provided hints in the terms it creates as possible, rather than as few as possible, which leads to overly complicated terms that can be reduced quite simply through further inspection. Both these limitations are depicted on Figure 2.10. Figure 2.10a shows the invocation of AGSY with the goal of proving the commutativity of the addition. The options are specified in the corresponding hole: "-c" allows AGSY to use case splitting while "-m" provides AGSY with the definitions of the module with the addition of `cong` and `trans` which are required to solve the goal as shown in the expected result provided in the comments at the end of the file. Figure 2.10b shows the result given by AGSY, with its two main limitations being visible. The first term provided by AGSY, `trans (cong (λ z → z) n≡n+0) refl` is actually reducible to `n≡n+0` but AGSY apparently wanted to use `trans` and `cong` because they were given as hints, even thought these are just hints. Had I called AGSY on the first goal after case splitting manually, it would have found the term `n≡n+0`. The other limitation is shown by the yellow highlight in the buffer, which is AGDA's way of telling the user that the type checker lacks information (instantiation of implicit parameters) to complete its task. Here the instantiation of the first parameter of `comm`, `a`, is mandatory for AGDA's type checker to succeed while AGSY was somehow able to conclude without having to explicitly give this information.

On a positive note, term generation is very powerful when used on concrete proofs (examples) rather than abstract ones. When using AGDA frequently, it becomes more and more obvious when this term generation will succeed or not because it usually succeeds when the user is able to input the term by himself, in which case this results in nothing more than a convenient time saver. Overall I would say this feature is easily the less advanced one of the tool and should definitely get some improvement. Seeing AGDA connected to more advanced external provers would be a great advancement for the tool and some work has already been done in that direction [68].

2.3.4 AGDA: a short tutorial

While we went through a set of interesting design features of AGDA, we haven't yet gotten into the semantics of the language. This section presents a short AGDA tutorial for the reader to better understand the AGDA code snippets that will be presented in this document. The goal of this tutorial is not to fully present AGDA, but rather to give the reader an understanding of the basics of the language and to be able to follow this document. Other more complete tutorials have been written, such as the original tutorial by Ulf Norell in [125] and another tutorial which can be found online in [107]. This tutorial is separated into different sections, each presenting a different aspect of the language. Many features that are present in multiple of these aspects will be detailed as they appear in this development. AGDA is a dependently

typed functional programming language, and as such, the definition of data types, functions and proofs is to be expected.

2.3.4.a Data types

We assume that we start from scratch in the writing of an AGDA module, and we would like to work on natural numbers and lists. These are the most intuitive inductive data types and we could import them from the standard library, but we define them ourselves for the purpose of this tutorial.

```
1  data ℕ : Set where          1
2    zero : ℕ                  2
3    suc : ℕ → ℕ               3
```

This is the classical definition of natural numbers from the Peano's axioms. `Set` is the type of types – level of universe 0, as explained in section 2.2.4 – so we state here that ℕ is a type for which we give two constructors : `zero`, that directly creates an element of ℕ and `suc` which transforms an element of ℕ into another element of ℕ. Such data type definitions are the basic bricks on which an AGDA program is built. We will see later in this tutorial that correctly defining such data types is mandatory when dealing with types that represent a given predicate, and thus depend on values. In the case of natural numbers, the type is not dependent and this definition is the same as in any non-dependent functional programming language. Now we can advance to the definition of a somewhat more advanced data type, in the sense that it is polymorphic.

```
4  data List {a} (A : Set a) : Set a where      4
5    [] : List A                                5
6    _::_ : A → List A → List A                 6
```

The data type of lists, as stated before, is different from the one of natural numbers because it is parametrized by the type of its elements, which is `A` of type `Set a`. It is also parametrized by `a`, which is the level of universe of the type `A`. Note that this additional parameter is surrounded by braces because we declare it as implicit. As mentioned in Section 2.3.3.d, implicit parameters take a very important place in an AGDA developement. When the developer of an AGDA program makes a parameter implicit, they ask AGDA to figure out the value whenever possible. There is no limitation as to which parameters can be made implicit, but there are also no guarantee that AGDA will figure out the right value for these parameters when needed. In the case of the levels of universe, they are usually made implicit because they are usually related to another parameter (a type) that will be given explicitly, thus allowing AGDA to infer the value of the level of universe in most cases.

Going back to the semantics of our data type, we see that an element of type (`List A`) can either be built from the `[]` or `_::_` constructors. The first one rep-

resents the empty List and the second one stands for the aggregation of an element and another list.

An associativity and a priority can both be provided for such an operator with the following AGDA instruction, as depicted in Section 2.3.2.b:

```
7   infixr 20 _::_                                                   7
```

Here we state that _::_ is right associative and has a priority – arbitrarily – set to 20. When used alongside other operators, their respective priorities and associativities will be used by AGDA to correctly parse the expression.

After defining a simple data type and a polymorphic data type, we can move on to the definition of dependent data types. Dependent data types can represent either simple data, or predicates. We start by defining a simple dependent data type to represent the lists of a given size, called vectors. Vectors represent the most simple dependent data type that is usually used to introduce new users to such types. They must not be mistaken with the vectors representing the arrays used for instance in imperative languages.

```
8    data Vec {a} (A : Set a) : ℕ → Set a where                      8
9      [] : Vec A zero                                                9
10     _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)                    10
```

This definition is particularly interesting because it features both parameters and indices. The parameters of this data type are the level of universe `a` and the type of the elements in the vector, `A`. This data type also has an indice of type ℕ, which is the size of the vector. Note that this indice is on the right of the semi colon in the signature of the data type. This is what separates parameters, that are fixed for the data type – they can be seen as inputs that cannot be changed afterwards, and indices, that are the set of values on which this data type can be inhabited through the use of its constructors. In this specific case, and since this data type represents data and not a logical predicate, all indices (all natural numbers) are possible – it is possible to create vectors of any length, hence the set of indices is ℕ itself, but this will not always be the case, to say the least.

Similarly to the lists, this data type features two constructors, one to build the empty vector, in which case the indice is 0, and one that aggregates an element to the vector, in which case the original indice is incremented by one, thus keeping track of the size of the vector in its type, as expected.

Note that this definition also features another occurence of the ∀ keyword, as mentioned in Section 2.3.2.b. This keyword is both a syntactic sugar, allowing AGDA types to look more like mathematical properties, but also allows us to omit the type of the following quantities, letting AGDA infer it. For instance here we don't give the type for `n`, and AGDA succesfully infers its type because it is later used in the quantity `Vec A n`, whose indice has been declared to be of type ℕ. In this case, both the value of `n` and its type are left for AGDA to infer.

Dependent data types can also be used to model predicates, in which case the set of indices will be limited to the ones that satisfy the property that the data type is supposed to model. Here is a simple example of such data type.

```
11  data Even : ℕ → Set where                         11
12    z-even : Even zero                               12
13    s-even : ∀ {n} → Even n → Even (suc (suc n))     13
```

In this example, we define a predicate over the natural numbers, that is satisfied if a number is even. Since not every natural number is even, and we want our data type to embrace the consensual semantics of being even, we cannot allow this data type to be inhabited by any indice. This means we have to provide constructors only for the indices that are indeed even. This is done by providing two constructors: the first one, `z-even` states that zero is even while the second one, `s-even`, states that if n is even, then $n+2$ is also even. These constructors are represented with inference rules on Figure 2.11.

$$\frac{}{Even(0)}\ \texttt{z-even}$$

(a) `z-even` constructor

$$\frac{n : \mathbb{N} \quad Even(n)}{Even(n+2)}\ \texttt{s-even}$$

(b) `s-even` constructor

Figure 2.11: Inference rules for the `Even` predicate

From this definition, we can prove that 6, for instance, is even.

```
14  six : ℕ                                            14
15  six = suc (suc (suc (suc (suc (suc zero)))))       15
16  -                                                  16
17  6-even : Even six                                  17
18  6-even = s-even (s-even (s-even z-even))           18
```

AGSY here automatically finds the inhabitants of these types (ℕ and Even six) and the proof it provides corresponds to the deduction tree depicted on Figure 2.12.

This is the first example of data types that embeds a semantic restriction through the set of indices on which they can be inhabited. A second one, the propositional equality, will be presented later on in this tutorial. It is interesting to note that there is no restriction, to my knowledge, as to what the type of indices can be. For instance, one could write (and possibly give a meaning to through constructors) such a data type that would be a predicate over predicates of level a:

```
19  data dummy {a b} : Set a → Set b where -  . . .   19
```

$$\cfrac{\cfrac{\cfrac{\cfrac{}{Even(0)}\ \texttt{z-even}}{Even(2)}\ \texttt{s-even}}{Even(4)}\ \texttt{s-even}}{Even(6)}\ \texttt{s-even}$$

Figure 2.12: deduction tree proving that 6 is even

2.3.4.b Functions

From the definitions of the natural numbers and the lists, we can write functions that compute or accept as parameters the elements of these data types. The first operation we propose to define is the addition between two natural numbers:

```
20  _+_ : ℕ → ℕ → ℕ                20
21  zero + b = b                   21
22  (suc a) + b = suc (a + b)      22
```

Every AGDA function is defined using one or more lines which are the different possible cases for the arguments we pattern match on. Each of these lines are in the form (`patterns = result`) where `patterns` are the current patterns for the inputs and `result` is the returned value corresponding to these patterns. In this function, we pattern match on the first argument which results in two different cases. In the first case, we return `b` since `a` is equal to `0`. In the second case, we recursively call the `_+_` function whose result is then encapsulated with the `suc` constructor to build the right natural number. As mentioned in Section 2.3.3.c, termination checker is not decidable. Hence, recursive functions must be written in such a way that the termination checking succeeds. This means that recursive calls must be made on structurally smaller arguments which is the case here since `a` is smaller than `suc a`. If AGDA fails at automatically synthesizing the termination proof, it means that a structurally smaller argument has not been provided and AGDA will reject the definition until the programmer fixes the issue. In the case of this definition, the recursion is naturally structural and AGDA accepts it as it is.

Since we have the natural numbers and the lists at our disposal, it is natural to provide a way of computing the size of a list:

```
23  size : ∀ {a} {A : Set a} → List A → ℕ      23
24  size [] = zero                              24
25  size (hd :: tl) = suc (size tl)             25
```

Note that this function uses two implicit arguments, `a` and `A`. The resolution of the latter by AGDA is shown in the following snippet:

```
26  myList : List ℕ                                        26
27  myList = zero :: []                                    27
28  -                                                      28
29  sizeImp : ℕ                                            29
30  sizeImp = size myList                                  30
31  -                                                      31
32  sizeExp : ℕ                                            32
33  sizeExp = size {A = ℕ} myList                          33
```

In the example above, we define a list and then we call the size function on it twice. The first call is `size myList` and AGDA here is able to infer the value of `A` since `myList` has been defined as a list of natural numbers. In the second call, `size {A = ℕ} myList`, we explicitly (for the purpose of this tutorial) gave AGDA the value of `A`. These two calls are equivalent. Please keep in mind that when we write `{A = ℕ}` we do not state that `A` is equal to ℕ, but we only assign in the function call the value of the `A` parameter to ℕ

The last function we would like to write is the concatenation of two lists, which we define as an infix operator.

```
34  _++_ : ∀ {a} {A : Set a} → List A → List A → List A   34
35  [] ++ l = l                                            35
36  (hd :: tl) ++ l = hd :: (tl ++ l)                      36
```

2.3.4.c Properties

Concatenation of lists From the definitions above, we would like to prove a simple property: the sum of the size of two lists is equal to the size of the list resulting from the concatenation of these two lists. To do that, we need the notion of equality which we define as an inductive family generated by the reflexive rule, a very fundamental approach taken from Martin-Löf:

```
37  data _≡_ {a} {A : Set a} (x : A) : A → Set where       37
38    refl : x ≡ x                                         38
```

Here we state that, for all `x` in all `A`, there is only one indice that inhabits the family of elements that are equal to `x`, and this is `x` itself. The constructor which can build an element of this type is `refl` that corresponds to the reflexivity of the propositional equality. Figure 2.13 presents the inference rule associated to this data type.

Now, when we pattern match on a proof that `x ≡ y`, not only will AGDA know there is only one possible constructor, `refl`, but it will also unify `x` and `y` to the same element knowing that the indice `y` has to be equal to the parameter `x`. For example, this is how we can prove the congruence of the propositional equality:

$$\frac{}{x \equiv x}\ \texttt{refl}$$

Figure 2.13: Inference rules for the `x ≡_` predicate

```
39 cong : ∀ {a b} {A : Set a} {B : Set b} {x y} (f : A → B) → x ≡ y → f x ≡ f y  39
40 cong f refl = refl                                                          40
```

When we pattern match on the proof that `a` equals `b`, AGDA unifies `a` to `b` and the goal becomes `f a ≡ f a` whose proof is then simply `refl` because both sides of the equality are the same element. This last property leads us to the proof we wanted to achieve regarding the size of the concatenation of two lists:

```
41 ≡size : ∀ {a} {A : Set a} {l l₁ : List A} → (size l + size l₁) ≡ size (l ++ l₁)  41
42 ≡size {l = []} = refl                                                          42
43 ≡size {l = hd :: tl} = cong suc (≡size {l = tl})                               43
```

This proof is done the same way any function is written, by case splitting on the first list parameter, `l` and providing a result of the right type for the different possible cases. In the case where `l` is empty, then AGDA reduces the output type to `size` $\texttt{l}_1$ `≡ size` $\texttt{l}_1$ by applying the definitional equalities of the functions used in this signature, whose proof is trivially `refl`. In the second case, the function is called recursively and the constructor `suc` is applied to both sides of the resulting equality using `cong` to provide a correctly typed term.

Concatenation of vectors We can apply the same process on vectors, in which case the proof elements are encoded directly in the concatenation function:

```
44 _++ᵥ_ : ∀ {n n₁ a} {A : Set a} → Vec A n → Vec A n₁ → Vec A (n + n₁)  44
45 [] ++ᵥ v₂ = v₂                                                       45
46 (x :: v₁) ++ᵥ v₂ = x :: (v₁ ++ᵥ v₂)                                   46
```

The distinction between lists and vector is very instructive. List is a very simple and flexible type, which requires to prove length properties explicitly, while vector embeds the length property in its signature. Both possibilities can be used depending on the context and will be discussed in Section 4.3.2.

2.3.4.d Decidability

The goal of this last part of our tutorial is to give AGDA rules to compute the proof whether two natural numbers are equal or not. For that, we need a data type that encapsulates these two possible cases. The first step in this direction is to define the false predicate:

```
47  data ⊥ : Set where                                                    47
```

We create a data type for which we do not give any constructor. It exists, but there is no way anyone can provide an element of this type (a proof that ⊥ is inhabited). We can then move on to the definition of the logical negation:

```
48  ¬ : ∀ {a} → Set a → Set a                                             48
49  ¬ A = A → ⊥                                                           49
```

This is simply a function that transforms a predicate into another one. We can now define the data type we wanted:

```
50  data Dec {a} (A : Set a) : Set a where                                50
51    yes : (p : A) → Dec A                                               51
52    no : (¬p : ¬ A) → Dec A                                             52
```

We can build an element of `Dec A` either from an element of `A` (a proof that `A` holds) or from an element of `¬ A` (a proof that `A` does not hold). This data type is fundamental because it binds the proof and the computing worlds. If, for any elements on which `A` depends, we can build an element of type `Dec A` then we have both proven `A` to be decidable and computed the proof that `A` does or does not hold.

We end this tutorial by using this methodology to prove that the equality between two natural numbers is decidable:

```
53  dec≡ : ∀ {x y : ℕ} → Dec (x ≡ y)                                      53
54  dec≡ {zero} {zero}  = yes refl                                        54
55  dec≡ {zero} {suc y} = no (λ ())                                       55
56  dec≡ {suc x} {zero} = no (λ ())                                       56
57  dec≡ {suc x} {suc y} with dec≡ {x} {y}                                57
58  dec≡ {suc x} {suc y} | yes x≡y = yes (cong suc x≡y)                   58
59  dec≡ {suc x} {suc y} | no ¬x≡y = no (λ {refl → (¬x≡y refl)})          59
```

From the patterns in a function line, AGDA allows us to add extra arguments to pattern match on in order to collect additional information from the base inputs. This is done with the `with` keyword. For instance here, we want to recursively call our function and split cases on its result. We add this argument to the pattern matching on line 57 and this results in 2 new cases: `yes` line 58 and `no` line 59. There is no limitation on how many new arguments one can add to a function. Each time it is done, the derived cases will appear as new lines in the function, for which outputs will have to be provided.

Before moving on to the next section, we can have a closer look at this example. We split cases on the two implicit arguments that are the two natural numbers we want to compare. When both of them are equal to `zero`, we can provide a proof

that they are equal, which is `refl` by definition. In the cases that one of them is equal to `zero` but the other is not, they are obviously not equal so we will return `no`. However, we have to provide a proof that they are not, which should be encapsulated in the `no` constructor. This proof should be a function that takes a proof of equality and returns false. Since there is no way to build such a proof in these cases, then this function is just the empty function `λ ()`. Note that λ is the operator used to define anonymous functions as explained in Section 2.3.2.b.

When both inputs are built from the `suc` constructor, we can decide by recursively calling the function on the arguments of this constructor, which is done using the `with` keyword, as explained above. In case this new call returns `yes` then we just combine the resulting proof with our `cong` property. In the other case, we have to provide a function which takes a proof that `suc x ≡ suc y` and builds a proof of ⊥. This is done by applying the proof that `¬ x ≡ y` to the result of the application of our lemma `suc≡` on the proof that `suc x ≡ suc y`. We have now proven the equality between two natural numbers decidable and hence provided AGDA with a way of building its related proofs.

2.3.5 AGDA vs COQ: a matter of taste

While the present section gave an overview of AGDA, it did not actually compare it with other similar languages like COQ. They are similar because they are both based on intuitionistic type theory, but they differ in many ways. This section aims to compare them objectively, rather than trying to find which one is the best. I believe neither is above the other, but that they both exhibit pros and cons depending on what they were designed to achieve. This section will put some perspective towards these differences and possibly help to choose between them depending on the targeted use. We will also explain why, considering these differences, we used AGDA for our development.

2.3.5.a Tactics

In dependently typed languages, thanks to the Curry-Howard isomorphism, dependent types can represent properties over the data on which they depend. This means that providing an element of a correct type is equivalent to proving the properties represented by this type. The element one has to provide is a lambda term that represents the proof. Writing these terms by hand is rather difficult (if not impossible) when it comes to non-trivial proofs. This is why such languages must provide a way of building these terms more intuitively. Both AGDA and COQ provide such help, but they do it quite differently.

COQ chooses to completely hide the terms and to provide higher level procedures to internally build them. They are called tactics and they work on hypothesis and goals to build the terms step by step. Although the underlying term is hidden, it exists nonetheless and can be retrieved if required as well as explicitly written when needed – usually in some very specific cases – also it can only be done by users with a strong expertise in the tool as well as its underlying theory. A COQ

proof usually consists in the invocation of several tactics whose behaviour can be compared to logical rules. New tactics can be created at will if they are written using COQ's tactic language. The benefit of this approach is the hiding of potentially useless technical details and difficulties. The main drawback is that these tactics must be somehow learned by the user before he can actually develop any proof.

AGDA chooses to display everything. The user has to provide the actual lambda term and cannot rely on tactics to build it for them. This can be confusing at first because these terms, as previously said, are usually too complicated to comprehend, but AGDA provides a convenient way to build them. Instead of writing them entirely in one go, the user will create them step by step using a system of holes, which represents unresolved goals, as explained in detailed in Section 2.3.5.b. Each hole is a small brick in the proof construction and can potentially be refined several times to further ease the proof effort. The benefit of this approach is the direct handling of the preoccupation (proof part) the user wants to address. The main drawback is the apparent complexity of the terms the user has to provide. An AGDA proof is also often hard to comprehend after it has been validated by the type checker. However, once it is proven type-correct, there is usually no need to try and understand the proof beyond its building.

Figure 2.14 provides visuals to assess how different these two ways of building proof terms are. In this example, both COQ and AGDA are used to prove the commutativity of the addition between natural numbers. They both use two lemmas to reach the final proof and they display the same overall structure. Figure 2.14a displays the proofs in COQ, where the different lemmas and theorems are proved using a succession of tactics – note that this is by far not the simplest way to do this proof in COQ, as shown later on in Figure 2.16a, but it is understandable. Figure 2.14b displays the proofs in AGDA, where no tactics are provided and the lambda terms are visible. It is important to note that these terms are mostly the same internally in both tools, only the view the user has changes as well as the way they were built.

Figure 2.15 shows the COQ lambda term for the first lemma in Figure 2.14a. This term looks more complex than the AGDA's version, but this is only due to AGDA's implicit arguments as well as COQ's expansion of the `cong` function.

2.3.5.b Automation

When it comes to theorem proving, automation is an important feature. It helps focusing on difficult aspects of the proofs while leaving the easiest part to solvers. They work faster and they provide safe results with arguably no possible mistakes when exported in tools like COQ and AGDA, as the output of these solvers will create a lambda term which will be type-checked, hence proved correct toward the target goal.

In COQ, this automation is embedded through the use of high level tactics whose underlying behaviour is hidden. They can call external solvers (usually ATPs in the form of SAT or SMT solvers as described in Section 2.1.4) or use other tactics to eventually provide a well typed term. These tactics can fail depending on the heuristics they use, but they are solid assets in building simple proof terms. For

(a) Addition commutativity in COQ (b) Addition commutativity in AGDA

Figure 2.14: Comparison of addition commutativity proofs

```
pluszero =
fun b : nat =>
nat_ind (fun b0 : nat => b0 = b0 + 0) eq_refl
  (fun (n : nat) (sn : n = n + 0) =>
   eq_trans (f_equal (fun f : nat -> nat => f n) eq_refl) (f_equal
S sn)) b
     : forall b : nat, b = b + 0
```

Figure 2.15: A lambda term displayed by COQ

instance, our example in Figure 2.14a could have equally been proved as shown on Figure 2.16a, in which case only the syntactic aspects of the language are present. The proof only consists in tactics relying on automation to work, with the adequate imports for these tactics to succeed in this specific case. For instance, the tactic `intuition` here relies on the use of `Omega` which must be imported.

In AGDA, the proof with automation is depicted in Figure 2.16b. This figure is exactly the same as Figure 2.14b, which could be expected since AGDA proofs consist in raw lambda terms. Since the provers generate these terms, the automation must be handled while editing the file, and has no impact on the final result. AGDA uses automation in the process of edition. This is made possible by the dif-

```
Require Import Arith.
Require Import Omega.
Section Commut.

Theorem pluszero b : b = b + 0.
Proof.
auto.
Qed.

Theorem plussn b n : S (b + n) = b + S n.
Proof.
auto.
Qed.

Theorem comm a b : a + b = b + a.
Proof.
intuition.
Qed.
```

(a) Addition commutativity in COQ with automation

```
open import Data.Nat
open import Relation.Binary.PropositionalEquality

module Commut where

+0 : ∀ {b} → b ≡ b + 0
+0 {zero } = refl
+0 {suc _} = cong suc +0

+sn : ∀ {b n} → suc (b + n) ≡ b + suc n
+sn {zero} = refl
+sn {suc n} = cong suc +sn

comm : ∀ {a b} → a + b ≡ b + a
comm {zero} = +0
comm {suc a} = trans (cong suc (comm {a})) +sn
```

(b) Addition commutativity in AGDA with automation

Figure 2.16: Comparison of addition commutativity proofs with automation

```
n≡n+0 : ∀ {b} → b ≡ b + 0
n≡n+0 {b} = ?
```

(a) Initial user input

```
n≡n+0 : ∀ {b} → b ≡ b + 0
n≡n+0 {b} = { }0
```

(b) Creation of a goal

```
n≡n+0 : ∀ {b} → b ≡ b + 0
n≡n+0 {b} = {b█}0
```

(c) Before splitting on `b`

```
n≡n+0 : ∀ {b} → b ≡ b + 0
n≡n+0 {zero} = {█}0
n≡n+0 {suc b} = { }1
```

(d) After splitting on `b`

```
n≡n+0 : ∀ {b} → b ≡ b + 0
n≡n+0 {zero} = refl
n≡n+0 {suc b} = {█}0
```

(e) `refl` found

```
n≡n+0 : ∀ {b} → b ≡ b + 0
n≡n+0 {zero} = refl
n≡n+0 {suc b} = {cong█}0
```

(f) Input of `cong`

```
n≡n+0 : ∀ {b} → b ≡ b + 0
n≡n+0 {zero} = refl
n≡n+0 {suc b} = cong {█}0 { }1
```

(g) Refining via `cong`

```
n≡n+0 : ∀ {b} → b ≡ b + 0
n≡n+0 {zero} = refl
n≡n+0 {suc b} = cong suc {█}0
```

(h) Input of `suc`

```
n≡n+0 : ∀ {b} → b ≡ b + 0
n≡n+0 {zero} = refl
n≡n+0 {suc b} = cong suc n≡n+0
```

(i) Recursive call found

Figure 2.17: AGDA's automation

ferent interaction the AGDA editor provides, depicted in Section 2.3.3. However the automation is limited compared to COQ. It consists in finding simple terms and refining holes regarding a specific function. It is constantly upgraded by the AGDA developers but still lacks power compared to COQ.

AGDA's automation is depicted in Figure 2.17. This picture presents the different steps in the development of the first property found in Figure 2.16b. Since the automation in AGDA is found during the process of development and is invisible in the resulting code, this picture details the actions done by the user and how AGDA responded to help them prove the required property. The steps are as follows:

a This is the initial code the user has to input. It features the prototype of the property as well as the introduction line to start the proof, and a specific

symbol to represent the goal, `?`.

b After pressing ctrl+c, ctrl+l, AGDA created a hole for the unresolved goal.

c The user inputs the variable on which he desires to case-split, `b`.

d After pressing ctrl+c, ctrl+c, AGDA created as many lines as possible cases for the variable `b`.

e After pressing ctrl+c, ctrl+a, AGSY solved the first goal and placed `refl` at its place.

f The user inputs the function that he wishes to expand, `cong`.

g After pressing ctrl+c, ctrl+r, AGDA agreed that applying `cong` could indeed lead to a correctly typed term, and refined the goal into two new subgoals corresponding to the arguments of `cong`.

h `suc` is written in the hole by the user, which is validated by pressing ctrl+c, ctrl+Space.

i After pressing ctrl+c, ctrl+a, AGSY solved the last goal by using a recursive call and inferring the implicit arguments of this call.

2.3.5.c Usability

The usability of academic tools is, unfortunately, often neglected compared to their theoretical aspects. This is easily understandable since funds allocated for research purposes are limited, hence usually used to process these theoretical aspects instead of more practical preoccupations. Even though this is understandable, it can be critical since complicated tools usually need a well-designed usability to ease the handling of their underlying complexity.

In COQ, this complexity is already softened by the use of tactics, and the graphical tools used to edit COQ code are sufficient, but do not bring any more improvements than those provided by the language itself. Except for the display of the current context and goal, these tools currently lack, to my knowledge, additional features that would help the COQ developer to achieve his work because they are based on COQTOP which is the primary command line tool that allows user interactions with COQ. For instance, COQ does not natively provide mixfix operators nor unicode identifier and the tools do not allow any additional interaction between the user and the core of COQ. However, this is questionable if this would be actually useful. For instance, the points of interaction available in AGDA are useful because of AGDA's way of developing programs, which is definitely different from COQ's, which means that having the same features in COQ would not necessarily be a plus, except probably for the aesthetic ones and some ergonomic concerns, which have been or are currently possibly being integrated.

AGDA is more recent than COQ and embeds several high-level features, both in its language and tools. These features have already been presented in Chapter 2.3 and will not be revisited here. They provide a pleasant development experience for the user and allow us to make AGDA proofs look like paper mathematical proofs. This is a huge asset for the language and the feeling it provides. It also makes using AGDA possible since all the complexity of the underlying theory and notions is visible.

2.3.5.d Unification

```
Section Unif.

Theorem transEq (A : Type) (x y z : A) :
  x = y -> y = z -> x = z.
Proof.
intros xeqy yeqz.
rewrite xeqy.
exact yeqz.
Qed.

Theorem congEq (A B : Type) (x y : A) (f : A -> B) :
  x = y -> f x = f y.
Proof.
intro xeqy.
rewrite xeqy.
reflexivity.
Qed.
```

(a) Some proofs about equality in COQ

```
module Unif where

trans≡ : ∀ {a} {A : Set a} {x y z : A} →
  x ≡ y → y ≡ z → x ≡ z
trans≡ {x = x} {.x} {.x} refl refl = refl

cong≡ : ∀ {a b} {A : Set a} {B : Set b} {x y} →
  (f : A → B) → x ≡ y → f x ≡ f y
cong≡ {x = x} {.x} f refl = refl
```

(b) Some proofs about equality in AGDA

Figure 2.18: Comparison of equality proofs

One of the major features which AGDA fully embraces and which is less visible in COQ is the unification mechanism. People usually summarize this difference by stating that one can, in AGDA, pattern match on equality proofs. It is possible indeed, since the equality is defined as a data type with a constructor, but this is only one of the many consequences of the unification mechanism, which provides a much larger set of possibilities. Figure 2.18 displays two proofs about equality, proved both in COQ and in AGDA and illustrates the possibility of pattern matching on equality proofs. In dependent types, equality is usually defined using a family of types generated by the reflexivity. In other words, the reflexivity allows, for any x, to inhabit the family $x =?$ for a single index, which is x, as shown in Section 2.3.4. This means that every proof of equality will eventually be reduced to the constructor of these families. Let us call it `refl` since this is how it is named in AGDA. In COQ, `refl` is always hidden and is generated from higher level tactics which work on equality proofs. In AGDA however, `refl` is always visible and usually is the result of a pattern matching on an equality proof. If the context possesses an element of type $x = y$ for some x and some y of any type, pattern matching on this term will have two consequences: the only possible constructor, `refl` will be displayed, and the two operands, x and y, will be unified, which means that AGDA will understand that both quantities have no possibilities other than being the same. The second operand, y, is replaced by $.x$ which means the value has been inferred to be x.

Figure 2.18 displays the proof of two basic properties about equality: the tran-

sitivity and the congruence. In COQ, these properties are proved using a special tactic, `rewrite`, which allows us to replace one operand of the equality with the other in a given expression. In AGDA, it is done by pattern matching on the different elements of the context, thus unifying the operands with each other, then by providing `refl` as an answer, since the goal also varies considering the unification that has been done. In the first proof, y is first unified with x, then z is unified with y (hence with x) and the goal is then just $x = x$ whose proof is simply `refl`. As mentioned before, this unification mechanism is easily illustrated through equality proofs, but is not reduced to them. Further examples will be given throughout this manuscript.

2.3.5.e Prop, Set, Type

Dependently typed languages provide, in one single language, a formalism to write programs and to prove properties about them. Several other tools provide either one or the other, but the expressiveness of dependent types allows both to coexist in a single framework. This is particularly useful because both sides can directly be connected to the other without any additional glue. However, it has a drawback. When compiling[5] such a language, all the artefacts around the program, whose goal is to certify it regarding safety or other kind of properties, must be ignored as much as possible because they do not participate in the computational aspects of the program. This means that despite being developed in the same language, the logical and computational aspects must somehow be separated in order for the code extractor to figure out what to process and what not to.

In COQ, this separation is made explicit by some syntactic sugar. In dependently typed languages, there is a name for types. In COQ, this name is split in three different possible names: `Type`, `Set` or `Prop`. `Prop` is used to define data that is only relevant in the logical part of the work. `Set` is used to define data that is only relevant in the computational part of the work while `Type` is used to define meta types which can be instantiated either in the `Set` or in the `Prop` world. The COQ code extractor will only translate the data labelled `Set` and also the one labelled `Type` which have been instantiated by `Set` elements. The COQ code generator translates these elements into a CAML program which contains all the necessary elements regarding the computational aspects of the program. Note that there also exists support for other languages such as LISP.

In AGDA, there is no such syntactic sugar to help the compiler make the difference. Everything is defined as `Set` (with the universe variant $\texttt{Set}_1$, $\texttt{Set}_2$, ... used to forbid paradoxes in the usual ZF or ZFC theories [137]). There is, however, a way of pointing out arguments which should be ignored both at evaluation and extraction time. These are called irrelevant arguments in AGDA and are marked with a dot as a prefix – the dot here is different from the one depicting a quantity whose value has been inferred by AGDA, which might be confusing for newer AGDA users, but since both situations cannot occur over the same identifier at the same position

[5]the compilation here can be seen as a code extraction

```
module Irrelevance where

data SList (bound : ℕ) : Set where
  []    : SList bound
  scons : (head : ℕ) →
         .(head ≤ bound) → -- note the dot!
          (tail : SList head) →
          SList bound
```

Figure 2.19: An example of computationally irrelevant argument

in the code, there is no actual confusion. They offer less flexibility than COQ's system but are far less verbose. An example, extracted from AGDA wiki, is depicted in Figure 2.19. This represents sorted lists that only contain elements that are below a certain threshold named `bound`. In the second constructor, `scons`, the second parameter is prefixed by a dot, which means that it is only relevant for the correctness of the program (through type-checking) but not for runtime evaluation or code generation. This parameter is the proof that the head is below the threshold. Since the definition is recursive, all the proofs of inferiority will be ignored in the case of code generation or real time evaluation. Furthermore, any subsequent proof of equality between quantities of this type will be made easier since the equality between the irrelevant arguments can be omitted. An example of this omission is displayed in Figure 2.20. In this example, we prove that two sorted lists, $\texttt{sl}_1$ and $\texttt{sl}_2$, are equal. We deliberately build the proofs of inferiority differently for the two lists (we postulate the proofs for the sake of this example in the block "postulate" and we actually build them in the block "private"). The two lists are then built from these different sets of proofs but can still be proven equal because these proofs are irrelevant and thus ignored in this case. This is why the type checker accepts `refl` as a proof of equality between the lists.

If we remove the dot in the definition, we get the message displayed in Figure 2.21. This message states that the proofs have not been proven propositionally equal and thus the lists cannot be equal themselves (`refl` is no longer sufficient because of the sudden relevance of the inferiority proof). In this specific case, it is possible (and easy) to prove that they indeed are (there is only one way to build a given proof of inferiority) but for more complex proof elements, it is advised to use the irrelevant mechanism.

2.3.5.f Purpose & Renown

Until now, we compared several technical aspects of both tools, but they also differ through their purpose, and it often justifies why they differ technically. COQ is meant to be a proof assistant. It was designed to offer a development experience centred around the logical aspects of the programs through the use of tactics. As such, it focuses on its tactic language to give the developer tools to manipulate their data and the predicates around them. AGDA was designed to be a programming language before being a proof assistant. It was meant to ease the handling of de-

```
postulate
  5≤5₁ : 5 ≤ 5
  3≤5₁ : 3 ≤ 5
  0≤3₁ : 0 ≤ 3

private
  5≤5₂ : 5 ≤ 5
  5≤5₂ = s≤s (s≤s (s≤s (s≤s (s≤s z≤n))))
  3≤5₂ : 3 ≤ 5
  3≤5₂ = s≤s (s≤s (s≤s z≤n))
  0≤3₂ : 0 ≤ 3
  0≤3₂ = z≤n

sl₁ : SList 5
sl₁ = scons 5 5≤5₁ (scons 3 3≤5₁ (scons 0 0≤3₁ []))

sl₂ : SList 5
sl₂ = scons 5 5≤5₂ (scons 3 3≤5₂ (scons 0 0≤3₂ []))

sl₁≡sl₂ : sl₁ ≡ sl₂
sl₁≡sl₂ = refl
```

Figure 2.20: AGDA does not evaluate irrelevant elements

```
5≤5₁ != s≤s (s≤s (s≤s (s≤s (s≤s z≤n)))) of type 5 ≤ 5
when checking that the expression refl has type sl₁ ≡ sl₂
```

Figure 2.21: AGDA evaluates all relevant elements

pendent types in order to apply them to concrete concerns. This original purpose is what makes both tools look so different while they ultimately share the exact same core. This is also why several choices were made differently as they aim to suit the purpose of the tool better.

While COQ is far older than AGDA, both tools provide unique and different ways of building theories and proving properties around them. COQ, however, is far more used, mostly due to its wider standard library and its older creation. AGDA is being used more and more in the academic world and not only in the laboratories where it was created in 2007. It is also taught in different universities as well as in several summer schools [126]. Its reputation is yet to be fully made, but it is a solid and trustworthy alternative to COQ.

2.3.5.g Why I chose AGDA instead of COQ

This section, while purposely detailing the differences between both tools, ultimately aims at explaining why this thesis used AGDA instead of COQ. While neither is objectively better than the other (they do not have the same purpose so are hardly comparable in terms of overall quality) there are reasons why AGDA was our choice in this work.

The first reason is purely circumstantial. I had the chance to meet Ulf Norell at a summer school, and he succeeded at transmitting the underlying complexity and beauty of what he had created. This is purely subjective but it made me want to know more about the tool, which ultimately led to its use in this PhD.

Another reason is the overall design of the tool. While COQ's tactics are meant to ease the development of proofs, the design of the IDEs around COQ did not appeal to me enough to use it within this work. The elegance of unicode names, mixfix operators along with the overall design of AGDA made me want to use it instead of its elder.

A third reason is that I wanted to give a chance to a less used tool. COQ has already proved itself and giving a chance to growing tools like AGDA was for me a good starting point in my work. I also had more chances of discovering fancy ways of using AGDA and maybe to contribute to its growing community and library.

Finally, we used AGDA because our first work was rather concrete and AGDA is designed to help the developer build concrete programs surrounded by possible logical aspects. Since we wanted to model PETRI NET and then drive the execution of languages, AGDA seemed well suited to our work.

Ultimately and above all of these arguments, using AGDA instead of COQ was mostly for me a matter of personal taste and feeling about the languages, as well as the will to explore new territories.

Assessments

This chapter, while not presenting any concrete contribution of this work, did describe notions which are central to its understanding. From a possible classification of the existing kinds of formal methods to the thorough depiction of the AGDA language and associated tools, it provided a set of relevant notions which will be useful for the reading of this manuscript. It also provided a comparison between AGDA and its cousin COQ which helps understand the goal of these tools and the relevancy of AGDA in such works.

The features of the language and its tools that were described and illustrated are the following: unicode characters, mixfix operators, associativity and priority of such operators, AGDA's identifiers, definition of data types, functions, predicates, proofs, AGDA's termination checker, type checker and AGSY, AGDA's automated term generator.

Using AGDA has been a thrilling challenge for me and I believe that it is core for the reader to be able to understand the main ideas between its usage as well as its location in the field of formal methods, which was described in this chapter. The rest of this work assumes that the reader is familiar with AGDA's syntax and its basic features, while introducing new notions when needed.

Chapter 3

Globally unique lists: an AGDA library

Outline

This chapter provides a description of a library that was developed during this thesis: globally unique lists. This library was developed using a specific methodology that is described as well. Globally unique lists are an abstraction of various notions, such as maps and bags.

- Section 3.1 presents the methodology that was used to build this library, and to a greater extent, throughout this whole work's formal implementation. For that purpose, it uses an example around the verification of basic operations on natural numbers.
- Section 3.2 proposes a formalization of unique membership in a list. This formalization is based on a notion of unique satisfaction of a given predicate on a list.
- Section 3.3 provides the notion of globally unique list, as well as several instantiations of this notion to better grasp its semantics. In short, a globally unique list is a list that contains at most a single value for each member of a family of predicates.
- Section 3.4 provides numerous operations on globally unique lists, such as adding an element, retrieving an element or removing an element. This section also provides examples of globally unique lists on which such operations are called, as well as two notions of comparison between such lists, one of which requires the retrieval operation.
- Section 3.5 provides two specific instances of globally unique lists: maps and bags. These notions will be used in Chapter 4 to model the states of the transition systems.

3.1 Methodology

Before detailing the important notions defined in this library, I would like to talk about the methodology used to develop it. Since we use a language with dependant types – hence expressive enough to handle properties over data structures, functions and programs – we try to exploit the logical aspects of such languages as much as possible. Any data structure or predicate that is defined is also accompanied with some proofs of conformity. These proofs have a very specific purpose, which is to close the gap between expectations and reality. Let me explain this a bit further. When developing a system using the assistance of a computer, there are no guarantees that the developed system satisfies the requirements from a given user. This has always been a tough problem in computer science and more precisely in systems engineering. The user has a very specific idea on how the system should behave and one has to make sure that the modelled version of the system exhibits the expected behaviour.

In the case of more abstract entities, such as abstract functions or predicates, the same reasoning applies. When defining a data type for instance, one must give constructors that act as axioms on how to prove that this predicate holds – thanks to the Curry-Howard correspondence – and it is mandatory to assess the correctness of such data types by proving conformity properties that may seem obvious – a mathematician would probably consider them too trivial, although in my opinion they are not, as they ensure the correctness of the data type with regard to the specification. Same goes for functions, the correctness of which must be verified.

To illustrate the need of verification which will be explored throughout this whole document, we give an example in which we question how to verify a very simple notion: the addition of natural numbers. We investigate in that example the meaning of having confidence in a formal definition, and we also take this opportunity to present several ways of proving equality properties in AGDA. We start by defining the addition recursively, which is very common and has already been done several times in this manuscript for didactic purposes:

```
1  _+_ : ℕ → ℕ → ℕ                1
2  zero + b = b                   2
3  suc a + b = suc (a + b)        3
```

As stated before, this definition alone is not enough because one has to be sure that said addition embeds the semantics that is usually given to this operation. It must be proven associative...

```
4  associative+ : ∀ {a b c} → (a + b) + c ≡ a + (b + c)        4
5  associative+ {zero} = refl                                  5
6  associative+ {suc a} = cong suc (associative+ {a})          6
```

... and commutative at the very least to be able to trust the definition.

```
7   commutative+ : ∀ {a b} → a + b ≡ b + a                                  7
8   commutative+ {zero} = +0                                                 8
9     where                                                                  9
10      +0 : ∀ {a} → a ≡ a + zero                                            10
11      +0 {zero} = refl                                                     11
12      +0 {suc _} = cong suc +0                                             12
13  commutative+ {suc a} = trans (cong suc (commutative+ {a})) +suc          13
14    where                                                                  14
15      +suc : ∀ {a b} → suc (a + b) ≡ a + suc b                             15
16      +suc {zero} = refl                                                   16
17      +suc {suc _} = cong suc +suc                                         17
```

These proofs might seem useless as the definition could be considered enough, but in our opinion it is not. Nothing is considered true unless it has been proven in our proof environment. In addition, if one wants to define more advanced functions or operations relying on the addition (such as the multiplication) and if one wants to prove conformity properties for these new functions, it will then be possible to rely on the proofs about the addition to do so.

This is the opportunity for me to present several ways of proving equality properties in AGDA. First we start, as expected, by defining the multiplication (note that these definitions and the proof around them can naturally be found in the standard library, although the proofs might be built differently).

```
18  _*_ : ℕ → ℕ → ℕ                                                          18
19  zero * b = zero                                                          19
20  suc a * b = b + (a * b)                                                  20
```

The first property of correctness is the distributivity of the multiplication towards the addition. As explained before, it relies on a property over the addition, the associativity. In this case, we directly give the raw term for the proof of equality which corresponds to the first way one can treat equality proofs in AGDA, on lines 23 and 24.

```
21  distributive* : ∀ {a b c} → (a + b) * c ≡ (a * c) + (b * c)              21
22  distributive* {zero} = refl                                              22
23  distributive* {suc a} {b} {c} = trans (cong (c +_) (distributive* {a}))  23
24    (sym (associative+ {c} {a * c} {b * c}))                               24
```

The second property is the associativity of the multiplication. We present this property by using the `rewrite` keyword which allows us to substitute one term with another in the goal provided they are equal (this is the mechanism usually used by COQ for such proofs). Should we provide enough rewrite rules – by using the pipe symbol to chain them, the term in such proofs always ends up to be `refl`.

```
25  associative* : ∀ {a b c} → a * (b * c) ≡ (a * b) * c                25
26  associative* {zero} = refl                                            26
27  associative* {suc a} {b} {c} rewrite distributive* {b} {a * b} {c}    27
28    | sym (associative* {a} {b} {c}) = refl                            28
```

The last conformity property we would like to prove is the commutativity of the multiplication. It requires two additional intermediate lemmas. This first one is trivial and does not require any equality combination.

```
29  *0 : ∀ {a} → a * 0 ≡ 0                                               29
30  *0 {zero} = refl                                                     30
31  *0 {suc a} = *0 {a}                                                  31
```

The second one, however, is more complex and is treated here with a convenient library provided by AGDA which allows us to chain equalities by providing each required equality proofs in bracket to transform the term `a` to the term `b`. The operators `begin` and ∎ are used to wrap up the proof type-wise while the chain operator `a ≡⟨ p ⟩ b` states that `a` is equal to `b` using the proof term `p`. `a ≡˘⟨ p ⟩ b` is a variant of this operator which goes from `a` to `b` using the symmetric of `p`.

```
32  *suc : ∀ {a b} → a * suc b ≡ a + (a * b)                             32
33  *suc {zero} = refl                                                   33
34  *suc {suc a} {b} = cong suc (begin                                   34
35    b + (a * suc b) ≡⟨ cong (b +_) (*suc {a} {b}) ⟩                    35
36    b + (a + (a * b)) ≡˘⟨ associative+ {b} {a} {a * b} ⟩               36
37    (b + a) + (a * b) ≡⟨ cong (_+ (a * b)) (commutative+ {b} {a}) ⟩    37
38    (a + b) + (a * b) ≡⟨ associative+ {a} {b} {a * b} ⟩                38
39    a + (b + (a * b)) ∎)                                               39
```

As one can see from the three ways of proving equality (giving the term directly, using rewrite or using chained equalities), the latter is by far the most comprehensible one and should be advocated as often as possible. It shows every steps of the reasoning in a convenient chain of proofs. This is why we also use it to wrap up this example by proving the commutativity of the multiplication. Note that this proof uses a lot of properties over the addition through the `*suc` lemma mostly.

```
40  commutative* : ∀ {a b} → a * b ≡ b * a                               40
41  commutative* {zero} {b} = sym (*0 {b})                               41
42  commutative* {suc a} {b} = begin                                     42
43    (b + (a * b)) ≡⟨ cong (b +_) (commutative* {a}) ⟩                  43
44    (b + (b * a)) ≡˘⟨ *suc {b} ⟩                                       44
45    (b * suc a) ∎                                                      45
```

In addition to these conformity properties, it is possible to give examples on how these constructs behave. The purpose of these examples is not to test the elements which were defined, since the formal approach replaces the testing method by a proving method, but rather they can be seen as a form of validation of punctual requirements over these elements while being accessible for an audience not always familiar with the formal method domain. Let us start by giving an example of addition.

```
46  2+5≡7 : 2 + 5 ≡ 7                                                    46
47  2+5≡7 = refl                                                         47
```

Since AGDA computes the value of `2 + 5` it is natural for the proof term to be `refl` as long as the property is correct. Should the property be incorrect, then there would be no possible inhabitant of the type. However, just inputing `refl` is both not very instructive as to how the result is computed and also not really relevant as an example. This is why it's possible to show the computational steps by using a succession of equalities as presented earlier. This leads to the following example, which is a lot more instructive.

```
48  2+5≡7₁ : 2 + 5 ≡ 7                                                   48
49  2+5≡7₁ = begin 2 + 5 ≡⟨⟩ 1 + 6 ≡⟨⟩ 7 ∎                                49
```

Note that in this case, the resulting term is of course `refl` as well, but the syntactic sugar used here allows a better presentation of the result. We continue and end this lecture grid with an example of muliplication following the same pattern of presentation (without unfolding the addition computation):

```
50  45*89 : 3 * 45 ≡ 135                                                 50
51  45*89 = begin 3 * 45 ≡⟨⟩ (45 + (2 * 45)) ≡⟨⟩ (45 + (45 + 45)) ≡⟨⟩ 135 ∎ 51
```

Note that, for any expression, it is possible to ask AGDA to reduce this expression to its simpliest form, which includes function evaluation when possible. In the two previous cases, should we have asked AGDA for the reduced expression – through the EMACS mode – of `2+5` and `3*45`, AGDA would have naturally answered `7` and `135` respectively in a dialog buffer.

3.2 Unique membership in a list

3.2.1 The Any Unique predicate

With these examples of addition and multiplication in mind, it is possible to use the same methodology to create our own constructs and predicates that will eventually lead us to the definition of maps and the structural description of event-based languages. Since we need a predicate over a list that states if a single element

$$\frac{A : Set \quad P : Pred\ A}{Any : Pred\ (List\ A)}\ \text{Any}$$

(a) The typing rule

$$\frac{P\ x}{Any\ P\ (x :: xs)}\ \text{here} \qquad \frac{Any\ P\ xs}{Any\ P\ (x :: xs)}\ \text{there}$$

(b) The construction rules

Figure 3.1: The `Any` data type

of this list satisfies an element `P` of type `Pred A` (a predicate over `A`), it is natural to start with the `Any` predicate. The standard library of AGDA, and most likely the standard library of any language using dependent types provides a data type that represents the lists containing at least one element satisfying a given property `P`. This data type is called `Any` in AGDA and the typing rules as well as its inference rules coming from its constructors are depicted on Figure 3.1.

As a unicity counterpart for this data type, we define a representation of lists with one and only one element satisfying the predicate `P`. This is very useful in this part of our work because we need to represent maps, and keys in maps must appear only once inside it. This new predicate is called `Any!` as `!` usually represents unicity in mathematics.

```
1  data Any! {a b} {A : Set a} (P : Pred A b) : Pred (List A) (a ⊔ b)      1
2    where                                                                  2
3      here! : ∀ {x xs} → P x → ¬ Any P xs → Any! P (x :: xs)               3
4      there! : ∀ {x xs} → ¬ P x → Any! P xs → Any! P (x :: xs)             4
```

This data type can be represented with inference rules, as shown on Figure 3.2. These inference rules are somewhat more complicated because they need to take into account the fact that, provided an element satisfying `P` has been found, there are no other elements in the list that satisfy `P`.

$$\frac{A : Set \quad P : Pred\ A}{Any! : Pred\ (List\ A)}\ \text{Any!}$$

(a) The typing rule

$$\frac{P\ x \quad \neg Any\ P\ xs}{Any!\ P\ (x :: xs)}\ \text{here!} \qquad \frac{\neg P\ x \quad Any!\ P\ xs}{Any!\ P\ (x :: xs)}\ \text{there!}$$

(b) The construction rules

Figure 3.2: The `Any!` data type

As an example of satisfaction of `Any!`, we provide a proof that there is only one number smaller than 3 in a given list. Before presenting the proof, here are the data types representing the lower or equal relations and the strictly lower relations between natural numbers (both taken from the standard library):

```
5    data _≤_ : Rel ℕ lzero where                                    5
6      z≤n : ∀ {n} → zero ≤ n                                         6
7      s≤s : ∀ {m n} → m ≤ n → suc m ≤ suc n                          7
8    -                                                               8
9    _<_ : Rel ℕ _                                                   9
10   m < n = suc m ≤ n                                               10
```

Lines 6 and 7 represent the two constructors that can build a proof that a natural number is lower or equal than another. The first constructor, `z≤n` states that 0 is lower or equal to any other natural number, while the second constructor, `s≤s` states that the successor operator preserves this relation. As for the strict precedence, it is defined from its non-strict counterpart on line 10. The example proof is then done as follows (note that ↪ is used to delimit elements in the list in order to differentiate this operator from the comma in the product type):

```
11   3< : Any! (_< 3) (⟦ 5 ↪ 2 ↪ 3 ⟧)                                                      11
12   3< = there! (λ {(s≤s (s≤s (s≤s ())))}) (here! (s≤s (s≤s (s≤s z≤n)))                   12
13     λ {(here (s≤s (s≤s (s≤s ())))) ; (there ())})                                       13
```

This proof can be represented as the deduction tree depicted on Figure 3.3. On this tree, there are three new inference rules that we use and are natively present in Agda. The first one, which I called "()" states that Agda can deduce `¬ P x` for a given input `x` and a given predicate `P` when no constructor of `P` could ever produce an element of type `P x`. For instance, there exists no constructor that can build an element of type `x < 0` for any `x`. The second rule, which I called "←s≤s" states that if for any `x` and `y`, `¬ x ≤ y`, then `¬ suc x ≤ suc y` can be proven as well. The third one, which I called "all()" states that if all the possible constructors for a given value cannot occur, then the predicate on this value does not hold, or rather its negation holds.

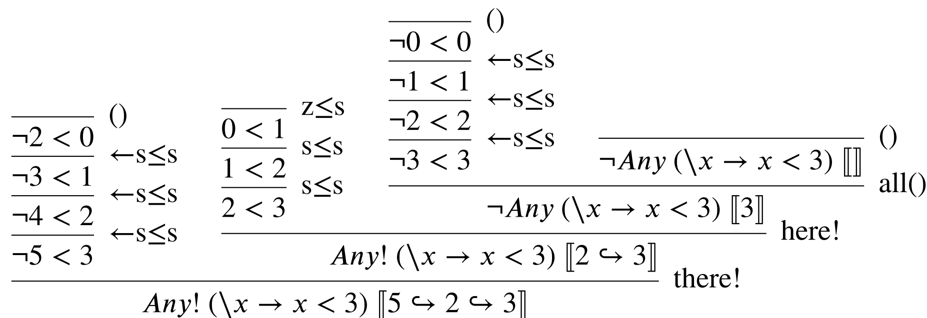


Figure 3.3: Satisfaction of `Any!` with the lower than 3 predicate

Other examples can be defined and studied, in order to acquire some more understanding on our definition. One such example is a list of strings containing only one string with a given length – here 5:

```
14  - _∘_ : Functional composition                                  14
15  s≡5 : Any! ((_≡ 5) ∘ length) ([[ "one" ↪ "two" ↪ "three" ]])   15
16  s≡5 = there! (λ ()) (there! (λ ()) (here! refl λ ()))           16
```

In this example, there is only one element (here the string "three") that contains five characters which has been proven accordingly. The intermediate proofs are very easy because they consist of empty functions since AGDA computes the size of the given string in the list, thus deducing whether or not their size is equal to five. This example is particularly interesting because it naturally leads to the notion of global unicity which will be described shortly after in this section. Indeed, we only selected the number five as input for this proof, and it is possible to select other numbers as well. For instance, had we selected three, then the predicate would not have been satisfied. The notion of global unicity, as stated before, will handle any allowed number in that case.

Figure 3.4 presents the deduction tree for this proof in which the evaluations of the size function have been hidden because they are done automatically by AGDA.

$$\dfrac{\dfrac{}{\neg|\text{"one"}| \equiv 5}\,() \quad \dfrac{\dfrac{}{\neg|\text{"two"}| \equiv 5}\,() \quad \dfrac{\dfrac{}{|\text{"three"}| \equiv 5}\,\text{refl} \quad \dfrac{}{\neg Any\ (|_| \equiv 5)\ [\![\,]\!]}\,()}{Any!\ (|_| \equiv 5)\ [\![\text{"three"}]\!]}\,\text{here!}}{Any!\ (|_| \equiv 5)\ [\![\text{"two"} \hookrightarrow \text{"three"}]\!]}\,\text{there!}}{Any!\ (|_| \equiv 5)\ [\![\text{"one"} \hookrightarrow \text{"two"} \hookrightarrow \text{"three"}]\!]}\,\text{there!}$$

Figure 3.4: Satisfaction of `Any!` with a given size of strings

3.2.2 Properties of Any Unique

It is required to prove some properties about these definitions as, for instance, the fact that `Any!` trivially implies `Any` with respect to the same predicate `P`. In order to prove these properties, we define some variables to lighten the type signatures. The subsequent elements will be implicitly parametrized with these variables:

```
17  a b : Level     17
18  A : Set a       18
19  P : Pred A b    19
20  x : A           20
21  l : List A      21
```

The first property which we provide states that if a value is a unique member of a list, then it is a member of this list. This is shown through a very simple recursive function.

```
22  A!→A : Any! P l → Any P l                                    22
23  A!→A (here! px _ ) = here px                                 23
24  A!→A (there! _ apl) = there (A!→A apl)                       24
```

This also allows us to prove some less trivial properties around this data structure, which will be useful in later definitions and proofs as well as in increasing the confidence in our definitions with respect to their expected semantics. The first one states that if an element `x` satisfies `P` as well as a list `l` then the list `x :: l` does not uniquely satisfy `P` because there are at least two elements in `x :: l` that satisfy `P` . Note that, by language abuse, we state that a given list `l` (resp. uniquely) satisfies `P` when there is at least (resp. exactly) one element in `l` that satisfies `P` ie. when `Any P l` (resp. `Any! P l`) holds.

```
25  P→A→¬A! : P x → Any P l → ¬ Any! P (x :: l)                  25
26  P→A→¬A! _ apl (here! _ ¬apl) = ¬apl apl                      26
27  P→A→¬A! px _ (there! ¬px _) = ¬px px                         27
```

The second property states that if `x` does not satisfy `P` and if `l` does not uniquely satisfy `P` then `x :: l` does not uniquely satisfy `P` – note that this is also true when `l` does not satisfy `P` because, by contraposition of the previous property, we deduce that `¬ Any P l → ¬ Any! P l`.

```
28  ¬P→¬A!→¬A! : ¬ P x → ¬ Any! P l → ¬ Any! P (x :: l)          28
29  ¬P→¬A!→¬A! ¬px _ (here! px _) = ¬px px                       29
30  ¬P→¬A!→¬A! _ ¬a!pl (there! _ a!pl) = ¬a!pl a!pl              30
```

The third property states that if `x :: l` uniquely satisfies `P` and `x` satisfies `P` then `l` does not satisfy `P`.

```
31  ¬A!→P→¬A : Any! P (x :: l) → P x → ¬ Any P l                 31
32  ¬A!→P→¬A (here! _ ¬apl) _ = ¬apl                             32
33  ¬A!→P→¬A (there! ¬px _) px _ = ¬px px                        33
```

The fourth property states that if neither `x` nor `l` satisfies `P` then `x :: l` does not satisfy `P` either.

```
34  ¬P→¬A→¬A : ¬ P x → ¬ Any P l → ¬ Any P (x :: l)              34
35  ¬P→¬A→¬A ¬px _ (here px) = ¬px px                            35
36  ¬P→¬A→¬A _ ¬apl (there apxl) = ¬apl apxl                     36
```

If we take a closer look at the type of this property, we can see that it exhibits a certain regularity. Indeed, the values on which the components of this signature depend on are `x`, `l` and `x::l` which makes it similar to the `cons` constructor that builds a new list from an element and a list. This regularity can be used to our advantage by some syntactic tricks that will allow us to use this property in an inductive manner when building proofs, which will be done in an upcoming example. In order to exploit this regularity, two steps are required: both these steps are purely syntactic. The first one consists of giving a name to the proof that no property is ever satisfied for elements in an empty list:

```
37  ¬A[] : ¬ Any P []                                    37
38  ¬A[] ()                                               38
```

The second step consists of creating an infix operator which allows us to chain the calls on this property (similar to the cons constructor of the lists). This infix operator is defined using the `syntax` keyword which allows us to create convenient syntactic aliases for already defined quantities. These aliases will be conveniently used later on in Section 3.2.4.

```
39  infixr 1 ¬P→¬A→¬A                                     39
40  syntax ¬P→¬A→¬A ¬px ¬apl = ¬px ↦ ¬apl                 40
```

In AGDA, it is possible to define aliases for any operator that was not defined using underscores (as a mixfix operator) but it is not always useful. For instance, the following example will only be useful to shorten the identifier, but not as an infix operator because the result of the property does not have the same structure as one of its parameters:

```
41  syntax P→A→¬A! yRx w₂ = yRx ¬! w₂                     41
```

3.2.3 Decidability of Any Unique

An interesting aspect of the predicates we define in this work is their decidability. Since we would like to use them in concrete applications (to model languages such as PETRI NET) we would like these predicates to be decidable provided the predicate on which they are based is decidable as well. More precisely, if we can either build, for any input `x`, the proof `P x` or the proof `¬ P x` then we should be able to build, for any list `l` the proof `Any! P l` or `¬ Any! P l`. This work is already done in the standard library for the `Any` predicate, through a property called `any`. Here, $\texttt{Decidable}_p$ is the AGDA notion of decidability for predicates (unary relations) which, as all notions of decidability, is based on the `Dec` data type depicted in Section 2.3.4. Here is the proof that `P` being decidable ensures that `Any! P` is decidable.

```
42  decAny! : Decidable_p P → Decidable_p (Any! P)
43  decAny! _ [] = no (λ ())
44  decAny! decP (x :: _) with decP x
45  decAny! decP (_ :: l) | yes _ with (any decP) l
46  decAny! _ _ | yes px | yes apl = no (P→A→¬A! px apl)
47  decAny! _ _ | yes px | no ¬apl = yes (here! px ¬apl)
48  decAny! decP (_ :: l) | no ¬px with decAny! decP l
49  decAny! _ _ | no ¬px | yes a!pl = yes (there! ¬px a!pl)
50  decAny! _ _ | no ¬px | no ¬a!pl = no (¬P→¬A!→¬A! ¬px ¬a!pl)
```

This proof is done by case splitting on the different proofs of decidabilty to build the final term in each case. Here are the cases:

- Line 43 : the list is empty. Since there are no constructors that build an element of `Any! P []` we can conclude `no`.
- Line 46 : the list is of the form `(x :: l)` and we have `P x` and `Any P l`. In this case we can conclude `no` by contradiction using the `P→A→¬A!` property.
- Line 47 : the list is of the form `(x :: l)` and we have `P x` and ¬ `Any P l`. In this case we can conclude `yes` using the constructor `here!`.
- Line 49 : the list is of the form `(x :: l)` and we have ¬ `P x` and `Any! P l`; In this case we can conclude `yes` using the constructor `there!`.
- Line 50 : the list is of the form `(x :: l)` and we have ¬ `P x` and ¬ `Any! P l`. In this case we can conclude `no` by contradiction using the `¬P→¬A!→¬A!` property.

3.2.4 Parametrized membership

Using these two definitions – `Any` and `Any!` – one being found in the standard library and the other being defined for the purpose of this work, we can define a parametrized notion of membership and unique membership. Given an heterogeneous relation `_R_` over two types `A` and `B`, an element `x` of type `A` and a list `l` of elements of type `B`, we state that `x` is a member (resp. a unique member) of `l` when there exists at least (resp. exactly) one element in `l` that is related to `x` by `_R_`, in other words such that the property `x R_` holds.

```
51  _∈_ : REL A (List B) _
52  x ∈ xs = Any (x R_) xs
53
54  _∈!_ : REL A (List B) _
55  x ∈! xs = Any! (x R_) xs
```

```
56  _∉_ : REL A (List B) _
57  x ∉ xs = ¬ (x ∈ xs)
58
59  _∉!_ : REL A (List B) _
60  x ∉! xs = ¬ (x ∈! xs)
```

These new relations are all decidable, provided that the underlying relation R is decidable as well. This is proved in the appendices in Section A.1.1

These parametrized memberships will be particularly useful when dealing with maps and some specific kind of sets that will be described later. In order to better comprehend this parametrization, here are two examples of usage with different instantiations of _R_. The first example uses lists of natural numbers and the strict precedence for _R_. If an element x is a member of a list l relatively to this relation, this means that there exists at least one element in l that is strictly greater than x. This example is conducted as follows: We start by importing the membership module with the desired parameter while also renaming some of the operators it exports in order to avoid conflicts in the future, in case of several imports of the same module with different parameters. (Note that it is also possible to rename the entire module, which will be done for instance in Section 7.1.2).

```
61  open Membership _<_ hiding (_∉_ ; _∈!_)                                61
62    renaming (_∈_ to _∈<_ ; _∉!_ to _∉!<_)                               62
```

Then, we define an example of a list which will be used to illustrate our notions of parametrized memberships.

```
63  list₁ : List ℕ                                                         63
64  list₁ = ⟦ 3 ↪ 5 ↪ 2 ⟧                                                  64
```

We can prove several membership properties about this list, but only two will be presented here. In both cases, the comparison proofs between natural numbers are hidden (they are trivial and automatically computed by AGDA). The first one is that 4 is a member of this list, since there exists a number, 5, that is in the list and that is greater than 4.

```
65  4∈list₁ : 4 ∈< list₁                                                   65
66  4∈list₁ = there (here 4<5)                                             66
```

The second one states that 2 is not a unique member of the list since both 3 and 5, that are in the list, are greater than 2. The proof is done by case splitting on the proof that 2 is a unique member of the list in order to produce an element of ⊥. In the case of 2 being related to the head of the list (2 is smaller than 3), on line 68, there is a contradiction, since it is also a member of the tail (2 is also smaller than 5). In the case of 2 only being part of the tail of the list, on line 69, there is a contradiction because 2 is smaller than 3.

```
67  2∉!list₁ : 2 ∉!< list₁ - 2 ∈!< list₁ → ⊥                               67
68  2∉!list₁ (here! _ ¬2∈<[5,2]) = ¬2∈<[5,2] (here 2<5) - gives ⊥          68
69  2∉!list₁ (there! ¬2<3 _) = ¬2<3 2<3 - gives ⊥                          69
```

The second example regarding membership is built over lists of strings and the relation `_R_` is this time heterogeneous: the membership of a character inside a string. In this example, a character is said to be a member of the list if there exists a string in the list that contains this character. We start by instantiating the example module with the right relation.

```
70  open Membership _∈l_ hiding (_∈_ ; _∉!_)                                   70
71    renaming (_∉_ to _∉s_ ; _∈!_ to _∈!s_)                                    71
```

Then we define a list of strings on which we will work.

```
72  list₂ : List String                                                          72
73  list₂ = ⟦ "Alice" ↪ "Bob" ↪ "Judith" ⟧                                       73
```

This time, the two properties that will be displayed here correspond to the two relations that have not been emphasized in the previous example. The proofs of a character being in a string are left aside, but can be found in the appendices Section A.1.3. The first proof is that 'z' is not a member of the list, because it is not in any of the strings the list contains which leads to contradictions.

```
74  z∉list₂ : 'z' ∉s list₂ - 'z' ∉s list₂ → ⊥                                   74
75  z∉list₂ (here z∈Alice) = z∉Alice z∈Alice - gives ⊥                          75
76  z∉list₂ (there (here z∈Bob)) = z∉Bob z∈Bob - gives ⊥                        76
77  z∉list₂ (there (there (here z∈Judith))) = z∉Judith z∈Judith - gives ⊥       77
78  z∉list₂ (there (there (there ()))) - no such possible case                  78
```

This proof is quite heavy in terms of code length, as well as somewhat redundant with some of the properties that have been proved beforehand. Since the goal here was to prove a non-membership, it is possible to substantially reduce the size of the proof by using the "inductive" syntax that was presented and discussed in Section 3.2.2. Here is the resulting proof displayed as a sequence of non-membership proofs thanks to the syntactic alias (combined with its priority and associativity) that was defined for the `¬P→¬A→¬A` property:

```
79  z∉list₂' : 'z' ∉s list₂                                                      79
80  z∉list₂' = z∉Alice ↦ z∉Bob ↦ z∉Judith ↦ ¬A[]                                80
```

This way of writing this proof is much more understandable, because it can simply be interpreted as the following: "Since 'z' does not appear in "Alice", nor in "Bob", nor in "Judith", and since no letter ever appears in the empty list, then 'z' does not appear in ⟦ `"Alice" ↪ "Bob" ↪ "Judith"` ⟧".

The second property is that 'J' is a unique member of the list because it is only contained in the string "Judith". Since this proof is a proof of membership, it can directly be built by the constructors of the `Any!` data type without having to use the

previous simplification method.

```
81  J∈!list₂ : 'J' ∈!s list₂                                         81
82  J∈!list₂ = there! J∉Alice (there! J∉Bob (here! J∈Judith λ ()))    82
```

This proof can easily be read as follows: "Since 'J' does not appear in "Alice" as well as in "Bob" but does appear in "Judith" and since no letter ever appears in the empty list, then 'J' is a unique member of ⟦ `"Alice"` ↪ `"Bob"` ↪ `"Judith"` ⟧". The reader may notice the symmetry of these proofs, thanks to the aliases we defined.

3.3 Global unicity

3.3.1 Definition

In a map, any key has to be represented a single time, which means that any key that is a member of a map has to be a unique member of this map. This reasoning leads to the definition of a new predicate on a list, which we call the global unicity. This predicate first relies on a predicate over a value and a list, which states that if this value appears once in the list, then it appears only once.

```
83  _∈∈!_ : REL _ _ _ - Agda figures out the missing part    83
84  x ∈∈! l = x ∈ l → x ∈! l                                 84
```

Note that this predicate does not imply that the value indeed is a member of the list. It only states that if it is, then it uniquely is. Assuming `_R_` is decidable, this predicate can be proven decidable as well using the decidability of the membership relations. This proof does not provide any new technical aspect and is only presented in the appendices in Section A.1.2

```
85  Dec∈∈! : Decidableᵣ _R_ → Decidableᵣ _∈∈!_    85
```

The globally unique predicate is then defined as follows:

```
86  GlobalUnicity : Pred _ _              86
87  GlobalUnicity l = ∀ {x} → x ∈∈! l     87
```

While it would be natural to consider that the `GlobalUnicity` predicate is necessarily decidable, it is actually a mistake in the general case. Indeed, this predicate encapsulates a universal quantifier which invalidates the proof of decidability in constructive proofs. To prove that `GlobalUnicity l` is decidable for any `l` we would have to exhibit all the values that are related to the elements of `l` in order to prove that no other value in `l` is related to the same elements. Exhibiting such values is impossible in our case, which means that this predicate is not decidable in the

general case. Fortunately, when instantiating the relation `_R_` with the appropriate relations (for instance to model maps), it will be.

From this predicate, a globally unique list is then simply a record containing a list and a proof of global unicity. (This could have equally be modelled using ∃ but such a modelling will only be detailed from Section 6.2.1.b on).

```
88   record GUList : Set (a ⊔ b ⊔ c) where                88
89     constructor gulist                                  89
90     field                                               90
91       content : List B                                  91
92       unique : GlobalUnicity content                    92
```

This record provides a constructor, `gulist`, the goal of which is to ease the creation of elements of type `GUList`. Indeed, instead of writing the usual record instantiation `record { content = c ; unique = u }` to create a globally unique list, one can simply write `gulist c u`.

Since we defined a new notion, it is now customary to prove conformity properties about it. The first property is the fact that a list with only one element is necessarily globally unique.

```
93   gu[a] : ∀ {a} → GlobalUnicity [ a ]                   93
94   gu[a] (here px) = here! px λ ()                       94
```

From now on we consider the additional following variables, which will be given as implicit parameters to subsequent definitions.

```
95   x : B                                                 95
96   l : List B                                            96
```

The second property that is shown here is the conservation of the global unicity when handling the tail of a list. Indeed, it is natural that, considering a globally unique list `l`, its tail should be globally unique too regarding the same underlying relation:

```
97   guTl : GlobalUnicity (x :: l) → GlobalUnicity l       97
98   guTl guxl y∈l with guxl (there y∈l)                   98
99   guTl _ y∈l | here! _ y∉l = contradiction y∈l y∉l      99
100  guTl _ y∈l | there! _ y∈!l = y∈!l                     100
```

While the fact that removing an element from a list does not change its global unicity seems pretty straightforward, adding an element to a list is more complicated. There has to be a given predicate that this new element should fulfill in order for global unicity to keep holding on the new list. To assess the nature of this prerequisite, the following question arises: what does a list being globally unique

actually stand for ? To answer this question, we can think of our underlying relation to being associated to an element as a witness for this element – note that this notion of witness is not that same as the witness in the ∃ type described in Section 6.2.1.b. In order to emphasize this new way of seeing the underlying relation, let us define new names for our quantities using the function `flip` which flips the parameters of a function, as follows: `flip = λ f x y → f y x`:

```
101  _witnesses_ : _                      101
102  _witnesses_ = flip _R_               102
103  - a witnesses b ⇔ b R a              103
```

```
104  _[witnesses]_ : _                    104
105  _[witnesses]_ = flip _∈_             105
106  - l [witnesses] b ⇔ b ∈ l            106
```

With these names in mind, it is easier to understand what a globally unique list stands for. It is basically a list that never contains two different witnesses for the same element. In other words, it should be a list which, for any element `y` that has two witnesses in it, these witnesses are always at the same position – `index` – in the list, thus being identical. Note that the function `index` returns an element of type `Fin (size n)` where `Fin` is the usual dependent type representing a set of `n` elements (from 0 to `n-1`), with two constructors `fzero` and `fsuc`. Let us express this property:

```
107  ≡index : _ → Set _                                                   107
108  ≡index l = ∀ {y} (w₁ w₂ : l [witnesses] y) → index w₁ ≡ index w₂      108
```

Then we can proceed by proving that a list that satisfies the `GlobalUnicity` predicate also satisfies the `≡index` predicate.

```
109  ⇒ : GlobalUnicity l → ≡index l                                       109
110  ⇒ _ (here _) (here _) = refl                                         110
111  ⇒ gul (here yRx) (there w₂) = ⊥-elim ((yRx ¬! w₂) (gul (here yRx)))  111
112  ⇒ gul (there w₁) (here yRx) = ⊥-elim ((yRx ¬! w₁) (gul (here yRx)))  112
113  ⇒ gul (there w₁) (there w₂) = cong fsuc (⇒ (guTl gul) w₁ w₂)         113
```

Let us take a closer look at that proof. We need to prove that if we have two proofs w_1 and w_2 that a given element `y` has a witness in a globally unique list `l` then these proofs can only point to the same element in that list. In order to proceed in the demonstration, we need to case-split on these proofs, which is the same as looking where they point to. There will be four possible cases bound to this case-split, two of them, on line 111 and 112 state that the two proofs actually point to different elements in the list (one points to the head and the other to the tail). These cases must somehow be impossible, and a contradiction has to be pointed out. The third case, on line 110 is when both proofs point at the head of the list and the last case, on line 113 is when both proofs point somewhere in the tail of the list. The four cases are handled as follows:

- case 1, line 110 directly leads to the result because both indexes pointed by the proofs are 1. This is the case that will eventually be reached every time by recursively calling the property.

- case 2 and 3, line 111 and 112 are similar because they are fully symmetrical. In these cases, as stated before, a contradiction needs to be pointed out. We know that the list is globally unique so we can apply this fact to the first proof that points out to the head. It gives us the fact that `y` has a unique witness in `l`. However, we also know that `y` has both a witness in the head of `l` and in the tail of `l`. Applying the property `¬P→¬A→¬A!` (with its alias `¬!`) gives us the fact that `y` has two witnesses in `l` which gives us the contradiction we were looking for.

- case 4, line 113 is solved by recursively calling our property on the tail of the list. Since we know that the tail of the list is still globally unique, thanks to the `guTl` property, we can call the property recursively with the right parameters. This gives us that both proofs point to the same location in the tail of the list, which, when adding one to both these indexes, proves that they also point to the same location in the whole list.

This result on globally unique lists also happens to be a characterization, which means that this `≡index` is an equivalent of `GlobalUnicity`. Proving this equivalence requires a helper that states that `x` and `l` cannot be both a witness of an element `y` when `x :: l` satisfies `≡index`.

```
114  helper : ∀ {y} → ≡index (x :: l) → x witnesses y → ¬ l [witnesses] y    114
115  helper ≡ind px x∈xs = case ≡ind (here px) (there x∈xs) of λ ()         115
```

This proof uses the construct `case of` which provides in-term case-splitting. In other words, it allows us to case-split on an intermediate value, the result of which is the empty pattern in this case, hence providing a proof of ⊥. This leads to the other side of the equivalence using a new syntactic sugar `x ⟨ f ⟩ y = f x y` which allows us to treat functions with two parameters as infix operators, and a proof of injectivity `fsuc-injective` for the constructor `fsuc` or the type `Fin n`:

```
116  ⇐ : ≡index l → GlobalUnicity l                                          116
117  ⇐ ≡ind (here px) = here! px (helper ≡ind px)                             117
118  ⇐ ≡ind (there x∈l) = there! (_⟨ helper ≡ind ⟩ x∈l)                      118
119     (⇐ (λ w₁ w₂ → fsuc-injective (≡ind (there w₁) (there w₂))) x∈l)       119
```

Now that we have shown precisely what a globally unique list actually stands for, we can easily formulate the predicate that a given value `x` must satisfy so it can be successfully added at the head of this list. For any value `y` that is witnessed by `x`, there must not exist any other witness of `y` in the list. This is expressed and proved as follows:

```
120  _∷ᵤ_ : (∀ {y} → x witnesses y → ¬ l [witnesses] y)
121    → GlobalUnicity l → GlobalUnicity (x ∷ l)
122  _∷ᵤ_ p _ (here yRx) = here! yRx (p yRx)
123  _∷ᵤ_ p gul (there y∈l) = there! (_⟨ p ⟩ y∈l) (gul y∈l)
```

If we add to this property the fact that an empty list is necessarily globally unique, it gives us an inductive way to build proofs of global unicity over lists, similarly to the way we exploited the `¬P→¬A→¬A` property:

```
124  gu[] : GlobalUnicity []
125  gu[] ()
```

As a consequence we can for instance very easily prove a second time that a list with a single element is globally unique.

```
126  gu[a]₀ : ∀ {a} → GlobalUnicity [ a ]
127  gu[a]₀ = (λ {_ ()}) ∷ᵤ gu[]
```

In order to emphasize the usefulness of the `_∷ᵤ_` operator even more, let us develop an example of a globally unique list. In this example, the relation will be the propositional equality, which means a globally unique list is a list with no duplicates in it. This example features intermediate proofs of non membership followed by the proof of being globally unique using inductive reasoning in each of its steps:

```
128  open GlobalUnicity {A = ℕ} _≡_
129  -
130  l : List _
131  l = ⟦ 3 ↪ 5 ↪ 2 ⟧
132  -
133  ¬⟦5,2⟧w3 : ∀ {y} → 3 witnesses y → ¬ ⟦ 5 ↪ 2 ⟧ [witnesses] y
134  ¬⟦5,2⟧w3 refl = (λ ()) ↦ (λ ()) ↦ ¬A[]
135  -
136  ¬⟦2⟧w5 : ∀ {y} → 5 witnesses y → ¬ ⟦ 2 ⟧ [witnesses] y
137  ¬⟦2⟧w5 refl = (λ ()) ↦ ¬A[]
138  -
139  ¬⟦⟧w2 : ∀ {y} → 2 witnesses y → ¬ [] [witnesses] y
140  ¬⟦⟧w2 _ ()
141  -
142  gul : GlobalUnicity l
143  gul = ¬⟦5,2⟧w3 ∷ᵤ ¬⟦2⟧w5 ∷ᵤ ¬⟦⟧w2 ∷ᵤ gu[]
```

3.3.2 Globally unique lists on different kind of relations

3.3.2.a Globally unique lists on equivalence relations

It is interesting to try and figure out what kind of relation could give a relevant meaning for globally unique lists. An example of such relations are equivalence relations. A globally unique list towards an equivalence relation contains at most one witness per equivalence classes bound to this relation. This sounds intuitive when using the concept of witnesses to talk about globally unique lists. Let us start by instantiating the GlobalUnicity module on an equivalence relation.

```
144 module GloballyUniqueEquivalence {a b} {A : Set a} {_≈_ : Rel A b}     144
145   (isEq : IsEquivalence _≈_) where                                   145
146   open GlobalUnicity _≈_                                             146
147   open Membership _≈_ renaming (_∈_ to _∈≈_)                         147
```

Then we can prove the fact that if two equivalent elements are part of a globally unique list over this equivalence relation, then they are necessarily the same.

```
148 gueq : ∀ {x y l} → x ∈ l → y ∈ l → GlobalUnicity l → x ≈ y → x ≡ y   148
149 gueq x∈l y∈l gul x≈y = ≡i→≡ (begin                                 149
150   index x∈l ≡⟨ i≡ic x∈l ⟩                                            150
151   index (∈→∈≈ x∈l)                                                   151
152     ≡⟨ ⇒ gul (∈→∈≈ x∈l) (trans∈≈ (≈sym isEq x≈y) (∈→∈≈ y∈l)) ⟩        152
153   index (trans∈≈ (≈sym isEq x≈y) (∈→∈≈ y∈l))                         153
154     ≡⟨ it≡i (≈sym isEq x≈y) (∈→∈≈ y∈l) ⟩                             154
155   index (∈→∈≈ y∈l) ≡˘⟨ i≡ic y∈l ⟩                                    155
156   index y∈l ∎)                                                       156
```

This proof depends on several lemmas – `∈→∈≈`, `trans∈≈`, `it≡i`, `i≡ic` and `≡i→≡` – that are presented in the appendices, in Section A.1.4. This proof is based on the equality of indexes.

3.3.2.b Globally unique lists on total orders

The same work can be done for instance when the relation is instantiated with a non-strict total order, in which case the globally unique lists are the lists with at most one element, which can be proved. Let us start by instantiating the globally unique lists on a total order.

```
157 module GloballyUniqueTotalOrder {a b c} {A : Set a} {_≈_ : Rel A b}   157
158   {_≼_ : Rel A c} (isTot : IsTotalOrder _≈_ _≼_) where               158
159   open GlobalUnicity _≼_                                             159
```

Then we can prove that these globally lists cannot indeed have more than 1 element. The proof is done by absurd. If the list has at least two elements then

one of them is smaller than the other, but is also smaller than itself, which results in a contradiction since it has two witnesses in the list which makes it not globally unique.

```
160  prop : ∀ {l} → GlobalUnicity l → ¬ Data.List.length l > 1                  160
161  prop {[]} _ ()                                                               161
162  prop {_ :: []} _ (s≤s ())                                                    162
163  prop {x :: y :: _} _ with total isTot x y                                    163
164  prop {_ :: _ :: _} gul | inj₁ x≼y =                                          164
165    contradiction (gul (here ≼refl)) (P→A→¬A! ≼refl (here x≼y))                165
166  prop {_ :: _ :: _} gul | inj₂ y≼x =                                          166
167    contradiction (gul (there (here ≼refl))) (P→A→¬A! y≼x (here ≼refl))        167
```

And we can also prove that any list of at most 1 element is globally unique using previous properties. Note that this property is always true regardless of the underlying relation, but it is an equivalence for the total orders.

```
168  prop₂ : ∀ {l} → ¬ Data.List.length l > 1 → GlobalUnicity l                  168
169  prop₂ {[]} _ = gu[]                                                          169
170  prop₂ {_ :: []} _ = gu[a]                                                    170
171  prop₂ {_ :: _ :: _} p = case (p (s≤s (s≤s z≤n))) of λ ()                     171
```

3.3.2.c Globally unique lists on other relations

Other logical relations Since we have investigated how such lists would behave when coupled with total orders or equivalence relations, it is possible to deduce what would be their behaviour with other kinds of logical relations. Globally unique lists are collections of elements that all have a distinct aspect. We can foresee that this property of separation will be preserved when considering strict total orders or partial orders. For partial orders, two elements in the list could not have a common ancestor in their timeline. As a consequence, they cannot be in the same timeline either. For strict total orders, if there is a smaller element, this lower bound can appear any number of times in the lists. Otherwise, the list will have the same properties as for non-strict total orders.

Functions Globally unique lists can be used to model collections of elements that do not have the same result when a given function is applied to them. This is done by considering, for a given function `f`, the relation `R` defined as follows: $yRx \Leftrightarrow y \equiv f\,x$. This aspect is particularly interesting and will be the one ultimately used to define maps and bags. In this case, the elements of the lists all have different images through the function `f`. As a consequence, if `f` is injective, the corresponding globally unique list can have any number of different elements. From this point on in this section, we will consider such relations.

3.3.2.d Globally unique lists over image equality

We define a short module which provides, given a function `f`, the `_≡f_` relation as well as the proof that it preserves decidability.

```
172  module FunctionRelation                                        172
173    {a b} {A : Set a} {B : Set b} (f : B → A) where              173
174    -                                                            174
175    _≡f_ : REL A B a                                             175
176    _≡f_ x = (x ≡_) ∘ f                                          176
177    -                                                            177
178    dec≡f : Decidable, {A = A} _≡_ → Decidable, _≡f_             178
179    dec≡f dec x = (dec x) ∘ f                                    179
```

From this point on, every module will be parametrized by such a function `f`, which means any globally unique list will embed this specific related semantics of these globally unique lists.

3.4 Commands and requests over globally unique lists

These globally unique lists parametrized with a function will be used to model maps and they should provide the usual commands their semantics induces. The first is `get` which, given a specific key, retrieves a specific value in a map. While the function that allows the comparison of keys is `f` itself, our current modelling lacks a second function, let us call it g which, in the case of maps, would return the value associated to an element inside the list. This is why the module which contains the different commands over globally unique lists, is also parametrized by g. As a side note, it is also parametrized by the decidability of the equality between the elements of `A` (the type of the keys) which means that the keys have to be comparable.

3.4.1 Commands

3.4.1.a Creation

We define a command which creates a new empty Globally Unique list. It uses the property of global uniqueness of the empty list. It shows how creating a globally unique list is stricly more than creating a list: it embeds the proof of correctness which is made possible through the use of dependent types. Note that, while this command is defined in this specific case, it can be defined the same way regardless of the underlying relation. However, it provides no real use so it is left aside.

```
180  newGUL : GUList                                                180
181  newGUL = gulist [] gu[]                                        181
```

This creation command is useful because it will allow us to populate the list with commands that preserve the globally unique property. However, sometimes it can be useful to create such a list from a list that is not necessarily globally unique, but might be. In this case, it is mandatory to be able to decide whether or not it is. As stated when describing the `GloballyUnique` predicate, it cannot be proved decidable in the general case but in the current case it is possible, and we will explain why after the proof is depicted:

```
182  decGU : Decidable_p GlobalUnicity                                    182
183  decGU [] = yes gu[]                                                  183
184  decGU (x :: l) with dec∈ (dec≡f dec) (f x) l                         184
185  decGU _ | yes fx∈l =                                                 185
186    no (λ guxl → P→A→¬A! refl fx∈l (guxl (here refl)))                 186
187  decGU (x :: l) | no _ with decGU l                                   187
188  decGU _ | no ¬fx∈l | yes gul =                                       188
189    yes ((λ {refl fx∈l → ¬fx∈l fx∈l}) ::_u gul)                        189
190  decGU _ | no _ | no ¬p = no (contraposition guTl ¬p)                 190
```

What makes the decidability provable in this case as opposed to the general one ? The difference lies on line 184 where, by invoking `f x` we are able, from the element `x`, to retrieve the unique element that `x` witnesses. In other words, it is possible because we have access to a computation of all the elements that are witnessed by `x` (in this case there is only one). In the general case, from a given `x`, it is not possible to retrieve such information hence trying and deciding if other members in the tail of the list are witnessed by one of values that `x` witnesses. Should we have a function that, for any input, would give the list of all elements that are witnessed by `x`, this could be proved decidable. However, such a function cannot exist in the general case because there is potentially an infinite number of such elements.

Since in our case, this is decidable, we can try and build a globally unique list from an existing list, using the `Maybe` monad to handle error cases.

```
191  newFrom : List B → Maybe GUList                                      191
192  newFrom l with decGU l                                               192
193  newFrom l | yes p = just (gulist l p)                                193
194  newFrom _ | no _ = nothing                                           194
```

3.4.1.b Insertion

The next command consists in adding an element in a list, provided it is not already a member of said list. As a first step, it is not relevant to check for the membership of said element, because it will come later. Putting an element inside a list consists in appending it at its head, hence this command is just an alias for the _::_ operator.

```
195  put : ∀ b l → List B
196  put = _∷_
```

We provide a trivial property of conservation of membership:

```
197  put∈ : ∀ {x b l} → x ∈ l → x ∈ put b l
198  put∈ = there
```

Since the put function returns a list, we have to ensure that, should the input list be globally unique, and should this input not witness the same value as the one witnessed by the new element, the output list remains globally unique. This is the first proof of preservation of global unicity regarding commands on said globally unique lists. This proof is made quite short using the operator `_∷ᵤ_`. While taking a closer look at this proof, it looks like the proof of global unicity of `l` is not used, which makes no sense. However, the last line of the proof could equally be written `putGUL p gul = (λ refl q → p q) ∷ᵤ gul` which highlights that this proof term was indeed used but was hidden by returning a function instead of a value, which is equivalent. More precisely, the clause `f x y = (F x) y` can equally be written `f x = F x` where `f` is the defined function and `F` a term that uses `x` to produce a function which can then be applied to `y` to provide a correctly typed term. This will be used several times in this manuscript.

```
199  putGUL : ∀ {b l} → f b ∉ l → GlobalUnicity l → GlobalUnicity (put b l)
200  putGUL p _ (here refl) = here! refl p
201  putGUL p = (λ {refl q → p q}) ∷ᵤ_
```

Having a function to append an element in a list and the proof of preservation of global unicity, we can extend the put definition to globally unique lists. Since we want the proof elements to remain hidden from the user, we will return an element of type `Maybe` as well in this case to encapsulate the error case when the value is already witnessed in the list. This requires an additional intermediate step which puts a value into a globally unique list when the value is a member of its content. This extra step could have been avoided here but will be useful when instantiating and using this module from outside.

```
202  put_into_when_ : ∀ b → (l : GUList) → f b ∉ (content l) → GUList
203  put b into (gulist l gul) when p = gulist (put b l) (putGUL p gul)
204  -
205  put_inside_ : B → GUList → Maybe GUList
206  put b inside l with dec∉ (dec≡f dec) (f b) (content l)
207  put b inside l | yes p = just (put b into l when p)
208  put _ inside _ | no _ = nothing
```

There is a legitimate question that can be asked at that point: why bother proving the preservation of global unicity when this predicate here is decidable ? Indeed, a variation of the previous function could be written as follows:

```
209  put_inside'_ : B → GUList → Maybe GUList                    209
210  put_inside'_ b = newFrom ∘ put b ∘ content                   210
```

Indeed, this definition seems shorter and overall simpler, however there are two main reasons why this is not the advocated approach:

- When computing the proof that a list is globally unique or not, we check for each element wether it conflicts or not with another element in the list. Thus, the complexity of such computation is n^2. When using the previous proof, we only have to check such conflicts for the new element, thus the complexity is only n. Since the field `unique` could not be declared irrelevant for technical reasons, it will be computed and such difference matters. It is also a lot more elegant to only compute once what can be computed only once, rather than computing over and over the same proof elements as if nothing had already been computed.
- The approach advocated here is somewhat general when dealing with functions that take dependant records as parameters. When fields of these records are proof elements over data embedded in the records, the functions have to preserve such properties to output an element of the right type. In the general case, there is no guarantee that the predicates used in such records are decidable, which makes the preservation proofs mandatory in such cases, which has to be provided by the function in question.

Having considered this option, this will not be further discussed, and conservation properties will be provided whenever required.

3.4.1.c Assignment

Assigning a value in a list consists in replacing the previous value by a new one. This assumes this element was already witnessed inside the list. The first step towards the assign function for globally unique lists does not assume that the value is only witnessed once. This will be taken into account when extending this definition.

```
211  assign_inside_when_ : ∀ b l (fb∈l : f b ∈ l) → List B        211
212  assign b inside (_ :: l) when here _ = b :: l                 212
213  assign b inside (x :: l) when there fb∈l =                    213
214    x :: (assign b inside l when fb∈l)                          214
```

The next step is similar to what has been done for the `put` command: it consists in proving that assigning a value in a globally unique list keeps it globally unique.

To do so, several lemmas are required which all provide partial proofs that assigning a value in a globally unique list preserves membership properties. These lemmas are presented in the appendices in Section A.1.5.

The resulting property of preservation is as follows:

```
215 assignGUL : ∀ {b l p} → GlobalUnicity l                                215
216   → GlobalUnicity (assign b inside l when p)                           216
217 assignGUL gul = assign∈! ∘ gul ∘ ∈assign                               217
```

This leads to the final definition of assignment, which returns `nothing` when the value that is to be assigned did not already have a witness inside the list, and thus there is noting to assign.

```
218 assign_inside_if_ : ∀ b l (fb∈l : f b ∈ content l) → GUList            218
219 assign b inside gulist l gul if p = gulist                             219
220   (assign b inside l when p)                                           220
221   (assignGUL gul)                                                      221
222 -                                                                      222
223 assign_inside_ : ∀ b (gul : GUList) → Maybe GUList                     223
224 assign b inside l with dec∈ (dec≡f dec) (f b) (content l)             224
225 assign b inside l | yes p = just (assign b inside l if p)             225
226 assign _ inside _ | no _ = nothing                                     226
```

3.4.1.d Deletion

The last command consists in removing an element inside a list. This command assumes that this element indeed is witnessed in the list. As usual, we start by writing a function which takes such a proof as input, and we use this proof to remove the right element in the list:

```
227 remove_from_when_ : ∀ a l (p : a ∈ l) → List B                         227
228 remove a from (b :: l) when here px = l                                228
229 remove a from (b :: l) when there a∈l = b :: remove a from l when a∈l  229
```

Then, we provide a lemma which states that removing an element from a list preserves the non membership.

```
230 ∉remove : ∀ {x a l p} → x ∉ l → x ∉ (remove a from l when p)           230
231 ∉remove {p = here _} = contraposition there                            231
232 ∉remove {p = there _} x∉l (here px) = x∉l (here px)                    232
233 ∉remove {p = there p} x∉l (there x∈rm) =                               233
234   ∉remove {p = p} (contraposition there x∉l) x∈rm                      234
```

This leads to the proof that removal preserves global unicity.

```
235 removeGUL : ∀ {a l p} → GlobalUnicity l                                    235
236   → GlobalUnicity (remove a from l when p)                                  236
237 removeGUL {p = here _} = guTl                                               237
238 removeGUL {p = there _} gul =                                               238
239   (λ x≡fb → ∉remove λ x∈xs →                                                239
240     P→A→¬A! x≡fb x∈xs (gul (here x≡fb)))                                    240
241       ::u removeGUL (guTl gul)                                              241
```

This leads to the final definition of removing an element in a globally unique list while hiding the proof computation from the user:

```
242 remove_from_ : ∀ a (gul : GUList) → Maybe GUList                            242
243 remove a from gulist l _ with dec∈ (dec≡f dec) a l                          243
244 remove a from gulist l gul | yes p =                                        244
245   just (gulist (remove a from l when p) (removeGUL gul))                    245
246 remove _ from _ | no _ = nothing                                            246
```

3.4.1.e Assignement or insertion

As an additional command, it is possible to use the precedent definition to either assign the value when present or put the value when absent. This is easily done using the following definition, which directly returns a `GUList` instead of a `Maybe GUList` because it can never fail.

```
247 assignOrPut_inside_ : ∀ b (gul : GUList) → GUList                           247
248 assignOrPut b inside gulist l _ with dec∈ (dec≡f dec) (f b) l               248
249 assignOrPut b inside gulist l gul | yes p =                                 249
250   gulist (assign b inside l when p) (assignGUL gul)                         250
251 assignOrPut b inside gulist l gul | no ¬p =                                 251
252   gulist (put b l) (putGUL ¬p gul)                                          252
```

3.4.1.f Multiple commands

While it is possible to chain several calls of the previous commands using the `Maybe` monad for instance, we provide higher level commands to directly iterate over a list of inputs. This consists of chaining the calls of a given command over this list of inputs. In that purpose, we provide a high-level function, `collapse`, which takes a function of type `X → GUList → Maybe GUList` for any `X` and provides a function of type `List X → GUList → Maybe GUList`. This function uses `foldl` but could definitely use `foldr` in which case the elements in the list would be treated in a different order, thus possibly changing the result. It also uses the monadic bind and return operators of `Maybe` inside the fold.

```
253 collapse : ∀ {u} {X : Set u} → (X → GUList → Maybe GUList)           253
254    → List X → GUList → Maybe GUList                                      254
255 collapse g l gul = foldl (λ mgu → (mgu »=_) ∘ g) (return gul) l        255
```

From this function, we can deduce a variant for the following operators:

```
256 putAll : List B → GUList → Maybe GUList                                 256
257 putAll = collapse put_inside_                                            257
258 -                                                                        258
259 removeAll : List A → GUList → Maybe GUList                              259
260 removeAll = collapse remove_from_                                        260
261 -                                                                        261
262 assignAll : List B → GUList → Maybe GUList                              262
263 assignAll = collapse assign_inside_                                      263
264 -                                                                        264
265 assignOrPutAll : List B → GUList → Maybe GUList                         265
266 assignOrPutAll = collapse λ x → just ∘ (assignOrPut x inside_)          266
```

3.4.2 Queries

In addition to commands, we provide queries on globally unique lists.

3.4.2.a Retrieving

The first query consists of retrieving a value for a specific key. The first step in the creation of the `get` command consists of defining a function which, given a list and a proof of membership inside this list, returns the valuation of g for this specific element. Note that, to retrieve this element, we use the structure of the proof which induces a specific structure of the list (it cannot be empty because it contains an element), which is deduced and used by AGDA, and made visible by the lack of a case for the constructor `[]`.

```
267 get_from_when_ : ∀ a l → a ∈ l → C                                       267
268 get _ from (b :: _) when here _ = g b                                    268
269 get a from (_ :: l) when there p = get a from l when p                   269
```

We also provide a variant on globally unique lists:

```
270 get_from_if_ : ∀ a l → a ∈ (content l) → C                              270
271 get a from l if p = get a from (content l) when p                       271
```

Then, we rely once more on the decidability of the membership relation to build the proof and hide its usage to the user through the use of the `Maybe` type. Note

that, instead of this type, we could use a more verbose approach, as for instance the Either monad which would provide more information in the form of a String of an enumerated type instead of just nothing that would contain information about why the call failed. However, in our case, this information is straightforward (the element was not in the list). In more complex case, this should be done, as advocated in the perspectives in Section 8.2.

```
272  get_from_ : ∀ a l → Maybe C                                  272
273  get a from l with dec∈ (dec≡f dec) a l                      273
274  get a from l | yes p = just (get a from l when p)           274
275  get _ from _ | no ¬p = nothing                              275
```

Finally, we can extend this definition to GULists in order to retrieve values from their content. Note that, in this case, the proof of conformity is not used in the computation, and, since this command does not produce an element of type GUList, no such proof has to be verified for this output.

```
276  get : ∀ a gul → Maybe C                                      276
277  get a = get a from_ ∘ content                                277
```

3.4.2.b Containment

It is important to be able to ask a globally unique list whether or not an element is contained in it. In that purpose, we provide a nothing of membership extended to globally unique lists, as follows:

```
278  _∈l_ : REL A GUList _                                        278
279  _∈l_ x = x ∈_ ∘ content                                      279
```

We then prove it decidable, which is proved as a simple functional composition since _∈_ is itself decidable.

```
280  dec∈l : Decidabler _∈l_                                      280
281  dec∈l x (gulist _ _) = dec∈ (dec≡f dec) x _                  281
```

Then we provide a notion of containment which uses the does operator to erase the proof of membership and produce a boolean[1].

```
282  contains : GUList → A → Bool                                 282
283  contains gul x = does (dec∈l x gul)                          283
```

[1]In the most recent version of the standard library, this is not really an erasure since the boolean is naturally contained inside the definition of the decidability

3.4.2.c Elements

It is useful to be able to retrieve the content of a globally unique list, modulo a function applied to its values.

```
284  private                                                    284
285    elements : ∀ {x} {X : Set x} → (B → X) → GUList → List X  285
286    elements f = (mapl f) ∘ content                           286
```

This is particularly relevant when considering the functions `f`, `g` and `id`.

```
287  elementsF : GUList → List A      287
288  elementsF = elements f           288
289  -                                289
290  elementsG : GUList → List C      290
291  elementsG = elements g           291
292  -                                292
293  elementsId : GUList → List B     293
294  elementsId = elements id         294
```

3.4.3 Examples

Here are two examples of globally unique lists over different parameters. The lists are built and populated using the operators defined in the precedent section. In every case, we display AGDA's evaluation using comments.

Example 1 `GUList` on equality over strings

```
295  module Example1 where                                           295
296    open Commands id Data.String._≟_ id                           296
297    open FunctionRelation {A = String} id                          297
298    open GlobalUnicity _≡f_                                       298
299    open RawMonad {f = lzero} monad                               299
300    -                                                             300
301    ex₁ : Maybe GUList                                            301
302    ex₁ = return newGUL                                           302
303      >>= putAll (⟦ "Judith" ↪ "Bob" ↪ "Alice" ⟧)                 303
304    - content ex₁ = ⟦ "Judith" ↪ "Bob" ↪ "Alice" ⟧                304
305    -                                                             305
306    ex₂ : Maybe GUList                                            306
307    ex₂ = ex₁ >>= putAll (⟦ "Judith" ↪ "Hector" ⟧)                307
308    - ex₂ = nothing                                               308
```

Example 2 `GUList` on equality over the first element of pairs of naturals

```
309  module Example2 where                                                      309
310    open Commands {B = ℕ × ℕ} (proj₁) (Data.Nat._≟_) (proj₂)                310
311    open FunctionRelation {B = ℕ × ℕ} proj₁                                  311
312    open GlobalUnicity _≡f_                                                  312
313    open RawMonad {f = lzero} monad                                          313
314    -                                                                        314
315    ex₁ : Maybe GUList                                                       315
316    ex₁ = return newGUL                                                      316
317      »= putAll (⟦ (3 , 4) ↪ (4 , 5) ↪ (5 , 5) ⟧)                            317
318      »= assignOrPutAll (⟦ (4 , 7) ↪ (3 , 12) ↪ (3 , 10) ↪ (5 , 1) ⟧)        318
319    - content ex₁ = ⟦ (5 , 1) ↪ (4 , 7) ↪ (3 , 10) ⟧                         319
320    -                                                                        320
321    ex₂ : Maybe GUList                                                       321
322    ex₂ = ex₁ »= assignAll (⟦ (3 , 12) ↪ (4 , 10) ↪ (5 , 1) ⟧)               322
323    - content ex₂ = ⟦ (5 , 1) ↪ (4 , 10) ↪ (3 , 12) ⟧                        323
324    -                                                                        324
325    ex₃ : Maybe GUList                                                       325
326    ex₃ = ex₂ »= putAll (⟦ (12 , 4) ↪ (20 , 5) ↪ (3 , 12) ⟧)                 326
327    - ex₃ = nothing                                                          327
328    -                                                                        328
329    ex₄ : Maybe GUList                                                       329
330    ex₄ = ex₂ »= assignAll (⟦ (3 , 6) ↪ (7 , 8) ↪ (5 , 1) ⟧)                 330
331    - ex₄ = nothing                                                          331
332    -                                                                        332
333    val₁ : Maybe ℕ                                                           333
334    val₁ = ex₄ »= get 3                                                      334
335    - val₁ = nothing                                                         335
336    -                                                                        336
337    val₂ : Maybe ℕ                                                           337
338    val₂ = ex₂ »= get 3                                                      338
339    - val₂ = just 12                                                         339
340    -                                                                        340
341    val₃ : Maybe ℕ                                                           341
342    val₃ = ex₂ »= get 6                                                      342
343    - val₃ = nothing                                                         343
344    -                                                                        344
345    el₁ : Maybe (List ℕ)                                                     345
346    el₁ = ex₂ »= just ∘ elementsG                                            346
347    - el₁ = just (⟦ 1 ↪ 10 ↪ 12 ⟧)                                           347
```

3.4.4 Comparison

It is possible to compare globally unique lists together. We provide two ways of doing so : we compare them directly towards their content, or if they always return the same value when calling the `get` query. These comparisons are proved equivalent when the function `x ↦ (f x , g x)` is injective. This developement is described in the appendices in Section A.1.6.

3.5 Two relevant instances of globally unique lists

The two examples presented in Section 3.4.3 have not been chosen randomly. They correspond to two specific kinds of globally unique lists.

Bags correspond to the mathematical sets, which could not be called this way because `Set` is already an AGDA keyword. Bags are defined as follows:

```
348 module ListUnique {a} (A : Set a) (decA : Decidable {A = A} _≡_) where 348
349   open FunctionRelation {B = A} id                                      349
350   open GlobalUnicity _≡f_ renaming (GUList to Bag) public               350
351   open Commands {B = A} id decA id                                      351
352     hiding (elementsF ; elementsG)                                      352
353     renaming (elementsId to elements ; newGUL to newBag) public         353
```

We use the keyword `public` to directly export what has been imported but we change some names and elements of such imports: we rename the globally unique lists to `Bag` and we hide primitives which are irrelevant when considering bags.

Maps are the usual associative lists. The maps are defined as follows:

```
354 module ListAssoc {a} (A : Set a)                                        354
355   (decA : Decidableᵣ {A = A} _≡_) {b} {B : Set b} where                 355
356   open FunctionRelation {B = A × B} proj₁                               356
357   open GlobalUnicity _≡f_ renaming (GUList to Map) public               357
358   open Commands {B = A × B} proj₁ decA proj₂ renaming                   358
359     (elementsF to keys ; elementsG to values ; newGUL to newMap) public 359
```

Once again, we use the keyword `public` to export the right elements. This time we rename three primitives to better match the common names used around maps: `keys` which gives the set of keys in the maps, and `values` which gives back the list of values contained in the map and `newMap` which builds a new empty map.

Another interesting thing to note is the use of braces (implicit argument) for the second type `B`. This type cannot be inferred using the other parameters of the module, which means it will have to be manually instantiated when using the module. So, why make it implicit ? This is because further use of functions exported

from this module will be able to infer this argument with the knowledge of the first instantiation, which will make all the work on petrinets, for instance, a lot clearer and less verbose, which is particularly convenient.

Assessments

This chapter presented a library that was developed using AGDA, which provides the notion of globally unique lists. Globally unique lists are abstract lists parametrized with a binary relation, that can be instantiated into associative lists (maps) or sets (bags). This library has been built using a specific methodology that is used throughout this work. This consists in providing conformity properties to each definition that is made, in order to assess the correctness of said definitions toward their informal specification.

This library introduces several queries and commands that can be used on globally unique lists. These functions have preconditions which have to be satisfied by their input. These preconditions are proven decidable, which means that, given a specific input, it is possible to compute the proof that said input meets the preconditions, or doesn't. This is the occasion to introduce the use of the `Maybe` monad to handle error cases and to hide the proofs that are built from said decidability from the user. Should the input satisfy the requirements, then the function returns a value in the form of `just value`. Otherwise, the function returns `nothing`. This approach will be reused in Chapter 4.

A discussion has been made to better understand what benefits these globally unique lists can provide, by instantiating the relation on which they depend by different kinds of relations. Giving a total order (strict or not) as a parameter does not give any interesting result, and globally unique lists in this case are either lists with at most 1 element or lists which contains an arbitrary number of the same element, which is not fruitful. Instantiating the relation with a partial order is more interesting, because globally unique lists in this case contain elements that comes from separated chains of the partial order. The most interesting relations, however, are equivalence relation, where the lists contain at most a single element per equivalence classes bound to this relation. Both maps and bags come from this category of globally unique lists.

Chapter 4

Toward the mechanization of event-based systems in AGDA

Outline

Since our work relies on trace semantics to express both models and languages behaviour in an heterogeneous context, we first decided to work on event-based transition systems, which are systems with an internal state that can evolve through the execution of different events. These systems naturally induce traces which are possible successions of states that can exist due to this process. Our goal is to exhibit a generic methodology for the modelling of such systems. This chapter presents this approach using the following outline:

1. Section 4.1 presents the objectives of this approach and places them in the context of model engineering, after which the different steps that this approach contains are described: the structural representation of the states, the relational representation of the transitions and the constrained evolution of the system.

2. Section 4.2 describes a commonly used transition system: the PETRI NET. The semantics of the nets is described along with the application of our methodology to mechanize this language and allow its temporal evolution through the execution of transitions which have been proven executable.

3. Section 4.3 describes another application of our methodology, on a language more domain specific than PETRI NET: SIMPLEPDL. We explain how internal states were modelled and how this state can evolve through a safe execution of events. This section is followed by assessments on the lessons that were learnt through the modelling of these two languages.

4.1 Presentation of the approach

This section presents the advocated approach to model event-based systems in proof assistants. As stated in the introduction of this chapter, there are three steps in this methodology, that are, respectively: representing the states in a structural manner, representing the possible transitions in a relational manner and finally allowing a constrained evolution of the systems that is conforming to the precedent representations. Each of these steps captures an important aspect in the representation of event-based transition systems. Before detailing these steps, we give an overview of the goal of this approach.

4.1.1 Objective of the approach

Our work is placed in the context proposed by the GEMOC project, in which languages can be defined and split into their behavioural and structural aspects. In this context, it is important to give a semantics to the languages that are defined inside the framework. In GEMOC, the elementary actions are expressed at a language level by a transformation of the current state of the model. While the global semantics of the behavioural part of the system is expressed using CCSL, which will be tackled in Chapters 6 and 7, the step by step evolution of the system is tackled in the current chapter using event-based systems. We explore a methodology to encode models in AGDA, and provide a semantics to the elementary actions that can be executed to change this internal state. The ultimate goal would be to provide an abstract context in which any such language that is defined using the MOF standard (more on this standard is explained in Chapter 6) coupled with OCL constraints [127] could be modelled. OCL allows us to provide additional constraints on a model which cannot be modelled directly with MOF. While this ultimate goal is not yet reached, we provide both the methodology and two concrete implementations that are important steps towards its completion.

4.1.2 A structural representation of the states

A transition system consists of an internal state which can evolve through the execution of transitions. In order to model this evolution, it is mandatory to model the state of such systems. Transition systems are often represented as graphs (various types of graphs) and these are somewhat challenging to represent in a functional manner, because one cannot create two vertices of the graph then bind them through edges as if in an imperative manner using pointers. To overcome this limitation, we give names to the elements of the graphs and we use these names to emulate the binding between them. The graph is hence represented as a set of maps binding its elements together rather than a real graph as it would be displayed. For that purpose, we use the library on globally unique lists which was presented in Chapter 3. Note that, recently, maps were introduced in AGDA standard library, with an implementation using AVLs (binary search trees) [141]. Should this implementation

have been present in this standard library when we did this work, we would have used it. However, it was not the case and we did our own maps as a preliminary work, as depicted in Chapter 3. Regardless of the nature of the maps, it is convenient to be able to use them to represent a given model, which is done over our two target languages.

4.1.3 A relational representation of the transitions

The second step consists in defining the possible events that can occur on a given system, after which the approach consists in assessing the requirements that need to be fulfilled by a certain state in order for the system to evolve through one of these events. These requirements must be modelled in the form of predicates over the states of the system. In transition systems, the transitions are often guarded which has to be taken into account when formulating such predicates. If there are structural restrictions on the graph, these also have to be taken into account in the process, because one should not be able to create an ill-formed model, as well as making it evolve through paths it cannot take. In other words, this step consists in expressing the preconditions that must be satisfied over the various transitions that can be executed to modify the state of the system. The transitions here are said to be represented relationally, as opposed to operationally, because at this step they are not considered as functions but rather as relations that must be satisfied by the various constituents of the systems. Using these relations, it is possible to define a notion of liveness to such systems: the system is live (as opposed to dead-locked) when at least one event can be executed. The next step proposes a concrete way of executing events, provided the system is not dead.

4.1.4 A constrained evolution of the system

Once the predicates that formulate whether or not a given transition can be executed have been modelled, one needs to be able to exploit them in order to actually make the system evolve in that direction when required by the user. At a certain point of time, several transitions can maybe be executed in which case one of them has to be chosen. In this section, we don't take into account the overall required behaviour of the system (whether or not a certain succession of transitions is acceptable) but this issue will be tackled in Section 6.1. However, we allow an evolution of the state of a system when the requested transition has indeed been proven valid in regards to the current state of the system. This statement somehow implies that the properties that have to be valid for a certain state of the system have to be decidable. When asking the system to evolve in a certain way, the user should not have to provide the proof it can indeed be done, but rather the system should automatically build this proof when possible and, when it is not, provide the related proof as well. The decidability of the predicates we use in this part is very important in that regard. Thus, this step proceeds in the following manner: the user asks a certain transition to be executed, then the system computes the proof that either

it can actually be fired or not. If not, an error is raised, and if it is indeed possible, the proof that has been computed is used to actually create a new state that results in the execution of the transition. This step uses monads to handle errors in this process and to allow the user to chain its requests of transitions to execute. This completes the modelling of a step by step execution of transition systems. This approach has been developed and validated over two languages. The first one is the common PETRI NET formalism [132] and the second one is SIMPLEPDL [45], an executable language used to assess many of our works, for instance [39]. These languages correspond to a simple and well-known case study of language transformation that is often taught to students in their engineer curriculum [40], since these two languages are in fact in a relation of weak-bisimulation [41].

4.2 Application to PETRI NET

4.2.1 Presentation of the PETRI NET language

4.2.1.a A quick definition

Structural aspects A PETRI NET is a bipartite oriented graph used to represent concurrent executable models that was first introduced in 1962 by Carl Adam Petri in his PhD thesis [132]. The nodes of these graphs can be of two different natures, places or transitions – hence the name bipartite – while the edges only embed a single nature, the arcs, that are given a weight. The edges always connect a place to a transition, which means they cannot connect two nodes of the same nature. The graph is usually coupled with a marking, which allows us to track the resources contained in the net, by associating each place with a number of tokens. The nature of the tokens is ambivalent: they can be considered as part of the net itself or as a means of depicting its temporal evolution. This distinction is hard to make and will be discussed as well as detailed in Section 4.2.2. The different constitutive elements of the nets are usually named in order to be able to make reference to them. The following list summarizes the different structural elements that are used to represent a net:

- The places are represented as a circle. They model the different types of resources available in the net. These resources can be of various natures, corresponding to physical or logical resources, depending on the purpose of the user.

- The tokens are represented as black dots. They are located in the places of the net. Each token models one occurrence of the resource represented by the place that contains it. The number of tokens in a place can also be pictured as a natural number instead of the same amount of back dots. Thus, the current marking of the net is represented as a mapping between each place and the positive number of tokens it contains.

- The transitions are represented as rectangles. They model the different actions that can potentially occur during the evolution of the net. Transitions are bound to places by the use of arcs.

- The arcs are represented by arrows going from a place to a transition, or vice-versa. They are annotated with a natural number as weight, which has a value of one by default. They model the resources that are produced or consumed during the execution of the transition they are bound to.

All these elements are summarized in a simple PETRI NET meta-model on Figure 4.1.

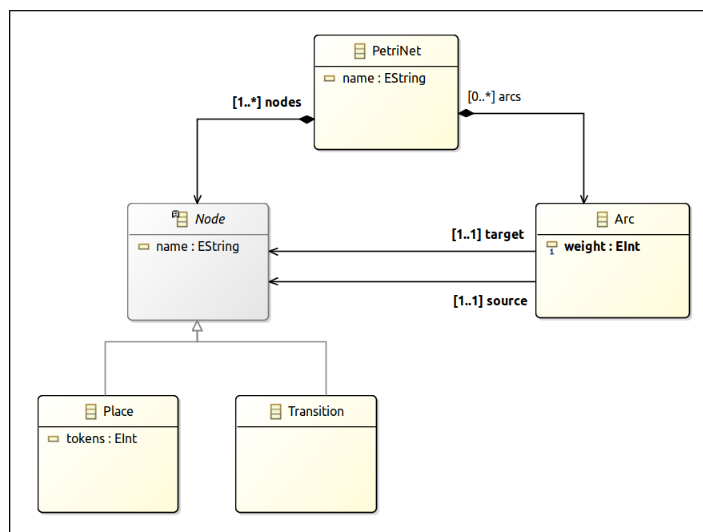


Figure 4.1: PETRI NET meta-model

Behavioural aspects The aspects that are depicted in the previous paragraph and summarized in the meta-model on Figure 4.1 are structural, in the sense that they give no information on the possible evolution of the state of the net, in other words, its behaviour. Rather, they allow the unambiguous definition of such state, that is all. These structural aspects can embed structural constraints, such as the fact that an arc can only bind two nodes of different natures, but do not embed any behavioural constraints. However, they embed the structural elements over which behavioural constraints can be expressed. In order to provide such constraints, one must start by expressing which are the possible events that can occur on a given model.

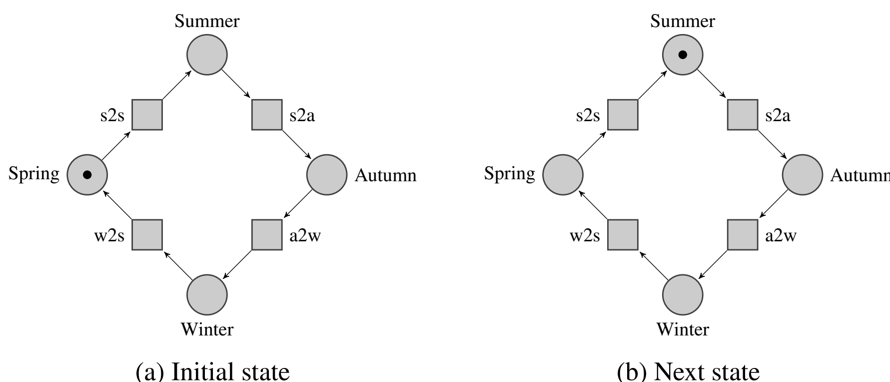
The semantics of PETRI NET tells us that, in a PETRI NET model, we can associate a possible event to each of the transitions that the net contains. Each such event is thus labelled the same way as its corresponding transition. Once such events are expressed, constraints on their execution can be defined. In the case of a PETRI NET, the event associated to the transition `t` can be executed when all its input places, –

the places connected to its consumer arcs – contain at least the number of tokens required by the weight of the arc that binds the two. As a direct consequence of this constraint, a transition that has no consumer arc can be fired in all circumstances.

Once these constraints are expressed, we describe how the system evolves through the execution of an event that can be executed. In the case of PETRI NET, the net evolves as follows: firing a transition – executing its associated event – results in the following changes: each place that is connected to this transition through a consumer arc is emptied of as many tokens as this arc's weight, while each place that is connected to it through a producer arc is filled with that many tokens.

4.2.1.b Examples

Figure 4.2 is an example of a simple PETRI NET representing the changes between seasons, which corresponds to a simple automata. There is initially only one token which depicts the current season, Spring as shown in Figure 4.2a. The transitions can be fired to change one season to another. Since they all consume one token to produce one token, the number of tokens in this net shall not vary – which is fortunate because it is meant to represent seasons, which cannot appear simultaneously or disappear. After the firing of the transition "s2s" (the only one that was fireable in the initial state of the net), the resulting net is depicted in Figure 4.2b.

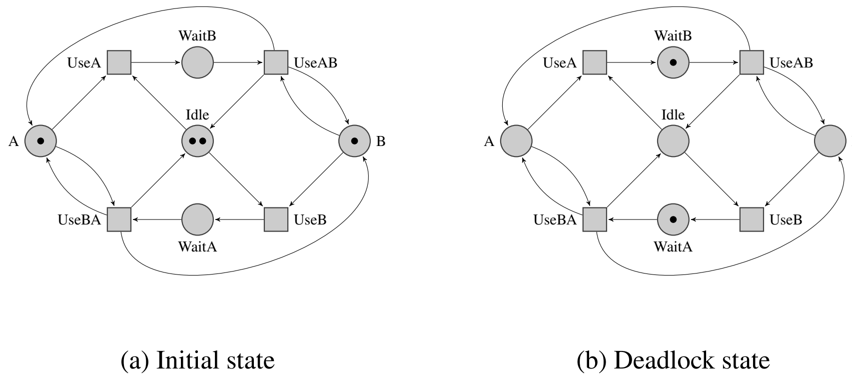


(a) Initial state (b) Next state

Figure 4.2: The seasons PETRI NET

The season net, while being somewhat simple, is not uninteresting. As stated before, it has the property to preserve the number of tokens throughout its execution. It is also deterministic: there are no possible concurrent aspects in the net because at all times there is only one transition that can be fired. More interesting nets are concurrent, and the order of firing of the transitions can have drastic consequences on its future (through the existence of deadlocks for instance). An example featuring a deadlock is depicted in Figure 4.3.

This example features two processes that are initially idle, represented by two tokens in the idle place. Both processes simultaneously need two resources, A and B, to execute their task. Initially, both resources have one instance available, represented by one token in each of these places. One process starts by requesting access



(a) Initial state

(b) Deadlock state

Figure 4.3: The deadlock PETRI NET

to A then B, while the other asks for B then A. The deadlock happens when both processes start their work concurrently by requesting access to their first resource then waiting for the other to be available to pursue their task. Since both wait for the other to free the resource they need, none of them can pursue their work. This net does not necessarily end in a dead lock, when both processes start and finish their action independently. This example emphasizes PETRI NET's ability to model concurrent aspects of system executions and illustrates that PETRI NET can have several executions, some leading to dead locks, others to endless liveness.

4.2.1.c The TINA toolbox

Many tools are available to simulate the temporal behaviour of PETRI NET, one of which is the TINA toolbox [23] that we rely on in our work, both for research and teaching purposes. This tool allows us to edit nets, to import nets, to manually simulate net, to make a structural or a reachability analysis of the net, and so on. An example of TINA's usage – in the stepper simulator – is depicted on Figure 4.4, where the transitions marked in red are the one that can currently be fired.

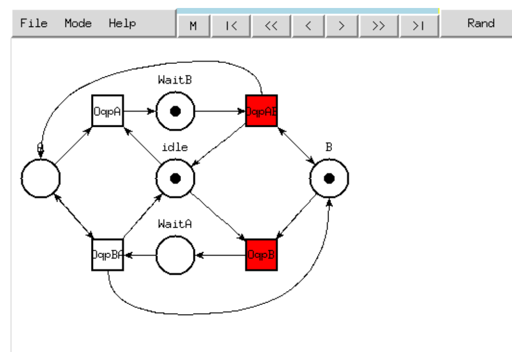


Figure 4.4: A snapshot of TINA's usage

4.2.2 Our approach applied to PETRI NET

4.2.2.a Structural description of the net

Formal definition of PETRI NET We start by creating a module describing PETRI NET. This module will use maps with keys that are instantiated with Strings, which is why the module `ListAssoc` is imported with the first two fixed parameters. One could chose to label the elements of the nets with different types of elements, provided the propositional equality between them is decidable.

```
1  open import ListAssoc String _≟_                                1
```

As for the PETRI NET themselves, we chose to model them as two maps, one representing the places and the associated marking of the net (the number of tokens in each places) and the other representing the transitions and the arcs going from these transitions to the places. The first map, named `marking` contains all the names of the places of the nets, associated to 0 when there are no tokens in them. This means no additional list of places is required. The second map, named `transitions`, contains all the names of the transitions in the net and, for each of these transitions, the number of tokens entering and leaving a given place through the arcs. The nets also have an additional parameter of conformity which ensures that the transitions do not refer to non-existing places. This predicate of conformity corresponds to a structural constraint that is not directly embedded in the other fields of the record. Overall, this record corresponds to the meta-model on Figure 4.1 with the addition of the constraint that an arc necessarily binds a place and a transition.

```
2  record Petrinet : Set where                                     2
3    constructor [m_-_t][_] ; field                                3
4      marking : Map {B = ℕ}                                       4
5      transitions : Map {B = Map {B = ℕ × ℕ}}                     5
6      .conformity : ∀ {t t∈trans p} →                             6
7        p ∈l (get t from transitions if t∈trans) → p ∈l marking   7
```

Creation of an empty net As explained when describing maps, it is required to instantiate the second parameter of the `Map` module, because it cannot be inferred. In our case, these instantiations are ℕ, ℕ × ℕ and `Map {B = ℕ × ℕ}`. All further functions written in this module will take an implicit parameter which will be automatically resolved by AGDA using this initial instantiation. This is the case for the creation of an empty PETRI NET done as follows:

```
8  newPet : Petrinet                                               8
9  newPet = [m newMap - newMap t][ (λ { {_} {()}}) ]               9
```

In this case, the two calls to the function newMap are different, since they accept different implicit parameters, but thankfully AGDA is able to sort this out for us.

Addition of a place We provide a way of adding a place in a correct net:

```
10  +Place : String → ℕ → Petrinet → Maybe Petrinet                 10
11  +Place s _ [m m - _ t][ _ ] with dec∈l s m                      11
12  +Place _ _ _ | yes _ = nothing                                   12
13  +Place s n [m m - t t][ c ] | no ¬p =                            13
14    just [m put s , n into m when ¬p - t t][ put∈ ∘ c ]            14
```

- Line 11 invokes the decidability of the membership of the place in the marking using the with construct.
- Line 12 handles the case where the place is already present, as depicted by yes _, returning nothing.
- Lines 13 and 14 handle the case where the place is not already present by returning a new net with the addition of this place. Since we add a place, the conformity property is preserved easily with the term put∈ ∘ c which is a composition of the property of conformity of the input net and the preservation of membership through the addition of an element inside a list depicted in Section 3.4.1.b.

Addition of transition We provide a way of adding an arcless transition to a net.

```
15  +Trans : String → Petrinet → Maybe Petrinet                      15
16  +Trans s [m _ - t t][ _ ] with dec∈l s t                         16
17  +Trans _ _ | yes _ = nothing                                     17
18  +Trans s [m m - t t][ c ] | no ¬p =                              18
19    just [m m - put s , newMap into t when ¬p t][                  19
20      (λ { {_} {there _} → c}) ]                                   20
```

- Line 16 invokes the decidability of the membership of the transition inside the transitions of the net using the with construct.
- Line 17 handles the case where the transition is already present by returning nothing, as for the place.
- Lines 18 to 20 handles the case where the transition is not already present by returning a new net with the addition of this transition. In this case, the preservation of the conformity property is a simple anonymous function. Indeed, since the transition is empty, it does not point to any place which ensures that the conformity is preserved.

Addition of an arc We finally provide a primitive to add an arc to a transition.

```
21  +Arc : String → String → ℕ → ℕ → Petrinet → Maybe Petrinet                21
22  +Arc st _ _ _ [m _ - t t][ _ ] with dec∈l st t                             22
23  +Arc _ _ _ _ _ _ | no _ = nothing                                          23
24  +Arc _ sp _ _ [m m - _ t][ _ ] | yes _ with dec∈l sp m                     24
25  +Arc _ _ _ _ _ _ | yes _ | no _ = nothing                                  25
26  +Arc st sp _ _ [m _ - t t][ _ ] | yes p | yes _                            26
27    with dec∈l sp (get st from t if p)                                       27
28  +Arc _ _ _ _ _ _ | yes _ | yes _ | yes _ = nothing                         28
29  +Arc st sp -n +n [m m - t t][ c ] | yes p | yes q | no ¬p =                29
30    just [m m - assign st ,                                                  30
31      (put sp , -n , +n into get st from t if p when ¬p) inside t if p t][   31
32      prop2 {t} {m} (putpres {get st from t if p} {m} {¬k∈m1 = ¬p} q c) c ]  32
```

- Line 22 invokes the decidability of the membership of the transition in the transitions.
- Line 23 handles the absence of the transition returning `nothing`.
- Line 24 invokes the decidability of the membership of the place in the marking.
- Line 25 handles the absence of the place returning `nothing`.
- Line 26 and 27 invokes the decidability of the membership of the place in the places that are already linked to the transition.
- Line 28 handles the case where this place is already bound to this transition by returning `nothing`.
- Line 29 handles the only favourable case: the place and the transition are in the net, and not yet bound. In this case, the arc is added by retrieving the arcs bound to the transition, adding the arc inside it and reassigning the resulting arcs in the transition. In this process, the property of conformity must be preserved. This preservation is proved in the appendices along with lemmas in Section A.1.7 and A.2.1.

Correctness preservation All these primitives preserve the properties of correctness, which allows us to create step by step nets which are correct at all points during their construction. If we were using pointers, such permanent correctness could not be satisfied, but by using names it is possible. In the case of PETRI NET, the user should start by creating an empty net, adding all the places, then all the transitions and finally the arcs to ensure the correctness of its construction, as shown in the following examples.

4.2.2.b Interface with TINA

We provide functions to translate our nets into files which can then be read by TINA. These functions are only depicted in the appendices in Section A.2.2.

To illustrate them, we provide two examples of nets which have been exported to TINA. They are created using the `Maybe` monad to chain the addition of the elements of the net, starting by the places, the transitions and then the arcs in order for the net to be correct at all times during its construction.

The first example is the net depicting the changes of seasons, which was represented in Figure 4.2.

```
33 seasons : Maybe Petrinet
34 seasons = return newPet
35   >>= +Place "spring" 1
36   >>= +Place "summer" 0
37   >>= +Place "winter" 0
38   >>= +Place "autumn" 0
39   >>= +Trans "s2a"
40   >>= +Trans "a2w"
41   >>= +Trans "w2s"
42   >>= +Trans "s2s"
43   >>= +Arc "s2a" "summer" 0 1
44   >>= +Arc "s2a" "autumn" 1 0
45   >>= +Arc "a2w" "autumn" 0 1
46   >>= +Arc "a2w" "winter" 1 0
47   >>= +Arc "w2s" "winter" 0 1
48   >>= +Arc "w2s" "spring" 1 0
49   >>= +Arc "s2s" "spring" 0 1
50   >>= +Arc "s2s" "summer" 1 0
```

The second example is the net which exhibits a potential deadlock, and which was depicted in Figure 4.3.

```
51 deadlock : Maybe Petrinet
52 deadlock = return newPet
53   >>= +Place "waitA" 0
54   >>= +Place "waitB" 0
55   >>= +Place "A" 1
56   >>= +Place "B" 1
57   >>= +Place "idle" 2
58   >>= +Trans "UseA"
59   >>= +Trans "UseB"
60   >>= +Trans "UseAB"
61   >>= +Trans "UseBA"
62   >>= +Arc "UseBA" "A" 1 1
63   >>= +Arc "UseBA" "idle" 0 1
64   >>= +Arc "UseBA" "B" 0 1
65   >>= +Arc "UseBA" "waitA" 1 0
66   >>= +Arc "UseB" "waitA" 0 1
67   >>= +Arc "UseB" "idle" 1 0
68   >>= +Arc "UseB" "B" 1 0
69   >>= +Arc "UseAB" "B" 1 1
70   >>= +Arc "UseAB" "idle" 0 1
71   >>= +Arc "UseAB" "A" 0 1
72   >>= +Arc "UseAB" "waitB" 1 0
73   >>= +Arc "UseA" "waitB" 0 1
74   >>= +Arc "UseA" "idle" 1 0
75   >>= +Arc "UseA" "A" 1 0
```

Using our function, we generate a `.net` files for both our examples. These files are correct regarding TINA's syntax, as shown in Figure 4.5, and their graphical representations give the Figures 4.2 and 4.3.

4.2.2.c Behavioural aspects

A net can evolve through the execution of a transition. When such a transition is executed, it is said that it has been fired in the language vocabulary. A transition can be fired if there are enough tokens in all the places from which it consumes tokens. When firing a transition, the tokens that are consumed by the transition are

(a) The seasons net (b) The deadlock net

Figure 4.5: The generated nets

retrieved from the right places while the tokens produced are added to these places. It is possible that a transition adds and retrieves tokens from the same place. If these numbers are equal, this emulates a read arc, which is a special arc which checks if there are enough tokens in a given place without consuming them. This section uses the steps from the methodology described in Section 4.1, as follows:

- Expressing the properties required to fire a transition in a relational manner.
- Proving that these properties are decidable.
- Providing functions which actually fire transitions when it has been decided possible, or fail (using `Maybe`) when it has been decided otherwise.

Predicates of firing We express the fact that a list of arcs is compliant with a given mapping relying on the `All P` predicate which states that all the elements of a list satisfy `P`, and the `Any P` predicate which states that an element of type `Maybe` has the `just` structure and encapsulates a value which satisfies `P`. It is possible when all the places that are linked by the arcs are present in the map and contains enough tokens, which is expressed as follows.

```
76  CanFireArcs : REL Map (List (String × ℕ × ℕ)) _                      76
77  CanFireArcs m =                                                      77
78    All (_⟨ (λ {(a , -n , _) → Any (-n ≤_) ∘ (get a)}) ⟩ m)             78
```

In that context, a transition can be fired in a net when the list of places retrieved from the transition map can be fired inside the places of the net. This definition also takes into account the fact that the transition has to be part of the net, once again using `Any` on the result of `get`.

```
79  CanFireTrans : REL String Petrinet _                                    79
80  CanFireTrans s [m marking - transitions t][ _ ] =                       80
81    Any (CanFireArcs marking ∘ Map.content) (get s transitions)           81
```

The two previous definitions are presented in their most epurate form. A more detailed version is presented in the appendices in Section A.2.4 for the curious reader. Note that these two definitions could have exploited the conformity property of a net, by taking as parameter a proof of membership, but this would lead to more complex definitions with very little upsides, which we chose not to do.

While it is relevant to know if a given transition can be fired inside a net, it can be even more interesting to know if there exists a transition which can be fired without the previous knowledge of this transition's name. This is the concept of liveness: a net is live if it contains a transition which can be fired, as follows:

```
82  live : Pred Petrinet _                                                  82
83  live pet = ∃ (_⟨ CanFireTrans ⟩ pet)                                    83
```

Decidability We give decidability proofs for these predicates to compute the proofs and use them to actually fire a transition in the net if the preconditions to this firing are met. We directly provide the proof of decidability that a transition can be fired in a given net. This proof is built from the decidability of the non-strict precedence over natural numbers and the fact that both `Any` and `All` preserve decidability, which is respectively expressed by `dec` and `all`. Similarly to the expression of the `CanFireTrans` predicate, we provide a more detailed version of this decidability in the appendices in Section A.2.5.

```
84  decFire : Decidable CanFireTrans                                        84
85  decFire _ [m _ - _ t][ _ ] =                                            85
86    dec ((all λ {_ → dec (_≤?_ _) _}) ∘ Map.content) _                    86
```

Proving the decidability of the `live` predicate is much more complicated because of the existential quantification. Although the lemmas that are used for that purpose are put in the appendices in Section A.2.3, one of them is particularly interesting because it gives a way of proving such decidability with a list of possible candidates. The global proof is as follows:

```
87  decLive : Decidable_p live                                              87
88  decLive pet = let t = transitions pet in                                88
89    fromSample (keys t) (flip decFire pet)                                89
90      (_∘ (prop← {m = t}) ∘ getp {m = t})                                 90
```

Firing a transition when decided possible Now that we can compute the proof that a transition can be fired inside a net, we can actually commit this computation in the net, thus creating a new net. The first step consists of creating a new map by applying all the changes of a list of arcs in a map of places.

```
91  fireArcs : ∀ (m : Map {B = ℕ}) {l} → CanFireArcs m l
92    → ∃ λ (m' : Map {B = ℕ}) → ∀ {x} → x ∈l m → x ∈l m'
93  fireArcs m [] = m , id
94  fireArcs m {l = (a , _ , _) ∷ _} (_ ∷ _) with dec∈l a m
95  fireArcs m (just _ ∷ p) | yes _ with fireArcs m p
96  fireArcs m {(a , _ , +n) ∷ _} (just -n≤x ∷ p) | yes q | m' , p' =
97    (assign a , (sub -n≤x) + +n inside m' if p' q) , assign∈ ∘ p'
```

Note that `fireArcs` also returns the proof that the keys of this new map are unchanged through this transformation, since we only modify its values. Returning this proof will be useful to prove the conformity property when creating a new net after firing a transition. This leads to the firing of a transition into a net knowing that it can actually be fired, which is materialized with the proof as parameter. This consists in modifying the places according to the tokens that are consumed and added by the transition, conforming to `fireArcs`.

```
98  fire_inside_when_ : ∀ s pet → CanFireTrans s pet → Petrinet
99  fire s inside pet when _ with dec∈l s (transitions pet)
100 (fire s inside [m m - _ t][ _ ] when just p) | yes _ with fireArcs m p
101 (fire s inside [m m - t t][ conf ] when just p) | yes _ | m' , q =
102   [m m' - t t][ q ∘ conf ]
```

Now it is possible to fire a transition in a net.

```
103 fire : String → Petrinet → Maybe Petrinet
104 fire s pet with decFire s pet
105 fire s pet | yes p = just (fire s inside pet when p)
106 fire _ _ | no _ = nothing
```

We can then change the state of a live net by firing its first fireable transition.

```
107 fireLive : Petrinet → Maybe Petrinet
108 fireLive pet with decLive pet
109 fireLive pet | yes (s , p) = just (fire s inside pet when p)
110 fireLive _ | no _ = nothing
```

Note that this behaviour could be improved by firing a random transition among the fireable ones. This would require to compute the list of all fireable transitions, which is easy to do considering what has already been done, but random behaviour

are not easy to model in proof assistants, and to my knowledge it still has not be done properly in AGDA. However, it could be improperly done by postulating a random function and giving the AGDA compiler rules to compile it in HASKELL.

Example As an example, we can use the `deadlock` net from Figure 4.3 to reach a deadlock state and compute the proof that it is indeed in a deadlock. We use a function which displays "nothing" when the net is malformed, "deadlock" when no transition can be fired, or the name of a transition that can be.

```
111 liveness-state : Maybe Petrinet → String                                111
112 liveness-state nothing = "nothing"                                      112
113 liveness-state (just x) with decLive x                                  113
114 liveness-state (just _) | yes (s , _) = "I can at least fire " ++ s     114
115 liveness-state (just _) | no _ = "deadlock"                             115
```

In this example, we fire different transitions of the `deadlock` net, until we reach a state of deadlock, after which firing a transition returns `nothing`.

```
116 l-s-d : String                                        116
117 l-s-d = liveness-state deadlock                       117
118 - "I can at least fire UseB"                          118
119 -                                                     119
120 deadlock₁ : Maybe Petrinet                            120
121 deadlock₁ = deadlock »= (fire "UseB")                 121
122 -                                                     122
123 l-s-d₁ : String                                       123
124 l-s-d₁ = liveness-state deadlock₁                     124
125 - "I can at least fire UseBA"                         125
126 -                                                     126
127 deadlock₂ : Maybe Petrinet                            127
128 deadlock₂ = deadlock₁ »= (fire "UseA")                128
129 -                                                     129
130 l-s-d₂ : String                                       130
131 l-s-d₂ = liveness-state deadlock₂                     131
132 - "deadlock"                                          132
133 -                                                     133
134 deadlock₃ : Maybe Petrinet                            134
135 deadlock₃ = deadlock₂ »= (fire "UseAB")               135
136 -                                                     136
137 l-s-d₃ : String                                       137
138 l-s-d₃ = liveness-state deadlock₃                     138
139 - "nothing"                                           139
```

4.3 Application to SIMPLEPDL

In Section 4.2, we proposed a first application of our approach described in Section 4.1. This approach aims at modelling event-based systems into AGDA by describing their structural and behavioural aspects. In order to confront this approach with other case studies, we propose a second application on a language which displays some differences with PETRI NET: SIMPLEPDL. These languages are different in various way, the most notable one being that one is more abstract, SIMPLEPDL, while the other is more concrete, PETRI NET, which can be confirmed by an existing transformation going from SIMPLEPDL to PETRI NET. Another difference lies in the kind of events that they propose: PETRI NET proposes a single type of event, which is the firing of a transition, while SIMPLEPDL exhibits two different kinds of events, as shown in the next section. We propose to apply our methodology to SIMPLEPDL to assess how these differences impact our modelling. Since this is the second application of our approach, we give less explanation about their similar aspects to focus on the differences between the two.

4.3.1 Presentation of the SIMPLEPDL language

4.3.1.a A quick definition

Structural aspects SIMPLEPDL is a language to describe processes composed of activities that can be executed in a given order. This language was first introduced in [42] and [45] while its executable version has been presented in [22]. The main notion in SIMPLEPDL is a process. A process is composed of at least one activity (WorkDefinition) and an arbitrary number of constraints that depict how these activities must be ordered with one another (WorkSequence). Each WorkSequence specifies a relation of causality between two activities through a relation of dependence. It states that a given activity can only start – or finish – when another activity has already been started – or finished. It is a convenient way of describing causality between activities inside a global process. A process also contains a set of resources, each of which in a specific quantity. Each WorkDefinition can require the availability of a subset of these resources to be started. The following list summarizes the different elements that are used to represent a process model:

- The WorkDefinitions are represented by ellipses labelled by the name of the activity it represents.
- The WorkSequences are represented by arrows going from the activity which induces this dependence to the activity which is the target of the dependence. These arrows are labelled with the nature of the dependence: start to start (s2s), start to finish (s2f), finish to start (f2s) and finish to finish (f2f).
- The resources are represented as squares labelled with their names. These squares contain a number that represents the number of available resources of this kind in the process similarly to PETRI NET places.

- The need for resources from an activity is depicted as an arrow going from the resource to the activity in question, labelled with the number of required resource from this kind.

The meta-model for this language is depicted in Figure 4.6.

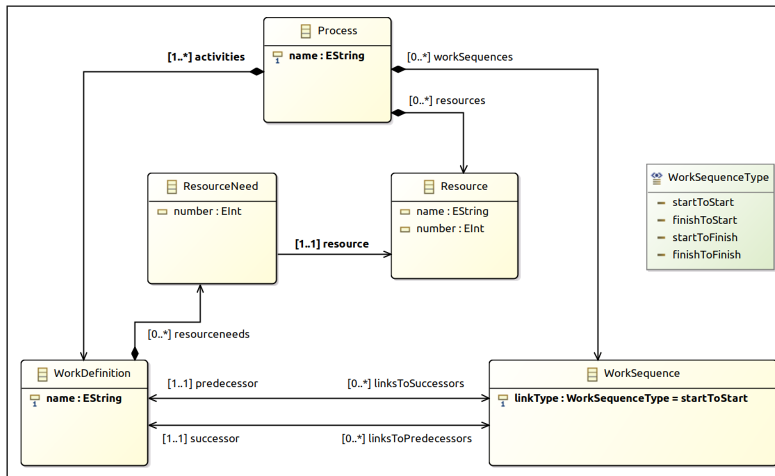


Figure 4.6: SPEM simplified meta-model

From this point on, we ignore resources. These are an important part of a process model but they offer no additional interesting modelling aspect that would not already be tackled in our case studies.

Behavioural aspects We start by identifying the different events that can occur during the execution of a process model. As opposed to PETRI NET, where these events were only of a single nature, the firing of transitions, the events here are of two kinds: starting and finishing an activity. In order to assess the possibility of executing such events for the activities contained in a process, we need to keep track of the state of each activity. Depending on this state, the events can possibly be executed. We consider three different states for the activities of a model: they are either not started, in progress, or finished. For each activity, depending on the dependences for which this activity is the target, and depending on the state of the activity it depends on, the start or finish event can or cannot be executed. The following list summarizes the different requirements bound to the different kind of dependencies to execute an activity `a` that depends on an activity `b`:

- s2s: `a` must not be started and `b` must be in progress or finished.
- s2f: `a` must be in progress and `b` must be in progress or finished.
- f2s: `a` must not be started and `b` must be finished.
- f2f: `a` must be in progress and `b` must be finished.

4.3.1.b Example

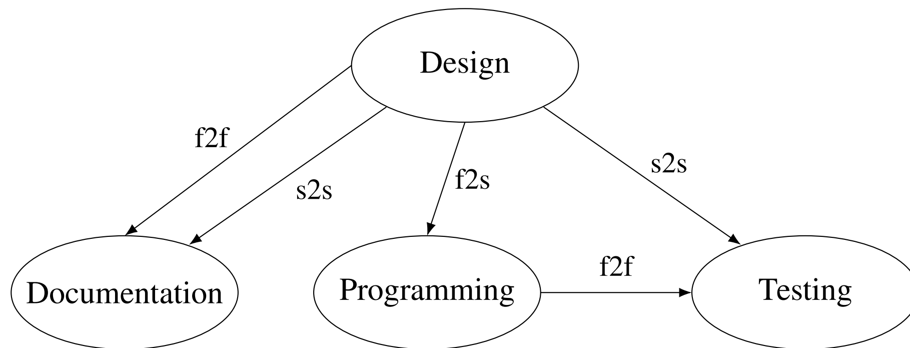


Figure 4.7: The development process

We consider a standard example of a development process depicted in Figure 4.7. This example is composed of 4 activities which are roughly the usual different steps in a program's development, namely the design, the programming, the documentation and the testing. These activities have the following dependencies:

- Documentation can only finish after Design is finished.
- Documentation can only start after Design is started.
- Programming can only start after Design is finished.
- Testing can only start after Design is started.
- Testing can only finish after Programming is finished.

While this example does not contain any resources, it is interesting because from a simple description of a development process and the relations between its constituents, it shows the importance of Design since every other step depends on it one way or another, while it does not depend on anything.

4.3.2 Our approach applied to SIMPLEPDL

As mentioned before, the formal modelling of SIMPLEPDL given in this section does not contain the resources, because we intend to show the differences and similarities in the modelling of the core aspects of PETRI NET and SIMPLEPDL. Adding the resources would not require an extended amount of work, and, most importantly, would not yield any particularly interesting modelling idea which are not already presented in this document.

4.3.2.a Structural aspects of a process model

Workdefinitions A WorkDefinition is composed of a name and a state, which can either be not started, in progress or finished. The WorkDefinitions will be encoded as map from `String` to `WDState`, where `WDState` is one of these possible states:

```
1  data WDState : Set where                      1
2    notStarted : WDState                        2
3    inProgress : WDState                        3
4    finished : WDState                          4
```

Worksequences The definition of WorkSequences depicted in Figure 4.6 makes them look symmetrical. They have a predecessor, a successor and a relation of dependence between the two. However, in a more operational manner, this symmetry needs to be broken down because the successor is the WorkDefinition on which an action needs to be performed while the predecessor remains unchanged when the WorkSequence is executed, hence a fundamental difference between the two. We start by defining the notion of `Action`:

```
5  data Action : Set where                       5
6    start : Action                              6
7    finish : Action                             7
```

Then we define the notion of dependence between WorkDefinitions, depicting the asymmetry between the predecessor of a WorkSequence and its successor:

```
8  data Dependence : Set where                   8
9    toStart : ∀ (a : Action) → Dependence       9
10   toFinish : ∀ (a : Action) → Dependence      10
```

The `Dependence` has the following meaning:

- (`toStart a`) means that, in order for the successor to be able to start, the action a must have been done on the predecessor.
- (`toFinish a`) means that, in order for the successor to be able to finish, the action a must have been done on the predecessor.

Having defined these notions, a WorkSequence is simply represented by the product of a string, a dependence and another string:

```
11 WorkSequence : Set                            11
12 WorkSequence = String × Dependence × String   12
```

In a process model, the WorkSequences will be represented as a `Bag` of WorkSequences which means the equality between them must be decidable, which is established in the appendices in Section A.2.6.

All in all, the bags and the maps can be instantiated with the right parameters to be used inside a process model, while also renaming some of the exported functions because both bags and maps are instances of globally unique list and thus export the same elements which must be differentiated.

```
13  open import ListAssoc String _≟_ {B = WDState}                                   13
14  open import ListUnique WorkSequence dec≡ws using (Bag ; newBag)                 14
15    renaming                                                                       15
16      (_∈ₗ_ to _∈g_ ; dec∈ₗ to dec∈g ; put_into_when_ to putg_into_when_)          16
```

Process models The process models can now be defined. They contain a `Map` of WorkDefinitions, a `Bag` of WorkSequences and three properties of conformity, encoding the following properties: the WorkSequences do not have the same predecessor and successor and both of them correspond to WorkDefinitions that are present in the model.

```
17   record SimplePDL : Set where                                                    17
18     constructor pdl ; field                                                       18
19       wds : Map                                                                   19
20       wss : Bag                                                                   20
21       .conf₁ : ∀ {x} → x ∈g wss → proj₁ x ∈ₗ wds                                  21
22       .conf₂ : ∀ {x} → x ∈g wss → proj₂ (proj₂ x) ∈ₗ wds                          22
23       .conf₃ : ∀ {x} → x ∈g wss → ¬ proj₁ x ≡ (proj₂ (proj₂ x))                   23
```

At this point, a question arises, which is similar to the one introduced and discussed in Section 2.3.4.c when comparing vectors and lists: why not embed the three conformity properties inside the `Bag` of WorkSequences directly ? This is indeed possible, but this makes the type of the `Bag` depend on the field `wds` which is by no means a problem, but which leads to more complicated types and a more complicated definition. This is a common issue when programming with dependent types: where to put the conformity properties ? Shall we put them really deep inside the data structure or rather at the root of the tree ? From my experience, and to my knowledge and understanding, there is no good answer to that question. Either choice makes some proofs easier and others harder with an overall equivalent proof effort in both cases, even though there might be some specific examples where one method should be advocated rather than the other. In this work, we expressed the overall properties of conformity at the root of our data structure. If the reader is interested, Section A.2.7 in the appendices shows an alternative definition of the process model data type when the properties are encoded in the leaves of the tree.

Creation of an empty process model We provide a function to build a process model which is initially empty.

```
24  newPDL : SimplePDL                                         24
25  newPDL = pdl newMap newBag (λ ()) (λ ()) (λ ())            25
```

Addition of a WorkDefinition We provide a function to add a WorkDefinition inside a process model given its name. The newly added WorkDefinition is attributed the state `not started`.

```
26  +wd : String → SimplePDL → Maybe SimplePDL                                26
27  +wd s (pdl wds _ _ _ _) with dec∈_l s wds                                 27
28  +wd _ _ | yes _ = nothing                                                 28
29  +wd s (pdl wds wss c₁ c₂ c₃) | no ¬p = just (pdl (put s ,                 29
30    notStarted into wds when ¬p) wss (put∈ ∘ c₁) (put∈ ∘ c₂) c₃)            30
```

Addition of a WorkSequence Finally, we provide a function to add a worksequence inside a process model, the definition of which is left for the appendices in Section A.2.8, as well as the proof of preservation. This function takes three strings as parameters, the second one corresponds to the kind of dependence, while the two others correspond to the two related WorkDefinitions.

4.3.2.b Behavioural aspects of a process model

Predicates of execution The next step in our methodology consists in expressing the predicates which allow a WorkDefinition to be executed. The first step in that purpose is the left compliance, which means the compliance towards the predecessor of a WorkSequence for a given action. In other words, `Complies← a wds` means that `a` has already been executed by the WorkDefinition whose state is `wds`.

```
31  Complies← : Action → WDState → Set                                31
32  Complies← start wds = wds ≡ inProgress ⊎ wds ≡ finished            32
33  Complies← finish = _≡ finished                                     33
```

The right compliance concerns the successor and checks whether its internal states allows it to execute the given action. In other words `Complies→ a wds` means that `a` is the action which is compatible with `wds`.

```
34  Complies→ : Action → WDState → Set                                 34
35  Complies→ start = _≡ notStarted                                    35
36  Complies→ finish = _≡ inProgress                                   36
```

The next predicate expresses the compatibility between the state of the predecessor, the action to perform and the dependence between the predecessor and the successor. There are four possible cases of compatibility, which can be summed up as follows: either the dependence is not relevant towards the action, or it contains an action which is compliant.

```
37  data Comp (wds : WDState) : Action → Dependence → Set where          37
38    cStart : ∀ {a} → Complies← a wds → Comp wds start (toStart a)      38
39    cFinish : ∀ {a} → Complies← a wds → Comp wds finish (toFinish a)   39
40    cStartFinish : ∀ {a} → Comp wds start (toFinish a)                 40
41    cFinishStart : ∀ {a} → Comp wds finish (toStart a)                 41
```

To assess whether a list of WorkSequences is compliant to the execution of an action for a given WorkDefinition, each of its members must either be irrelevant for this WorkDefinition (which means its successor is different from it) or must be compatible according to the previous predicate.

```
42  CompliesWithList : Action → String → Map → List WorkSequence → Set  42
43  CompliesWithList a s wds =                                           43
44    All λ {(prec , dep , succ) → (¬ succ ≡ s) ⊎                         44
45    Any (λ x → Comp x a dep) (get prec wds)}                           45
```

Ultimately, we can express the fact that a process model is compliant to the execution of a specific action on a specific WorkDefinition. This is possible when this WorkDefinition appears in the process model and when the content of the `Bag` of WorkSequences is compliant with this execution according to the `CompliesWithList` predicate.

```
46  CompliesWith : Action → String → SimplePDL → Set                     46
47  CompliesWith a s (pdl wds wss _ _ _) =                               47
48    Any (Complies→ a) (get s wds) ×                                    48
49      CompliesWithList a s wds (Bag.content wss)                       49
```

Similarly to the Petri Net we can define a notion of liveness of a model of process: it is live whenever it contains a WorkDefinition which can either be started or finished.

```
50  AliveAction : Action → SimplePDL → Set                               50
51  AliveAction a spdl = ∃ (λ s → CompliesWith a s spdl)                 51
52  -                                                                    52
53  Alive : SimplePDL → Set                                              53
54  Alive = AliveAction start ∪ AliveAction finish                       54
```

As a side note, if we consider a process model which is not live, but which only

contains WorkDefinitions that are finished, it is in a "correct" state of deadlock, although it still is in a deadlock state. This shows that the notion of deadlock itself is not sufficient to encapsulate the semantics of a correct execution of an event-based system, which brings us one more time to the problematics tackled in the following chapters.

Decidability According to our methodology we then intend to prove the decidability of our predicates. The decidability of `Complies←`, `Complies→` and `Comp` are trivial and are provided in the appendices in Section A.2.9. As for the decidability of `CompliesWith`, we only give the epurate form, because it follows the same steps and logic as what was explained in detail for the nets in Section A.2.5.

```
55  deccwl : ∀ {a s} → Decidable (CompliesWithList a s)                        55
56  deccwl wds = all λ _ → ¬? (_ ≟ _) ⊎-dec dec (λ _ → deccomp _ _) _           56
57  -                                                                           57
58  deccw : ∀ {a} → Decidable (CompliesWith a)                                  58
59  deccw {a} _ spdl = dec (decatpt a) _ ×-dec deccwl (wds spdl) _              59
```

The predicates of liveness are also decidable, thanks to the same properties that were used to prove the decidability of the liveness of Petri Net depicted in Section A.2.3 which allows us to prove the decidability of an existential quantification using a list of possible candidates.

```
60  decAliveAction : ∀ a → Decidable_p (AliveAction a)                          60
61  decAliveAction a spdl = let w = wds spdl in                                 61
62    fromSample (keys w) (flip deccw spdl)                                     62
63      (_∘ (prop← {m = w}) ∘ (getp {m = w}) ∘ proj₁)                           63
64  -                                                                           64
65  decAlive : Decidable_p Alive                                                65
66  decAlive x = decAliveAction start x ⊎-dec decAliveAction finish x           66
```

Executing an action We give functions which allow us to perform a given action on a given WorkDefinition of a process model. As advocated, these functions compute the proof that the action can be performed or not and act accordingly. First, we perform a change of state in a state compliant with a given action.

```
67  perform_from_when_ : ∀ (a : Action) (wds : WDState) →                      67
68    Complies→ a wds → WDState                                                 68
69  perform start from .notStarted when refl = inProgress                       69
70  perform finish from .inProgress when refl = finished                        70
```

Then we perform the actual execution of a WorkDefinition when we know it

can be done. This consists in extracting the current state of the WorkDefinition and replacing it with its new state which reflects the fact that the action has been performed using the previous definition `perform_from_when_`.

```
71  perform_on_inside_when_ : ∀ a s pdl →
72    CompliesWith a s pdl → SimplePDL
73  perform a on s inside spdl when (fst , snd) with dec∈l s (wds spdl)
74  perform a on s inside pdl wds wss c1 c2 c3 when (just q , snd) | yes p =
75    pdl (assign s ,
76      (perform a from (get s from wds if p) when q) inside wds if p)
77      wss (assign∈ ∘ c1) (assign∈ ∘ c2) c3
```

We use the proof of decidability to perform a given action when possible.

```
78  perform : ∀ a s pdl → Maybe SimplePDL
79  perform a s spdl with deccw {a} s spdl
80  perform _ _ _ | no _ = nothing
81  perform a s spdl | yes p = just (perform a on s inside spdl when p)
82  -
83  start_inside_ : ∀ s pdl → Maybe SimplePDL
84  start_inside_ = perform start
85  -
86  finish_inside_ : ∀ s pdl → Maybe SimplePDL
87  finish_inside_ = perform finish
```

Example As an example, we instantiate in our framework the process model depicted in Figure 4.7. By creating an empty model, then populating it with WorkDefinitions and finally with WorkSequences we guarantee a correct-by-construction creation of our model.

```
88  dev : Maybe SimplePDL
89  dev = return newPDL
90    »= +wd "Documentation"
91    »= +wd "Design"
92    »= +wd "Programming"
93    »= +wd "Testing"
94    »= +ws "Design" "f2f" "Documentation"
95    »= +ws "Design" "s2s" "Documentation"
96    »= +ws "Design" "f2s" "Programming"
97    »= +ws "Design" "s2s" "Testing"
98    »= +ws "Programming" "f2f" "Testing"
```

We define functions which output a list of all the possible WorkDefinitions that can either be started or finished in a given process model.

```
99  listCanPerform : ∀ a spdl → (candidates : List String) → List String        99
100 listCanPerform _ _ [] = []                                                   100
101 listCanPerform a spdl (s :: _) with deccw {a} s spdl                         101
102 listCanPerform a spdl (_ :: l) | no _ = listCanPerform a spdl l              102
103 listCanPerform a spdl (s :: l) | yes p = s :: (listCanPerform a spdl l)      103
104 -                                                                            104
105 listCanStart : ∀ spdl → List String                                          105
106 listCanStart spdl = listCanPerform start spdl (keys (wds spdl))              106
107 -                                                                            107
108 listCanFinish : ∀ spdl → List String                                         108
109 listCanFinish spdl = listCanPerform finish spdl (keys (wds spdl))            109
```

This allows us to execute some steps in our model, which are described using comments.

```
110 - Only the Design can be started                                             110
111 csdev : map_m listCanStart dev ≡ just ("Design" :: [])                       111
112 csdev = refl                                                                 112
113 - We start the Design                                                        113
114 dev1 : _                                                                     114
115 dev1 = dev »= start "Design" inside_                                         115
116 - Now, the Testing and the Documentation can be started                      116
117 csdev1 : map_m listCanStart dev1 ≡                                           117
118   just ("Testing" :: "Documentation" :: [])                                  118
119 csdev1 = refl                                                                119
120 - And the Design can be finished                                             120
121 cfdev1 : map_m listCanFinish dev1 ≡ just ("Design" :: [])                    121
122 cfdev1 = refl                                                                122
123 - We finish the Design                                                       123
124 dev2 : _                                                                     124
125 dev2 = dev1 »= finish "Design" inside_                                       125
126 - The Programming is now available                                           126
127 csdev2 : map_m listCanStart dev2 ≡                                           127
128   just ("Testing" :: "Programming" :: "Documentation" :: [])                 128
129 csdev2 = refl                                                                129
130 - No further activities can be finished at the moment                        130
131 cfdev2 : map_m listCanFinish dev2 ≡ just []                                  131
132 cfdev2 = refl                                                                132
```

Assessments

This chapter presented an approach to model event-based systems in AGDA. This modelling consists of several steps, which have been validated over two target languages: PETRI NET and SIMPLEPDL. Through the modelling of these languages, we can learn some lessons on how to approach such modelling in the general case for languages defined in MOF with the potential addition of OCL constraints.

The description of the internal state of the system can be defined using records containing two kinds of fields: the actual structure of the state, using maps and bags which allow us to emulate the graph structure of the states, and properties of conformity over this structure. These conformity properties can be declared irrelevant in AGDA as described in Section 2.3.5.e. Another possibility would be to encode these properties directly in the leafs of the structure but this would lead to more complicated types which we chose not to do. In both our target languages, the conformity properties were similar: it consists in stating that the edges in the graph do not point to non-existing vertices. They also embed some additional semantics, such as the fact that two elements should not be the same, as it was the case for SIMPLEPDL. Overall, any structural property can be expressed as a conformity property embedded inside the record depicting the state of the system.

The description of the possible events – atomic actions in the sense of GEMOC– which allow the system to evolve is then done relationally: it consists in defining predicates which, when satisfied, ensure that the internal state of the system can evolve according to the semantics of the language. In both our target language, defining these predicates is done in two steps: expressing locally what are the conditions of evolution (enough tokens in a place for instance in PETRI NET) and then incorporating these conditions in the whole data structure, using for instance primitives over maps, bags and lists.

Once the evolution predicates have been defined, the evolution of the system can be made possible with the previous knowledge that said predicate holds. Using the proof of possible evolution, we retrieve the concrete elements which allow the system to evolve and we use them to commit the changes it implies into the state of the system. Both in SIMPLEPDL and PETRI NET, it consists in looking for membership proofs to find the appropriate elements on which a change has to be committed. Since such evolution creates a new state, the properties of conformity have to be preserved through this process. Finally, the predicates of evolution are proved decidable, which allows us to try to execute a given action in all cases without the knowledge that it can be executed. This concludes our methodology on how to handle the modelling of event-based systems in AGDA.

Overall, this methodology embeds both the structural aspects of the languages as well as their step-by-step behavioural aspects, meaning the atomic actions that can possibly be executed for a given state. The third aspects advocated by GEMOC, which consists in reducing the possible traces of the systems (to avoid deadlocks, for instance, or to force synchronicity between actions) though constraints over these traces is tackled in the following chapters.

Chapter 5

Refining instants in asynchronous systems execution

Outline

This chapter treats the issue of instant refinement through the following outline:

1. Section 5.1 introduces basic notions which are used to model logical time in asynchronous systems such as instants, partial orders and time structures. It also introduces the notion of trace-based semantics, which allows us to give meaning to languages through their possible sequences of events during their executions, the so called traces.
2. Section 5.2 introduces the issue of instant refinement in asynchronous systems, with the notion of level of refinement and level of observation. It provides the core contribution of this chapter: the modelling of refinement as an order between partial orders.
3. Section 5.3 gives keys to understand how our notion of refinement was mechanized, and which properties have been deduced from this mechanization. It also emphasises the intuition that comes from these properties.
4. Section 5.4 presents a detailed example of instant refinement on a simple transition system on which a transition is refined using our approach. It consists in all the steps which should be done to treat such case of refinement, and concludes with a formal verification of this example.

5.1 Handling of time in asynchronous systems

Chapter 4 presented an approach on how to formally encode event-based systems, without handling their global behaviour. For instance, Figure 4.3 showed a PETRI NET which can possibly enter a state of deadlock, depending on the transitions we chose to fire. Should we give higher level properties on this execution of this model, we could avoid such cases and force an endless correct behaviour where both processes never take access to their first resource simultaneously. In order to express such properties, we advocate to work with trace-based semantics, which are described in the following section.

5.1.1 Introduction to traced-based semantics

Asynchronous systems There exist many definitions of the synchronous and asynchronous words in computer science, the ones coming from the hardware architecture including clocks that can be global or local, the one coming from concurrent systems, the one coming from the communication models, and even the ones coming from certain kinds of language or usage of languages. In our work, since CPS are related to the real world and the notion of time, we use a definition close to the one from hardware and related to clocks. In other words, asynchronous systems in our work are systems whose behaviours and actions are not subject to an hypothetical predefined global clock providing the set of all possible instants where events can occur. Rather, each action performed by these systems is done in an asynchronous manner, in other words, at an arbitrary point in time. As a consequence, there is no total order on the events that occur in these systems, but only possible partial orders, which will be detailed in Section 5.1.3.

Traces of execution The traces of execution of a given system are all the possible outcomes, represented as sequences of events, that the execution of the system can induce. A specific execution of this system will provide a trace which is a compilation of all that happened during this execution represented as a sequence of events. These traces do not take into account the internal mechanisms of the system, but only the events that have been observed throughout its execution.

Events The events are the constituents of the traces. If there are no observable events, then the traces will be empty and will yield no other information than that no observable event occurred. Usually, several events can be observed which makes the traces of execution relevant in describing the behaviour of the system, by interpreting the precedences between the occurrences of the events tracked by the traces.

Time structure A time structure is a notion derived from event structures introduced by Winskel in [152] and Lamport in [89] who propose to couple a notion of time with a partial order to bind certain events together with a notion of precedence,

while others are related through a notion of coincidence. Such structures are used in our work and depicted more precisely in Section 5.1.3. These structures induce the notion of causality, which is characterized by two instants being in this same thread of the time structure.

Observations The level of observation is fundamental when dealing with such systems because a trace only takes into account observable events . This introduces the notion of refinement, which categorizes certain sets of events as being part of the same level of observation, which is the topic of Section 5.2.

Time It is impossible to talk about traces of execution without mentioning time, which is the implicit notion on which they are based in order to build sequences of events (sequences are strictly ordered structures). Each event occurs on a certain instant, and these instants are ordered in a certain way which is fundamental when specifying the behaviour of a system. The following sections are meant to explain and describe these notions as thoroughly as possible.

5.1.2 Instants

Instants are the main concept on which asynchronous languages are defined. To better understand what these instants stand for, let us take an informal look at them. An instant can be seen as an imaginary point in a timeline where events can occur. It matches, to a certain extent, the common vision one has about time. However, this definition is somewhat misleading and incomplete because there is no good way of defining a timeline without talking about instants. Indeed, a timeline is a sequence of strictly ordered instants, which makes these definitions mutually recursive. Mutual recursivity, however sound it may be in computer science, can hardly be accepted in this case, especially since a sound, well-founded recursivity requires a base case. The reason behind this incapacity at explaining what time is, is that it stands as one of the most complex notion that human beings have encountered and it can hardly be explained or defined without any pre-existing opinion on the matter, which is usually made or given as a child. Time is a basic notion that cannot be defined without invoking one or several notions whose existence is subsequent to time. For instance, one can try to explain it as "what happens when nothing happens" but this definition already embeds the notion of time as "happens" contains it. This view about time and instants tells us that capturing "what time is", is very (if not too) ambitious for the human mind.

However, we can build models of time using notions that are easier to grasp, which is what is usually done in languages that cope with time in one way or another. However little is our comprehension of time, we live with the feeling that it is endless, which is why, when representing time, we tend to use numbers that can grow indefinitely. Natural numbers, rational numbers or real numbers come to mind in that regard, as they all grow indefinitely although they grow in different ways. Natural numbers are used to represent discrete time, with a finite number of instants

between two distinct instants, as rational and real numbers are used to represent dense time, where there always exists another instant between two instants and thus an infinite number of them, either countable (rational) or uncountable (real). This representation of time is very convenient (regardless of the nature of the numbers that are used) because it fits the usual and intuitive macroscopic vision one has of time that instants succeed each other in a single well defined timeline. However, at the beginning of the twentieth century, Albert Einstein showed that such a vision is only true given a specific observer while another observer would have his own vision of synchronicity and relationship between the events that occurred [62]. While the representation of time in asynchronous systems is hardly the same as in Einstein's restrained relativity, said relativity theory teaches us that what looks simple might hide some underlying complexity. It also shows that a single well defined timeline is not always an accurate description of how time should be seen, and this is true in CCSL as well as in our work, even though the distinction we make is different to the one Einstein made a hundred years ago.

Indeed, time in asynchronous systems cannot be seen as a single timeline consisting of instants. This is due to the lack of knowledge one can have regarding the execution of such asynchronous systems, when it is usually impossible to know, for all events and their respective instants, whether one has happened before the other. Another difference with our common perception of time is that several instants can be coincident, which means they "happen" simultaneously. This is the case for instance when two successive events happen so close to each other that they cannot be distinguished by a given observer. In some asynchronous languages, such as CCSL, this vision is completely embraced, since no instant can "host" more than one event. This means that two events that seem to occur simultaneously will still be carried out by different instants, but these instants will be coincident. This vision is closely linked to the notion of refinement proposed in the chapter, because it assumes that there always exists a thinner level of refinement until no coincident instants can be distinguished one from the other any more. This ultimate level of refinement corresponds to the complete knowledge of the system's behaviour.

As explained, while real, natural and rational numbers are all totally ordered, the reality in asynchronous systems is quite different, because these underlying total orders cannot be observed. They might exist – given a specific observer – but cannot necessarily be observed or described. Only partial information can be given as to which event occurred before or after another specific event. To be more specific, the underlying total order between events exists physically, but is hidden to the observers. This sounds like a limitation, but it can also be seen as a flexibility, since it allows us to only express the relations that we are interested in or that must be enforced, which is not possible in total orders. As for the underlying set of instants, as we saw, we should not directly rely on specific numbers, this is why it will remain unspecified in our framework until required. In concrete examples, it will eventually be instantiated but we would rather work with an abstract set that is only characterized by having a partial order. Fortunately, this partial order can easily be expressed using dependent types which we will see in Section 5.1.3.

5.1.3 Strict partial orders

5.1.3.a Definition

In order to model the temporal relations between instants in the execution of asynchronous systems, one cannot use total orders for the reasons depicted in Section 5.1.2. The advocated approach is to use strict partial orders which allows instants to be coincident and some instants to be unrelated, as opposed to total orders. A strict partial order is a mathematical structure composed of four entities, which are the following:

- A set of elements, which is called `Support` in our case
- A first relation over the elements of `Support` written _≈_
- A second relation over the elements of `Support` written _<_
- A proof that these relations satisfy the properties of strict partial ordering.

The last element of this structure is fundamental because it ties all the others together. By giving the right properties to the relations, it also gives them the intended meaning, that is: _<_ is a strict precedence between elements of `Support` and _≈_ is a relation of equivalence between its elements. These properties are as follows:

1. _≈_ is an equivalence relation
   - _≈_ is reflexive $\forall i \in I \, : \, i \approx i$
   - _≈_ is transitive $\forall (i, j, k) \in I^3 \, : \, i \approx j \wedge j \approx k \Rightarrow j \approx k$
   - _≈_ is symmetrical $\forall (i, j) \in I^2 \, : \, i \approx j \Rightarrow j \approx i$
2. _<_ is irreflexive towards _≈_ $\forall (i, j) \in I^2 \, : \, i < j \Rightarrow \neg\, i \approx j$
3. _<_ is transitive $\forall (i, j, k) \in I^3 \, : \, i < j \wedge j < k \Rightarrow j < k$
4. _<_ respects _≈_
   - on the left $\forall (i, j, k) \in I^3 \, : \, i \approx j \wedge i < k \Rightarrow j < k$
   - on the right $\forall (i, j, k) \in I^3 \, : \, i \approx j \wedge k < i \Rightarrow k < j$

5.1.3.b An example of Strict partial order

(a) A first possible behaviour (b) A second possible behaviour

Figure 5.1: Both possible behaviours

Let us consider a person named Alice and her usual morning routine: Alice gets up, after which she either takes a bath first followed by eating or vice versa. She always sings while in the bath. After that, she takes off for work. The two possible traces depicting her behaviour over a single day are shown in Figure 5.1a and 5.1b. They consider the following set of possible events: getting up, bathing, singing, eating and taking off, with their respective aliases "u", "b", "s", "e" and "o".

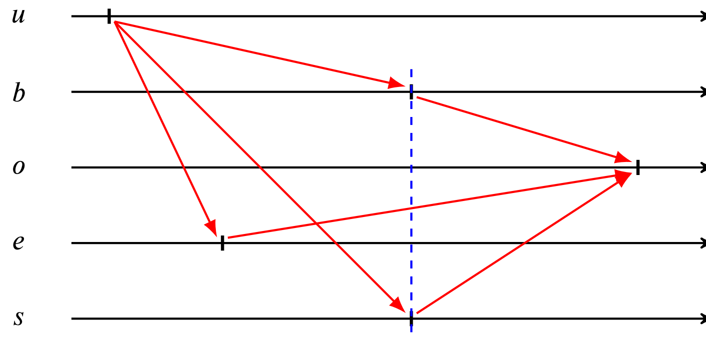


Figure 5.2: The underlying partial order

These possible behaviours are described by a time structure, as described in Section 5.1.1, with an underlying partial order, that is depicted on Figure 5.2. The events "b" and "e" are concurrent and are not linked by any of the two relations composing the strict partial order. The blue vertical dashed line represents coincidence (when events occur simultaneously) while the red arrows represent precedence.

5.1.3.c Strict partial orders in AGDA

The instants in our work are modelled as a strict partial order using the definition that is provided in the AGDA standard library, in the form of a dependent record, as depicted in Figure 5.3[1]. This structure that we called `Instant` as opposed to `Support` which is its underlying set, is passed as a parameter to all our modules that deal with time for future users to instantiate it with the strict partial order of their choice.

```
record StrictPartialOrder c ℓ₁ ℓ₂ : Set (suc (c ⊔ ℓ₁ ⊔ ℓ₂)) where
  infix 4 _≈_ _<_
  field
    Carrier              : Set c
    _≈_                  : Rel Carrier ℓ₁
    _<_                  : Rel Carrier ℓ₂
    isStrictPartialOrder : IsStrictPartialOrder _≈_ _<_

  open IsStrictPartialOrder isStrictPartialOrder public
```

Figure 5.3: The strict partial order dependent record

[1]Note that the AGDA definition contains a type signature with somewhat complex levels of universe that can be ignored by the reader.

The field `IsStrictPartialOrder` is also a dependant record which contains the various properties detailed earlier which enforce the structure of strict partial order.

5.1.3.d Additional relations

In our representation of time, we add three derived relations to this strict partial order, which brings the number of available relations over instants to 5, which means that two instants can be in a relation of:

- **Precedence** according to the precedence relation of the strict partial order,
- **Coincidence** according to the equivalence relation of the strict partial order,
- **Causality** when they are either precedent or coincident,
- **Dependence** when they are causally linked but not coincident,
- **Independent** when they are neither coincident nor dependent.

These relations are defined in AGDA as depicted in Figure 5.4.

```
-- Precedence
_≺_ : Rel Support lzero
_≺_ = _<_ Instant

-- Coincidence
_≈_ : Rel Support lzero
_≈_ = _≋_ Instant

-- Causality
_≼_ : Rel Support lzero
x ≼ y = x ≈ y ⊎ x ≺ y

-- Dependance
_≉_ : Rel Support lzero
x ≉ y = x ≺ y ⊎ y ≺ x

-- Independance
_∥_ : Rel Support lzero
x ∥ y = ¬ x ≉ y × ¬ x ≈ y
```

Figure 5.4: The five relations binding instants together

5.1.3.e Properties

All these relations exhibit certain properties that will be useful for further proving and developing. The properties of the relations of precedence and coincidence (transitivity, equivalence, and so on) are already present in the standard library, but those about the newly defined relations derived from them are yet to be expressed and proved. Figure 5.5 presents a simple example, `trans≼`, stating that the causality relation is transitive. The proof exploits the transitivity of the two underlying relations coupled with the symmetry of the coincidence and an additional property

```
trans≤ : Transitive _≤_
trans≤ (inj₁ i≈j) (inj₁ j≈k) = inj₁ (trans≈ i≈j j≈k)
trans≤ (inj₁ i≈j) (inj₂ j<k) = inj₂ (<-resp-≈₂ (sym≈ i≈j) j<k)
trans≤ (inj₂ i<j) (inj₁ j≈k) = inj₂ (<-resp-≈₁ j≈k i<j)
trans≤ (inj₂ i<j) (inj₂ j<k) = inj₂ (trans< i<j j<k)
```

Figure 5.5: A trivial example of property

stating that the two underlying relations behave correctly towards one another. Another example is depicted in Figure 5.6 where more complex properties are proven, $\|\rightarrow\approx\rightarrow\|_1$ and $\|\rightarrow\approx\rightarrow\|_2$. We state that if two instants are independent, they remain independent through coincidence on both sides thanks to the symmetry of the independence, `sym‖`, and an auxiliary lemma that corresponds to the De Morgan development of the logical negation over the logical conjunction, ¬⊎→×¬.

```
¬⊎→×¬ : ∀ {a} {A B : Set a} → ¬ (A ⊎ B) → ¬ A × ¬ B
¬⊎→×¬ ¬a⊎b = ¬a⊎b ∘ inj₁ , ¬a⊎b ∘ inj₂

‖→≈→‖₁ : ∀ {α β γ} → α ‖ β → β ≈ γ → α ‖ γ
‖→≈→‖₁ {α} {β} (¬α≠β , ¬α≈β) β≈γ with ¬⊎→×¬ ¬α≠β
‖→≈→‖₁ (_ , ¬α≈β) β≈γ | ¬α<β , ¬β<α
  = (λ { (inj₁ α<γ) → ¬α<β (<-resp-≈₁ (sym≈ β≈γ) α<γ) ;
         (inj₂ γ<α) → ¬β<α (<-resp-≈₂ (sym≈ β≈γ) γ<α)})
    , (λ α≈γ → ¬α≈β (trans≈ α≈γ (sym≈ β≈γ)))

‖→≈→‖₂ : ∀ {α β γ} → α ‖ β → α ≈ γ → γ ‖ β
‖→≈→‖₂ α‖β α≈γ = sym‖ (‖→≈→‖₁ (sym‖ α‖β) α≈γ)
```

Figure 5.6: A more complex example of property

These two examples of properties bound to the five relations binding the instants together are a small portion of everything that has been defined and established around instants. However, these are mostly small definitions and properties which are depicted in the appendix in Section A.3.1

5.2 A formal definition of instant refinement

5.2.1 On refining instants

While languages that describe and handle the temporal execution of complex systems – such as CCSL– are actively growing, their growth is mostly oriented horizontally, which means they can express more constraints between more event occurrences. But these languages and mostly CCSL which is the main target of this work lacks a formally defined notion of refinement, which CCSL designers call instant refinement. While calling their need for a refinement mechanism "instant refinement", they possibly imply that it should be possible to somehow split an instant which holds a single event so that it holds a set of events which all refine

the original event. Reciprocally, it also implies that it should be possible to merge a given amount of instants that hold events which all contribute to a common purpose to have an instant that bears an event representing said purpose. This way, engineers that build CCSL specifications would be able to give different constraints to each level of refinement. These constraints would depend on how close one looks at the system, which would be translated in how many times the instants have been broken into smaller pieces. This aspect corresponds to the vertical separation of concerns that was introduced in Section 1.1.5.

While this approach seems appealing as to what it could provide in terms of expressiveness, the fact that instants could be split into several sub instants is problematic. However we look at an instant, it is by definition punctual, solid, unbreakable, which comes in direct contradiction with the core implication of this name "instant refinement". In addition, the instants themselves do not bear any information on the system's behaviour, so refining the instants would only capture the relation between the parts of the refined instant, which is not sufficient to express behavioural properties.

While the will to express constraints at different levels of refinement is not only appealing but also often required by users, splitting an instant into sub-parts seems to be impossible in a literal or in a mechanized manner. As we saw when describing instants, they are unspecified as much as possible but they are a set – or a type – and elements of a set cannot be broken down unless they have a specific structure, which instants do not have because they can be instantiated in various ways. While the name "instant refinement" has been kept in this document because it embeds an intuitive notion of refinement that can be understood easily by system engineers, looking for a solution to refining instants directly is bound to fail. The answer definitely exists – since we often use the notion of refinement, there must be a way to properly model it – but it lies somewhere else.

5.2.2 Our proposal: relating strict partial orders

As our approach is part of a denotational context, we need to formulate a relation between given entities that are relevant to express the notion of refinement. As explained in the previous section, these entities cannot be the instants themselves because they are by nature unbreakable and they don't bear any information on the system's behaviour, which is instead encoded in the strict partial order that binds the instants together. This is this glue, this binding, that should be refined, meaning these are the strict partial orders that should be related to express the notion of refinement. Our idea is that two different levels of refinement should be described and studied with different partial orders, but these partial orders should satisfy a relation which encapsulates the common definition of refinement.

While the notion of refinement has a deep bond with partial orders, the relation of refinement that we propose is a relation between couples of relations. Indeed, as mentioned in Section 5.1.3.a, a partial order is composed of a set, two relations, and the proof that these elements form a strict partial order. Out of these four elements,

the ones that are relevant to be refined are the two relations and the way they interact with one another, because the set of instants is the same at all levels of refinement and the proof is only relevant to ensure that the strict partial order is sound. All in all, the core of our contribution on refinement is the following relation between two couples of relations over instants:

Let Ω be the set of all sets: $\forall I \in \Omega, \forall(<_c, <_a, \approx_c, \approx_a) \in (I \times I)^4$:
$(<_c, \approx_c) <_r (<_a, \approx_a) \overset{d}{\Longleftrightarrow}$

$$\forall(i, j) \in I : \begin{pmatrix} i <_c j \Rightarrow i <_a j \vee i \approx_a j & (1) \\ i <_a j \Rightarrow i <_c j & (2) \\ i \approx_c j \Rightarrow i \approx_a j & (3) \\ i \approx_a j \Rightarrow i \approx_c j \vee i <_c j \vee j <_c i & (4) \end{pmatrix}$$

In this definition, the level annotated by the index c is the lowest (the more concrete) level of observation and a is the highest (the more abstract). We state what it means for a pair of relations to refine another pair of relations. We can only compare pairs of relations that are bounded to the same underlying set of instants. This relation is composed of four predicates, each of which indicates how one of the four relations is translated into the other level of observation.

- *Precedence abstraction:* If a strictly precedes b in the lower level, then it can either be coincident to it in the higher level or still precede it. This means that a distinction which is visible at a lower level can either disappear at a higher one or remain visible, depending on the behaviour of the refinement for these instants – *Equation (1)*

- *Precedence embodiment:* If a strictly precedes b in the higher level, then it can only precede it in the lower level. This means that the distinction between these instants already existed in the higher level, thus cannot be lost when refining. Looking closer at a system preserves precedence between instants – *Equation (2)*

- *Coincidence abstraction:* If a is coincident to b in the lower level, they stay coincident in the higher level. Looking at the system from a higher point of view cannot reveal temporal distinction between events – *Equation (3)*

- *Coincidence embodiment:* If a is coincident to b in the higher level then these instants cannot be independent in the lower level ; they will still be related but nothing can be said on the nature of this relation – *Equation (4)*

There are languages where an instant can only bear a single event. This definition allows and even justifies such approach. Indeed, two instants appearing coincident in a given level of refinement can always potentially be refined up to a point where a distinction appears, which justifies the fact that they can always be attached to a different physical instant. In CCSL, this approach is somewhat advocated even

though it is possible to take the propositional equality as introduced in ref 2.3.4 as the underlying equality between instants in which case two coincident instants will be the same. But the important aspect is that this definition allows such a possibility.

5.3 Mechanization of the refinement relation

5.3.1 Transformation between levels of abstraction

Our relation of refinement is a comparison between pairs of relations. This comparison is by nature partial and ultimately we would like to prove that it is a partial order, as refinement should be. To achieve this goal, the need for comparing pairs of relations for equivalence arises, since a partial order is based on an underlying relation of equivalence. This relation of equivalence between orders should be that all four relations are simply transformed into the corresponding relation in the other level of abstraction. This leads to the following definition:

Let Ω be the set of all sets: $\forall I \in \Omega, \forall (<_c, <_a, \approx_c, \approx_a) \in (I \times I)^4$:

$$(<_c, \approx_c) \approx_r (<_a, \approx_a) \overset{d}{\Longleftrightarrow} \forall (i,j) \in I : \begin{pmatrix} i <_c j \Rightarrow i <_a j & (1) \\ i <_a j \Rightarrow i <_c j & (2) \\ i \approx_c j \Rightarrow i \approx_a j & (3) \\ i \approx_a j \Rightarrow i \approx_c j & (4) \end{pmatrix}$$

This definition is similar to the refinement relation, which directs us towards expressing a common factor between the two. This factor has an intuitive counterpart: the notion of transformation. In both cases, the definitions state how each of the four relations involved is transformed into the other level of abstraction. In the case of the equivalence, these transformations are a lot more straightforward than in the case of the refinement, but they are structurally similar. This leads to the definition of a `Transform` record which is parametrized by the four relations but also the four transformations bound to these relations. This will allow the refinement and the equivalence to be different instantiations of the same structure. As a side note, this factorization is also very convenient for technical reasons on how to manipulate both the equivalence and the refinement relation in the same formal context.

```
1  record Transform {ℓ} (≈₁ ≈₂ ≺₁ ≺₂ : Rel I ℓ)                          1
2    (T≈₁₂ T≈₂₁ T≺₁₂ T≺₂₁ : _ → _ → Rel I ℓ) : Set (lsuc a ⊔ ℓ) where     2
3    field                                                                3
4      ≈₁→₂ : ≈₁ ⇒ T≈₁₂ ≈₂ ≺₂                                             4
5      ≈₂→₁ : ≈₂ ⇒ T≈₂₁ ≈₁ ≺₁                                             5
6      ≺₁→₂ : ≺₁ ⇒ T≺₁₂ ≈₂ ≺₂                                             6
7      ≺₂→₁ : ≺₂ ⇒ T≺₂₁ ≈₁ ≺₁                                             7
```

The four fields of this record express how each of the four relations is translated in terms of the other level of abstraction. This translation is abstractly done by the

transformation functions which takes two orders and builds a third one. From this definition, expressing the equivalence between orders is straightforward, it consists of instantiating the transformation function with the identity over their first or second parameter, as follows:

```
8   _≈≈_ : ∀ {ℓ} → Rel (Rel I ℓ × Rel I ℓ) _                8
9   (≈₁ , ≺₁) ≈≈ (≈₂ , ≺₂) = Transform ≈₁ ≈₂ ≺₁ ≺₂            9
10      (λ ≈₂ _ → ≈₂)                                        10
11      (λ ≈₁ _ → ≈₁)                                        11
12      (λ _ ≺₂ → ≺₂)                                        12
13      (λ _ ≺₁ → ≺₁)                                        13
```

The refinement relation is also a transformation between orders, where the transformation functions correspond to the four predicates of the refinement defined in Section 5.2.2. They are defined using the union of relations and the `flip` function which flips the parameter of a function as described in Section 3.3.1.

```
14  _≺≈_ : ∀ {ℓ} → Rel (Rel I ℓ × Rel I ℓ) _                14
15  (≈₁ , ≺₁) ≺≈ (≈₂ , ≺₂) = Transform ≈₁ ≈₂ ≺₁ ≺₂            15
16      (λ ≈₂ _ → ≈₂)                                        16
17      (λ ≈₁ ≺₁ → ≈₁ ∪ (≺₁ ∪ flip ≺₁))                      17
18      (λ ≈₂ ≺₂ → ≺₂ ∪ ≈₂)                                  18
19      (λ _ ≺₁ → ≺₁)                                        19
```

5.3.2 Proof of partial ordering

The reason behind the definition of an equivalence between pairs or relations is that we expect the refinement relation to exhibit a certain structure when combined with this equivalence. This section presents the proof that this structure is a partial order and binds its constituent to the common properties of the usual notion of refinement as given in Section 1.1.5.

Equivalence of _≈≈_ While we have called this relation an "equivalence" relation since it was defined, the matter of fact is this remains unproven, even though it seemed natural. As a reminder, an equivalence relation is transitive, reflexive and symmetrical. All of these proofs are trivial but are provided in the appendices in Section A.3.2, they lead to the proof of equivalence as follows:

```
20  equiv≈≈ : ∀ {ℓ} → IsEquivalence (_≈≈_ {ℓ})                          20
21  equiv≈≈ = record { refl = refl≈≈ ; sym = sym≈≈ ; trans = trans≈≈ }   21
```

Preordering between _≺≈_ and _≈≈_ A preorder is a structure where the precedence relation is transitive and reflexive towards an equivalence relation. We start by establishing the transitivity of our refinement relation:

```
22  trans≺≈ : ∀ {ℓ} → Transitive (_≺≈_ {ℓ})                                          22
23  trans≺≈ i≺≈j j≺≈k .≈₁→₂ = (≈₁→₂ j≺≈k) ∘ ≈₁→₂ i≺≈j                                  23
24  trans≺≈ i≺≈j j≺≈k .≈₂→₁ x with ≈₂→₁ j≺≈k x                                         24
25  trans≺≈ i≺≈j _ .≈₂→₁ _ | inj₁ y = ≈₂→₁ i≺≈j y                                      25
26  trans≺≈ i≺≈j _ .≈₂→₁ _ | inj₂ (inj₁ y) = inj₂ (inj₁ (≺₂→₁ i≺≈j y))                 26
27  trans≺≈ i≺≈j _ .≈₂→₁ _ | inj₂ (inj₂ y) = inj₂ (inj₂ (≺₂→₁ i≺≈j y))                 27
28  trans≺≈ i≺≈j _ .≺₁→₂ x with ≺₁→₂ i≺≈j x                                            28
29  trans≺≈ _ j≺≈k .≺₁→₂ _ | inj₁ y = ≺₁→₂ j≺≈k y                                      29
30  trans≺≈ _ j≺≈k .≺₁→₂ _ | inj₂ y = inj₂ (≈₁→₂ j≺≈k y)                               30
31  trans≺≈ i≺≈j j≺≈k .≺₂→₁ = (≺₂→₁ i≺≈j) ∘ ≺₂→₁ j≺≈k                                  31
```

This transitivity was expected because the common notion of refinement is transitive. We should be able to chain the refinements while still being a refinement of the original specification. Another expected property is the reflexivity. Indeed, a specification should naturally be a refinement of itself when nothing has been clarified. This reflexivity is relative to the equivalence relation that was defined earlier.

```
32  refl≺≈ : ∀ {ℓ} → (_≈≈_ {ℓ}) ⇒ (_≺≈_ {ℓ})                                           32
33  refl≺≈ i≈≈j .≈₁→₂ = ≈₁→₂ i≈≈j                                                      33
34  refl≺≈ i≈≈j .≈₂→₁ = inj₁ ∘ ≈₂→₁ i≈≈j                                               34
35  refl≺≈ i≈≈j .≺₁→₂ = inj₁ ∘ ≺₁→₂ i≈≈j                                               35
36  refl≺≈ i≈≈j .≺₂→₁ = ≺₂→₁ i≈≈j                                                      36
```

Compiling the different elements that were proven, we can instantiate the preorder structure with our relations.

```
37  preorder≺≈≈ : ∀ {ℓ} → IsPreorder _≈≈_ (_≺≈_ {ℓ})                                   37
38  preorder≺≈≈ = record {                                                              38
39    isEquivalence = equiv≈≈ ;                                                         39
40    reflexive = refl≺≈ ;                                                              40
41    trans = trans≺≈ }                                                                 41
```

Partial ordering between _≺≈_ and _≈≈_ A partial order is a preorder with the additional property of antisymmetry between its two relations. In the case of refinement, if two specifications refine each other, they should definitely be equivalent and represent the exact same level of abstraction which tells us that the antisymmetry should hold, hence implying that our two relations form a partial order, as required when assessing refinement. The proof is given in the appendices in Section A.3.3.

5.4 An example of instant refinement

5.4.1 Presentation of the example

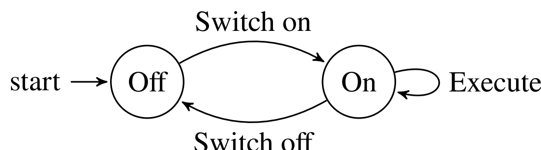


Figure 5.7: A simple system

The mathematical relation defined above aims at providing a formal construct to model and assess refinement between traces of execution. To illustrate its relevance, we propose to apply it to an example chosen for its simplicity and accuracy with respect to the idea of refinement. This is a simple system whose behaviour is represented as a transition system depicted on Figure 5.7. This system can be switched on and off. While it is on, an action can be executed any number of times. A possible trace of this system – amongst an infinite number of them – is depicted in Figure 5.8. t_{on}, t_{off} and t_{ex} respectively represents the occurrence of "switch on", "switch off" and "execute" transitions.

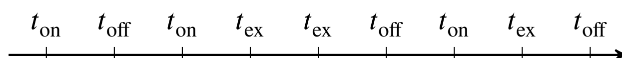


Figure 5.8: A trace on a single timeline

This trace starts with the birth of the system – an abstract moment on which events start to occur – and possibly goes on indefinitely, which makes this representation partial. In addition, this design places each event on the same timeline, thus ignoring horizontal separation. In order to make it visible, from now on we will represent every different event on a specific timeline, as shown on Figure 5.9. This approach is used in CCSL, where each timeline is represented by a clock which tracks the occurrences of a specific event. The instants on each timeline are totally ordered and those in the same vertical dashed blue lines are coincident.

Figure 5.9: One timeline per event

The action executed by the system while running can be specified in various ways. Here, we imagine that our system is connected to a light through the use of memory containing a variable x. This variable is assigned the value 1 or 0 by our

system and the light is turned on and off accordingly. When the system is switched on, the light remains down until a button is pressed which turns it on. Pressing the same button will alternatively turn it off and on. Shutting down the system turns it off. This behavior is depicted in Figure 5.10.

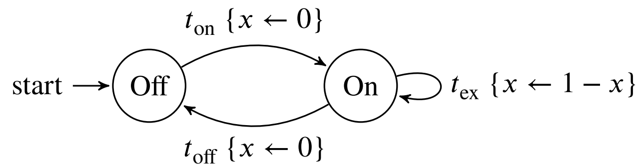


Figure 5.10: The system pilots a light

By specifying our system behaviour, we defined events that can be added to its traces. t_{x_0} and t_{x_1} respectively correspond to the variable x being assigned 0 and 1. These additions belong to horizontal separation since we added a new part to our system (the module linked to the light). One of the possible traces is depicted in Figure 5.11. Some events are occurring simultaneously, for instance t_{on} always occurs on an instant coincident to an occurrence of t_{x_0}. Such relation between events can be defined in CCSL (a simple case of sub-clocking).

Figure 5.11: The trace of the system with the addition of the variable x

It is important to notice that when specifying the action executed by this system, we implicitly took a certain point of view. We deliberately ignored some lower level concerns such as the way a computer system handles memory. This is where vertical separation takes place. Looking closer at the machine will lead to other events which can refine the access to the variable x. For instance, the "switch on" event can be viewed as a succession of actions, such as powering up the system, retrieving the address of x, computing (here there is no actual computation since 1 is a constant value, but there could be some in the case of a more complicated expression) the value of 1 and storing this value at the right address. These events, except for the first one, are used to handle the computation and the storage of a value in memory. Taking into consideration these events requires us to view the system at a lower level than before, in which case its representation as a transition system is depicted in Figure 5.12.

The "switch on" transition has thus been refined in several transitions. t_{on} represents the powering of the system, t_{stack} the stacking of the address of x, $t_{compute}$ the

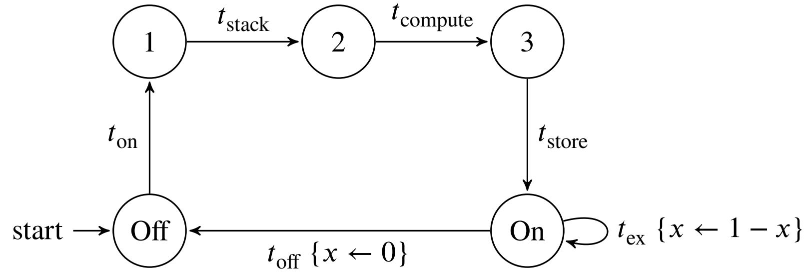
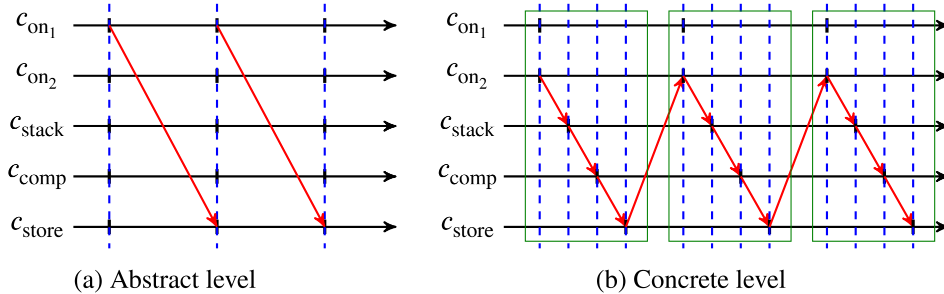


Figure 5.12: The refined system

computation of the value of the expression 1 and t_{store} the storage of the computed value at the stacked address. Note that we only refined one transition here for the sake of clarity and simplicity. Refining the other transitions would rely on exactly the same reasoning which is of no use for the relevance of this example.

This analysis induces two different points of view on our system. The higher level of observation is represented on Figure 5.13a. From now on, for the sake of clarity, the events that are not refined are omitted. They don't influence the reasoning we are conducting, thus their omission is acceptable.



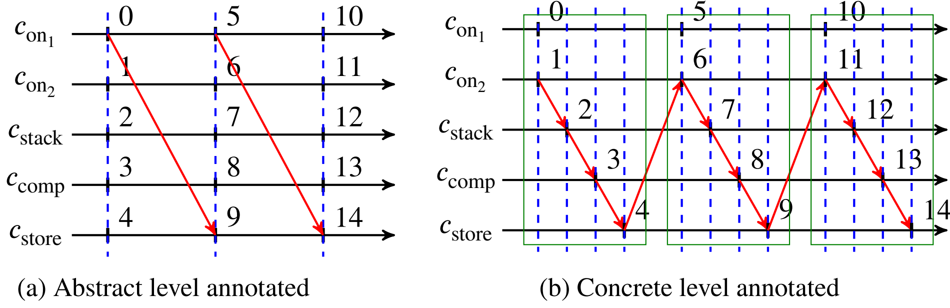
(a) Abstract level (b) Concrete level

Figure 5.13: Both levels of observation

From the higher point of view, all the instants on which the sub-events occur are both equivalent to each other and to the containing event. Their underlying order is hidden and has no impact on the trace of the system at this level. The lower point of view, however, is different, as depicted on Figure 5.13b.

For the lower level of observation, the different instants are ordered in a way such that they respect the specification in Figure 5.12. The blue dashed lines represents the equivalence classes induced by the respective partial orders while the red arrows represent the precedent relations of these orders (we did not represent the links that can be deduced by transitivity or other properties of partial orders).

Until now, the instants on which the events occur formed an unspecified set. Since our goal is to mechanize this example, we need to instantiate it to an actual set. We chose the natural numbers because they allow us to annotate the traces while expressing quite easily the relations at both levels of refinement. The annotated



(a) Abstract level annotated

(b) Concrete level annotated

Figure 5.14: Both annotated levels of observation

higher levels of observation are given in Figure 5.14.

This representation allows us to define the coincidence and the precedence relations that bind its different instants, as subsets of $\mathbb{N}\times\mathbb{N}$. Since both these relations must be transitive, the coincidence must be symmetrical and they must form a strict partial order. We omit the related elements which can be deduced from these properties.

Coincidence Relation			**Precedence Relation**
(0 , 1)	(0 , 2)	(0 , 3)	(0 , 5)
(0 , 4)	(5 , 6)	(5 , 7)	
(5 , 8)	(5 , 9)	(10 , 11)	(5 , 10)
(10 , 12)	(10 , 13)	(10 , 14)	

Since the traces are infinite, there are an infinite number of couples in each relation. We only expressed them for the visible subset. We now define these relations for any natural number, by relying on the euclidean division of their operands by 5:

$$\forall(a,a') \in \mathbb{N}^2, \exists!\,(q,r,q',r') \in \mathbb{N}^4 \;:\; a = 5q+r \wedge r < 5 \wedge a' = 5q'+r' \wedge r' < 5$$

These relations, using the same notation, are defined as follows:

$$\forall(a,a') \in \mathbb{N}^2, a \approx_2 a' \overset{d}{\iff} q = q'$$
$$\forall(a,a') \in \mathbb{N}^2, a <_2 a' \overset{d}{\iff} q < q'$$

The same work can be achieved for the lower level of observation, which is displayed on Figure 5.14b. The relations extracted from Figure 5.14b are depicted in the table below. As previously explained, only the relevant couples are mentioned.

Coincidence Relation	**Precedence Relation**		
(0 , 1)	(1 , 2)	(2 , 3)	(3 , 4)
(5 , 6)	(4 , 5)	(6 , 7)	(7 , 8)
(10 , 11)	(8 , 9)	(9 , 10)	(11 , 12)
...	(12 , 13)	(13 , 14)	...

By taking the same decomposition as before, we can mathematically define the relations at the lower level of observation.

$$\forall (a, a') \in \mathbb{N}^2, a \approx_1 a' \overset{d}{\iff} (q = q') \wedge ((r = r') \vee (r + r' = 1))$$
$$\forall (a, a') \in \mathbb{N}^2, a <_1 a' \overset{d}{\iff} (q < q') \vee ((q = q') \wedge (r < r') \wedge (r' \neq 1))$$

Since both couples of relations have been defined mathematically, we can prove that they correspond to a situation of refinement. This has been done in AGDA and is the purpose of the next section.

5.4.2 Verification of the example

The verification activities for the example consist in the following steps:

- Definition of the 2 couples of relations, one for the abstract level and the other for the concrete level of refinement.
- Proof that both these couples form a partial order.
- Proof that these partial orders are in a relation of refinement.

The complete verification has been developed and is presented in the appendices in Section A.3.4.

Assessments

This chapter proposed a new perspective of the usual notion of refinement between systems, by adapting it to trace semantics to answer the following question: how refinement should be expressed inside a context containing instants and partial orders ? Rather than refining the systems directly as often seen in the literature as depicted in Section 1.1.5, we proposed a formal context where this refinement can be expressed: a context where several partial orders coexist, each of which corresponds to a given level of observation, hence a given level of refinement. These partial orders must obey certain conditions to take part into this hierarchical structure: these conditions have been expressed and specified through four predicates which, when combined together, form a relation between partial orders, which corresponds to our view on refinement. This relation of refinement must be established for each couple of partial orders when the first depicts a level of observation thinner than the latter. Thanks to some properties which have been established in a formal context around our relation of refinement, and especially its transitivity, it is sufficient to prove that each partial order refines the next one to establish the soundness of the whole structure. This combination of partial orders refining one another gives a formal context richer than the usual single partial order on which languages which deals with trace semantics usually rely. One of these languages is CCSL, whose existing semantics has been mechanized and enriched accordingly, as depicted in the upcoming Chapters 6 and 7.

Chapter 6

A mechanized denotational semantics of CCSL

Outline

Since our main goal is to prove properties on languages whose semantics can be expressed as traces, it is natural to work on a language which allows us to express such semantics. This chapter describes our work on one of them, the Clock Constraint Specification Language (CCSL), on which we made several contributions, using the following outline:

1. Section 6.1 presents the CCSL language in a model-oriented engineering perspective. It explains both the conceptual and technical contexts in which it was defined. It presents the notions of modelling language, V-cycle, UML, ECORE and places CCSL among these notions. It is a section mostly composed of context and state-of-the-art references.

2. Section 6.2 presents our contributions around the mechanization of CCSL which contains the CCSL constructs that are not index-dependent, ie that do not require a discrete representation of time. This mechanization consists in transferring the CCSL elements which only had a paper version of their semantics in a formal context. This mechanization contains an adaptation / correction of these CCSL notions according to our formal context and some corrections and discussions around minor imprecisions which were discovered. Each construct has been verified through conformity properties which embed the usual semantics that CCSL users are used to. In this context, some constraints over clocks have been added which, in certain cases, are useful for such properties to be sound, ie for the validation of the informal CCSL specification. Concretely, this means that the formal definition of the clock has been completed when it was not rich enough to establish some required properties.

6.1 CCSL: A language to abstract event occurrences

Our notion of refinement is generic and thus can be applied and reused in any language that either results in temporal executions or that describes such temporal executions. This section presents one of them, the Clock Constraint Specification Language CCSL. We first introduce this language in a generic manner, then we describe the tools that already exist around it (namely, TIMESQUARE) and talk about the paper version of the denotational semantics made by the designers of this language to build up to the next section where we present our mechanization of this semantics that ultimately allowed us to apply our notion of refinement to CCSL. But first, we need to introduce Domain Specific Modelling Languages (DSMLs) as CCSL is one of them.

6.1.1 Domain Specific Modelling Languages

CCSL is a Domain Specific Modelling Language (DSML) which we chose as our target language for our mechanization objectives, for several reasons that will be detailed in this section. But before getting to the technical aspects of CCSL as well as the work we did around this language, let's first present the context of DSMLs and, more generally, model engineering that led to the development of CCSL.

This section is a short overview of the appearance of DSMLs in computer science in order to contextualize CCSL in the field of model engineering. It is not exhaustive in the sense that this domain is complex and can hardly be described in a short manner. However, the required information will be given to understand what problems CCSL aims at solving and why, for instance, it has first been defined as a UML profile instead of a stand-alone modelling language. The distinction between these two options will also be contextualized and somewhat explained, even though, after researching the matter, this appears mostly to be two different sides of the same coin. This introduction on modelling languages will be separated in three parts, each of which brings us closer to CCSL and further from the global context of modelling languages.

6.1.1.a The rise for the need of modelling languages

For centuries, and I might even say for millennias, humans have developed complex systems. These systems, however simple as they might appear to us now, did take some engineering to be developed, long before computer science started to infiltrate every field and become a mandatory part of most system development. These ancient systems – The Nil irrigation system, the Pyramids, the great wall of China, ... – have always required a succession of steps in order to complete their design and building. These steps, that we have now formalized, are often modelled and explained as a V cycle, depicted in Figure 6.1. This cycle, well known by any engineer and engineering student, summarizes the succession of steps required to develop and assess the correctness of a system. The left part contains the different phases in the development while the right part focuses on the different steps

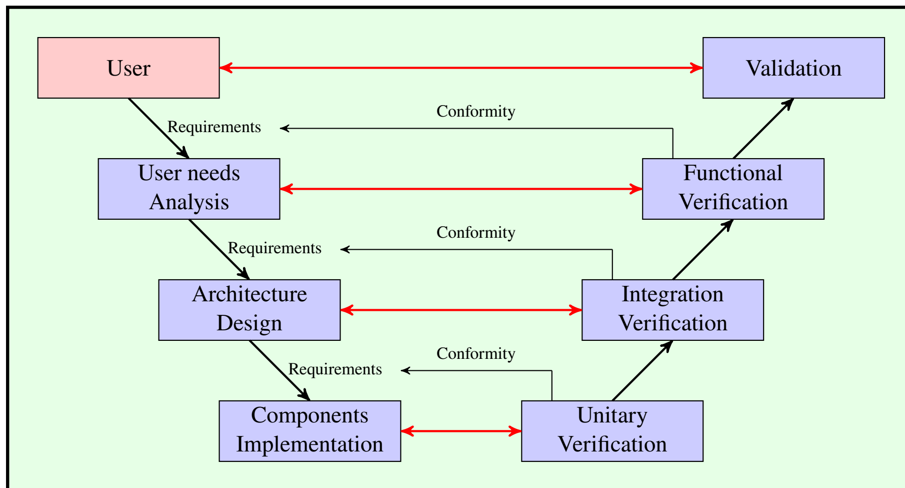


Figure 6.1: The V cycle in a system development

in the development assessment, that are all linked to a specific development step associated to a set of requirements. These steps are as follows:

- The development steps:
  1. The analysis of the user's needs should ensure that he has been correctly understood by the system developers.
  2. The architecture of the system is then designed according to the result of this analysis.
  3. The components are designed and implemented regarding the chosen architecture for the system.
- The validation and verification (V and V) steps:
  1. Unitary verification is made towards the components that have to satisfy the architectural requirements.
  2. Integration verification is held towards the architecture to satisfy the analysis of needs requirements.
  3. Functional verification is done to assess the conformity with the original client's requirements.
  4. User validation requires the user to validate the finite product.

In this theoretical life-cycle model, the development and verification steps are done one after the other, which means that the verification can only be done after the development has been accomplished because it is applied on the built system. Iterative and agile processes allow us to split the whole development in various iterations that allow us to conduct V and V activities on parts of the system, yet these

parts need to be fully built before being assessed. This is a huge drawback of the V and V model because, should the verification fail, the developer has to redo everything from scratch for each failure. In system development, this might be costly in a lot of different aspects (time and money being the most obvious ones) and should be avoided as much as possible. In that regard, engineers experienced new ways to work around this V pattern to accelerate the verification process and move towards the development process. The idea is to represent the artefacts resulting from each step with models on which V and V activities can be made without having to complete the whole process and build the product before running said activities. These models are of various sorts and are motivated by various concerns, but they all serve the purpose of a V and V that precede the next phase in the development of the system. In our examples from the antiquity, the systems are straightforward yet already complex, and mathematics was developed as the language for building models and the scientific method was designed in order to build and assess these models. However, nowadays, in systems such as air-planes or communication systems, their underlying heterogeneity and complexity induces the need for a lot of different models that have to be synchronized. All these models are either expressed in concern dedicated languages (Domain Specific Modelling Languages (DSML)), or in generic purpose languages that allow us to express most concerns, such as UML. These generic purpose languages are languages with which they can be developed and verified. Another justification for this need is the V and V process, that was commonly done in an informal manner, through the reading of even more informal documents. Models can help increase the efficiency of such verification through a more formal account of the system development artefacts.

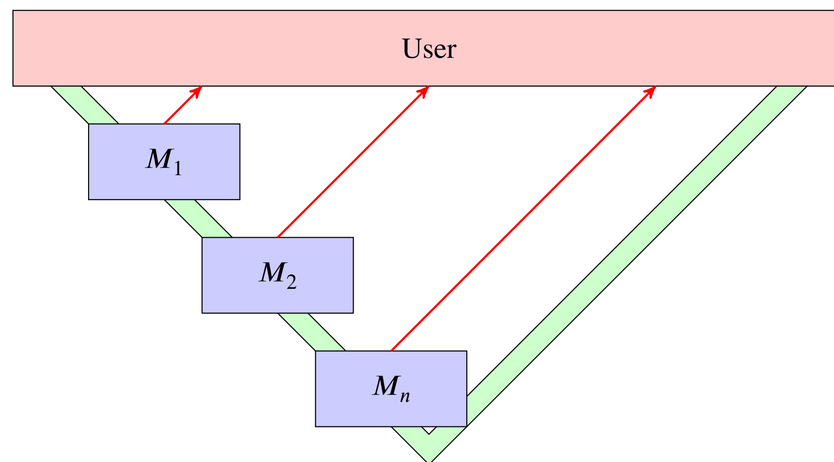


Figure 6.2: A simplified V cycle with the use of models

As depicted in Figure 6.1, the user is on top of the cycle because he provides the system's developer with the functional requirements that induce its different steps. However, the contact with the user is then broken until the functional verification has been conducted, which means that there are no guarantees that the product will

indeed satisfy the user's needs. The models will also be used as a bridge between the user and the developer while the product is being developed. At each step, the models will allow the user to conduct validation activities, which means having an overview of the current state of the project and supervising that it's being developed correctly according to its requirements, that might have been misunderstood. If such a misunderstanding would happen without the use of models, it might have been propagated all the way until the functional verification, at which point any important change in the system would be overwhelming. This cycle, with the use of models, can be depicted in Figure 6.2 where the relation between the system and the user is maintained throughout the development process.

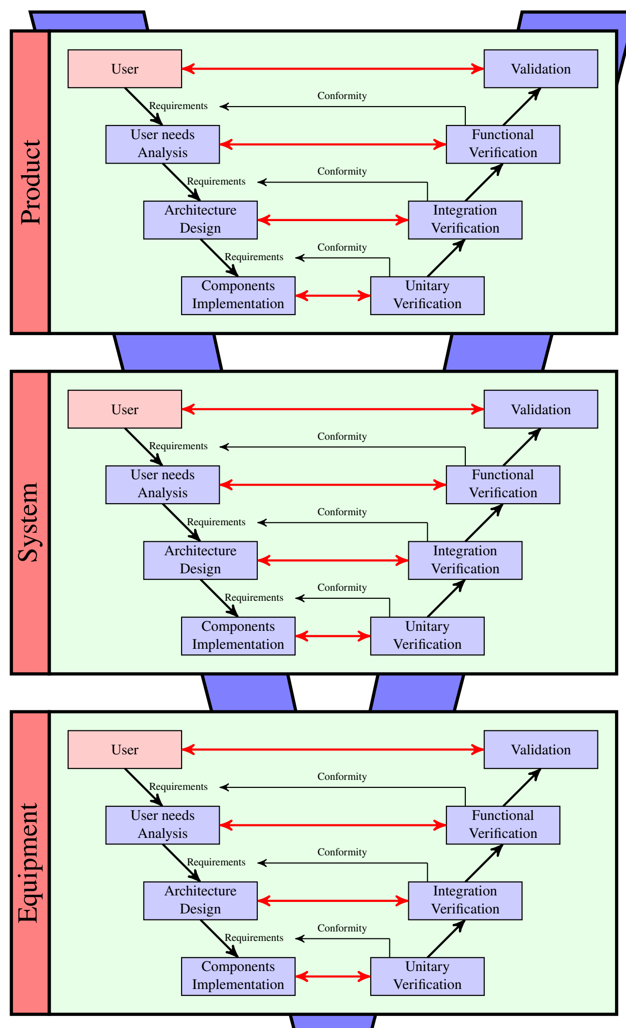


Figure 6.3: The V cycles in a product development

As complex as these systems may appear, and as relevant as the need for models is, there still are some more steps that induce an even greater need for models. This

V cycle alone can handle the development system of a reasonable size, but several of these cycles have to be used to scale towards greater projects. In these cases, the global product is split into several conceptual entities, each of which derives from another and has to be handled through a V cycle. These steps are as follows:

- The system of systems, which is the most global entity. Its level of complexity is very high, but it is understood by the user. The product can be a car, an air-plane, a satellite, a communication system, and so on. A system of systems is composed of several systems.

- The systems, which are abstract entities derived from the systems of systems, each of which aims at solving a specific issue. In the example of an air-plane, a system is for instance the propulsion system, the navigation system or the safety system. Each system can be seen as a set of equipments.

- The equipments, which are the concrete elements to support the system. They can be wings, engines, sensors or any concrete device whose goal is to provide one or more functions inside one or more systems.

Having these elements in mind, the global picture of the product development is in fact a succession of V cycles for each of these entities, as depicted in Figure 6.3.

6.1.1.b The development of modelling languages

The number of steps in a system development as well as the need for the users to be able to witness its evolution and assess its intermediate correctness led to the need of defining models to describe all those steps. These models had to be defined following a syntax and semantics from a specific language, and in that regard many languages have been defined throughout the past decades. Such languages are Modelica [143], Simulink [155], Stateflow [38], Scade [27], Petri Nets [132], Automatas, StateCharts [104], flow charts [122], AADL [65], EAST-ADL [151] and many more. These languages were very promising and some of them are still used today to model specific systems. However, they were separate entities specific to certain concerns and did not give the option to handle big models constituted of different sub-parts. Another flaw of this approach is that the engineers using these languages had to learn several different ones when defining different models. This led to the creation of the Unified Modelling Language (UML) in the beginning of the 1990s [128] whose goal was to provide a unique language with sufficient expressiveness to allow the modelling of any preoccupation in the development of information systems using software.

This goal was very ambitious and the three creators of UML realized that to accomplish it, they had to embed the ability to extend the language itself inside the language. Thus create new language constructs and express their relation with the constructs already available in the base language. This mechanism is called a profile and these profiles allow us to define anything in UML, while still staying in the UML world. The consequence of this approach is that any profile benefits from all

of the UML tools and properties, which might or might not be needed. One of these profiles is MARTE (Modeling and Analysis of Real-Time and Embedded systems), which contains CCSL to model timing aspects mandatory in the modelling of real time systems. This hierarchy is depicted in Figure 6.4

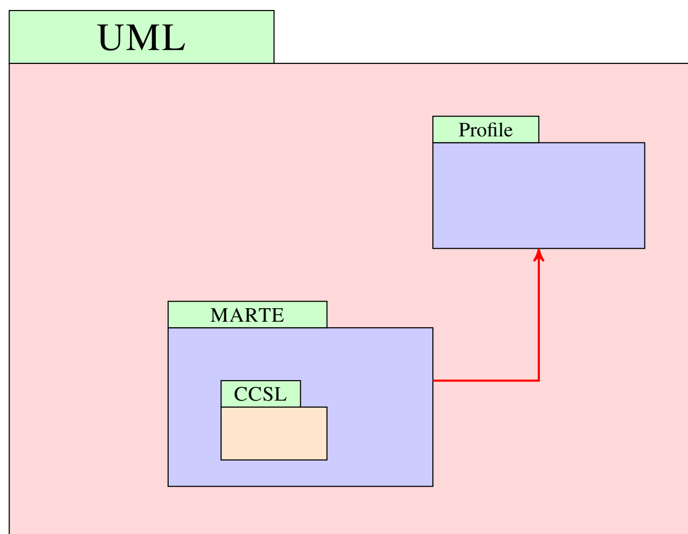


Figure 6.4: The place of CCSL inside the UML world

The definition of UML was originally informal, which means its specification remained written on paper using the English language as well as informal graphical schemes (whose semantics relied on human interpretation) and did not have any formal counterpart. However, since UML was designed to model anything, it is possible to model a language in UML and thus to model UML in UML. This assessment led the UML developer to seek the smallest part of UML with which UML itself could be formally defined. This smallest part, after being discovered, was called MOF and is both a part of UML and the language with which UML is now formally defined, as depicted in Figure 6.5.

6.1.2 Presentation of CCSL

CCSL [6] is a DSML that was designed by Charles André and Frédéric Mallet as a UML profile as part of the MARTE standard [129]. More precisely, MARTE is the current OMG standard for modelling real-time and embedded applications, and CCSL is part of it. CCSL is defined as a meta-model (a language) that is presented in a simplified version in Figure 6.6. All the details of this meta-model are not useful for the understanding of this document, and the notions that are relevant will be detailed thoroughly when describing the mechanization we did of the language. However, here is a short introduction for these concepts. CCSL is used to model concurrency between the execution of either different systems or different parts inside a system. This means that CCSL has its own definition of events

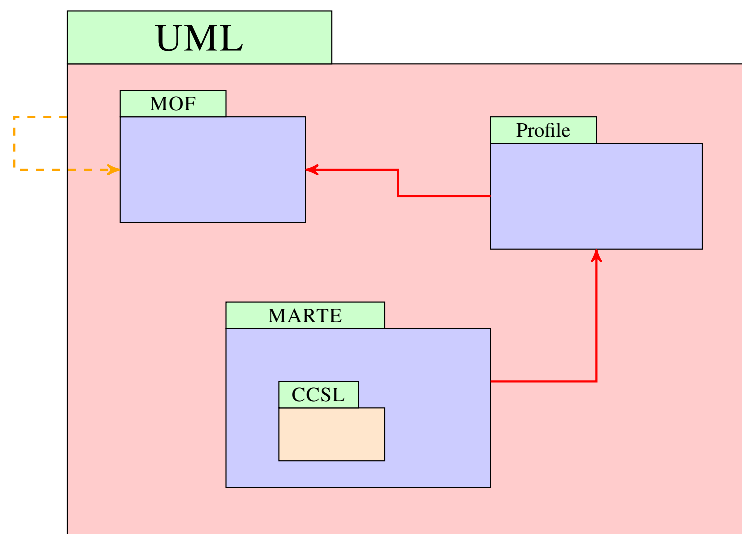


Figure 6.5: UML as an instance of MOF

executed asynchronously as well as systems, even though this one is hidden for a very convenient reason and is only defined in other parts of MARTE. But before explaining how CCSL defines events, let us have a look at what the dictionary [1] says about them. It reads "a thing that happens or takes places, especially one of importance". This definition is particularly instructive, for two specific reasons.

The first reason is the most pragmatic one. It says an event is something that happens, and this is indeed what is represented by events in CCSL, even though there is a slight difference. CCSL distinguishes the phenomenon itself from its occurrences. Basically, there exists a CCSL construct to model the events as "things that *might* happen" (namely the clocks), and there are other constructs to model the events as "their occurrences" (the instants). Although the instants can be seen as pre-existing entities on which events can occur, there are different reasons to consider that they have no real existence until they have to bear an event occurrence. This question is, however, mostly philosophical and has little impact on our mechanization.

The second part of the event definition leads to the second reason why I find this definition very instructive. It says "especially one of importance". As stated in the previous section, depending on the level of observation we take regarding the definition of our system, some events might or might not be considered. When they are not, to a certain extent, this means that from our point of view, they do not exist or relating to the definition, that they are not of importance. I find it very comforting that the event definition found in an ordinary dictionary explains our comprehension of events in computer science so well. To summarize, the events in CCSL, are represented by clocks and each different event is represented by a

[1] dictionary.com

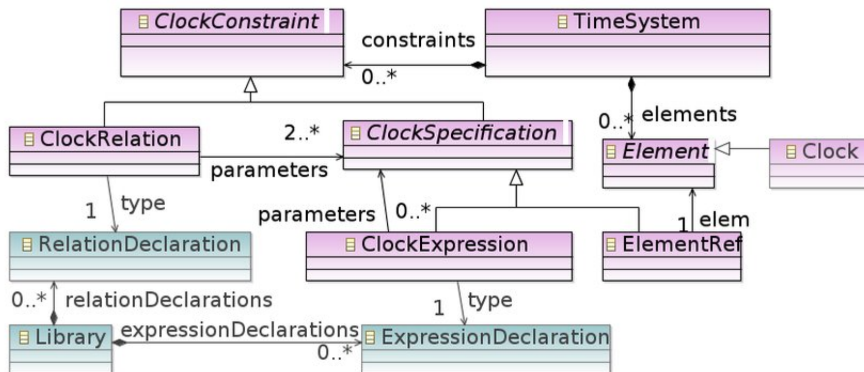


Figure 6.6: A simplified CCSL meta-model

different clock. By associating a relevant name to a given clock, it allows us to keep track of which clock represents which event in a given system. These clocks are said to "tick" whenever the event they represent occurs, in which case an instant emerges for this clock (similar to the emergence of new particles in quantum physics when the void is given a substantial amount of energy, although this analogy is purely anecdotic) and bears this event occurrence. Considering this definition, it becomes obvious that an instant cannot bear several event occurrences.

On top of these clocks we can define the notion of system, although it is not native in CCSL for a very specific reason. A system can be defined as a set of clocks that have a collective purpose, although it is more often seen as an existing entity from which clocks are extracted regarding its behaviour. The reason why the notion of system is not natively defined in CCSL is why this language is so beautiful in my opinion. It is simply because systems are not relevant when defining a CCSL model. Knowing that a given subset of clocks form what we consider as a system might be useful when extracting information from the model, but not when defining it. This simple fact has two direct implications: first, this means that several systems can be modelled using CCSL as though they were only a single one. Secondly, this means that properties between systems will be expressed in the same manner as properties inside systems, which will ultimately be very convenient for composition. The CCSL operational semantics (namely the TIMESQUARE tool, that will be briefly presented in the following part) can then try to solve the constraints associated to the model without taking into account which system the clocks come from. This is particularly interesting because it questions what we usually call a system.

Since each clock in a system represents a distinct event that occurs during the system temporal execution, these clocks are, by default, not connected to each other. Thus, defining the different clocks a system provides is mandatory but is only the first step towards the description of the temporal semantics of such systems. In order to complete this description, one must provide ways of relating these clocks to each other, thus allowing the system's designer to express constraints to the execution of the system. In that regard, CCSL provides constructs that allow us to do such

bindings. These constructs are called constraints which are of two kinds: the relations and the expressions. The relations express temporal constraints between two clocks, for instance, one can express a notion of precedence between two clocks, which means that the occurrence of the slower clock is always preceded by a distinct occurrence of the faster clock. As for expressions, they allow the creation of a new clock from an existing clock, for instance by uniting their occurrences. All these notions will be heavily detailed and clarified when mechanized in Section 6.2.

6.1.3 TIMESQUARE: an operational semantics to CCSL

In order to give a meaning to a language, one must provide a semantics for it. There are several ways to do so (several kinds of semantics), but the usual one is the operational semantics. These semantics are simply processes which, from an instance of the language they represent, construct the resulting semantic elements for this instance. The term "element" is purposely vague, because different languages can induce various kinds of outputs. In the case of CCSL, whose goal is to represent the possible temporal executions of a system from a set of constraints, its operational semantics must provide at least one trace of execution for a given system (traces of execution were defined in Section 5.1.1). An operational semantics provides a deterministic procedure to build the result of what the language describes. It is a semantics in the sense that it gives a meaning to the language by giving the semantics of the models / programs this language describes by explicitly building the result of their execution. For instance, an operational semantics would say that $1 + 2$ is equal to 3, without explaining what the quantities 1, 2 and 3 are, nor what $+$ means. In the case of CCSL, as stated before, such an operational semantics must give at least one possible trace of execution for a given set of clocks and constraints. Such a semantic exists and is named TIMESQUARE. It takes the form of an Eclipse product designed using the Eclipse framework EMF (Eclipse Modelling Framework) and its associated tools. TIMESQUARE takes as input a CCSL specification, an example of which has been presented in Julien Deantoni's work [58] and is depicted in Figure 6.7.

This specification defines four clocks as well as an expression built from these clocks and three relations over them. This specification forms the CCSL model – conforming to the CCSL meta-model – and the operational semantics of CCSL can be applied to such models to assess whether or not there exists a trace that satisfies the specification and to give a possible trace of execution in the positive case. TIMESQUARE provides such a trace, as depicted in Figure 6.8. This figure shows the ticks of each of the four clocks as they can occur regarding the specification. This timeline also features red and blue arrows which act as witnesses of the underlying partial order induced by the constraints and visible on the diagram.

Since CCSL provides constructs to describe the execution of distributed systems, it does not rely on a total order between the occurrences of the events they exhibit. Rather, it works on partial orders to bind the instants together whenever possible regarding the set of constraints attached to the system. This means that two

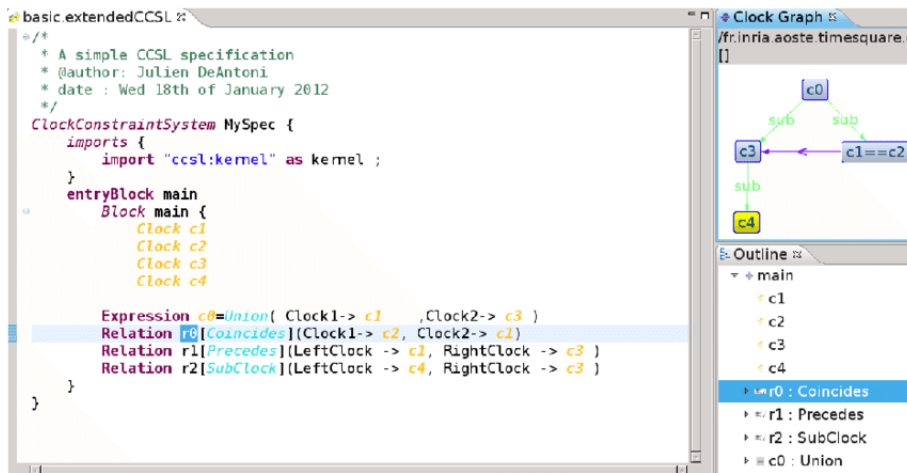


Figure 6.7: A simple CCSL specification in TIMESQUARE

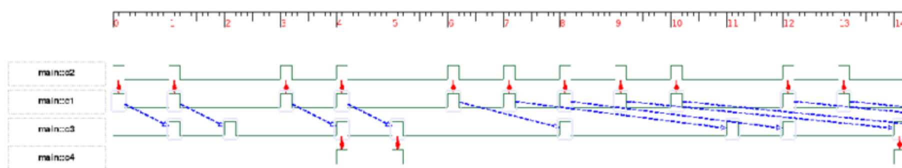


Figure 6.8: A possible trace associated to the specification

instants are either coincident, precedent, or unrelated in such orders, as described in details in Section 5.1.3. In the trace depicted in Figure 6.8, the red arrows stand for coincidence while the blue ones stand for precedence. When TIMESQUARE finds a solution for the system, it projects the partial order on an arbitrary total order (through a process called the linearisation of the partial order) and this clock diagram is indeed the representation of such total order, while the arrows are depicting what we actually know about the system – the partial order – rather than what we can arbitrary define through linearisation – the total order.

TIMESQUARE is fully operational, which means it consists of a function which takes a CCSL model as input and outputs, whenever possible, a possible trace of execution for that model, hence giving a meaning to it. According to TIMESQUARE, a CCSL model is an entity which allows the construction of a set of traces – which might be empty – nothing more and nothing less. As we stated before, this semantic, however useful for practical cases, is not sufficient to fully embrace the meaning of CCSL, which is why a denotational semantic has been defined. A denotational semantic (or relational semantic) gives meaning to a language by expressing relations over the elements it provides. It does not build anything but rather gives relations to verify if a given model is correct. These two kinds of semantics (operational and denotational) are usually complementary. Indeed, a denotational semantic defines a contract which the operational semantic should comply with in which case they

both convey the same underlying semantics of the language they describe. More on this link is expressed in the following Section 6.1.4.

6.1.4 A paper version of the denotational semantics of CCSL

While TIMESQUARE provides an operational and powerful way of solving CCSL constraints through the generation of a conforming trace when possible, its reach is limited to that extent. It does not say how the CCSL construct behaves outside of their operational function. In this operational semantic, every construct can be seen as an operator which reduces the possible set of traces for the specification, nothing more. This is very useful in practical works, and the team that develops CCSL uses TIMESQUARE on a daily basis to assess specifications provided by system developers. However, the operational semantics is limited because it does not express the meaning of the CCSL construct outside of a concrete specification on which they can be applied. Thus, nothing can be proved on them in a formal manner. This is why CCSL needed another kind of semantics, a relational one, both to grow confidence in the way the constructs had been defined as well as being able to prove high level properties on these constructs. This is why the CCSL developers made a technical document on which they defined relationally their constructs regardless of the models these constructs were used in. This relational semantic does not build any output regarding a specification, but rather describes the relations induced by the CCSL constructs. For instance, given a specific CCSL relation R and two clocks c_1 and c_2, this semantic explains what it means to write $c_1\ R\ c_2$. More precisely, it explains how the underlying partial order between the instants is impacted by this relation. This paper-version of the relational semantics of CCSL can be found in [56] and is the foundation of our mechanization of CCSL, where we picked the different constructs of CCSL as they were described in this document and mechanized them in our AGDA development. This work will be described in Section 6.2, as well as the semantics of the CCSL operators we mechanized and the changes we had to make. These changes are either related to AGDA itself, the way the semantics was defined in the paper, or some imprecisions we had to tackle and clarify. While the paper-version of this semantic was mostly exhaustive and precise, adapting it in a proof assistant always requires some changes because these tools allow no shortcuts nor imprecisions both in the definition of the concept and in the proofs around them. As doing such work on paper is more permissive, it can possibly lead to incomplete or even inconsistent semantics. Detecting and removing these issues is the first benefit of mechanizing these semantics.

6.2 A mechanized semantics of CCSL

Our contribution around CCSL can be summarized as follows: The core concepts depicted in the paper version of the denotational semantics have been mechanized in AGDA. Throughout this process, the notions that needed clarification have received so, and those that were not completely relational have been expressed in

this manner. Occasionally, minor mistakes that were present in the original semantics have been rectified. While the paper version of the denotational semantics of CCSL only consisted in the definition of the notion used in CCSL, without further properties or proofs, our mechanization follows the methodology advocated throughout this document. This means that during the process of modelling the constituents of CCSL in AGDA, we expressed properties over these constituents. These properties are the reflection of the understanding we have over how CCSL should behave, which has been acquired through the reading of publications but also by personally meeting and working with the team that created CCSL. These properties have been proven in AGDA and constitute a solid argumentation towards the confidence one can have regarding our mechanization. This section presents our mechanization in a linear manner in terms of the CCSL constructs, as opposed to a more thematic approach where the notions and the properties around them would be separated. We believe that, in a development such as this one, properties and proofs should be conducted throughout the whole process, and this is what motivates this choice. Since some definitions as well as notions about the modelling of time have been described and handled in Section 5.1, we only present here additional time elements that were not used then, the intervals and subsets of instants. All this work has been verified through the addition of properties that are presented as well.

6.2.1 Time-related notions

Basic notions regarding time have been introduced and modelled in Chapter 5. CCSL requires additional elements on top of instants and strict partial orders, which are the intervals and the subsets of instants.

6.2.1.a Intervals

CCSL requires the use of intervals to express constraints on given portions of time. This notion is defined in our framework as follows:

```
1  data Interval : Set where
2    ⟦_-_⟧ : (α β : Support) → Interval
3    ⟦_-∞⟦ : (α : Support) → Interval
```

We define a data type called `Interval` which provides two constructors. They represent two different kinds of intervals, one that is bounded on both sides and one that is only bounded on the left side. This asymmetry is inherent to the underlying asymmetry in a system life: it was started at a specific time but might be running forever. It is important to note that CCSL aims at handling the global lifetime of a system through the use of a birth instant and a possible death instant, hence expressing the asymmetry for the total duration of the system life. These notions have been tackled in this work and will be presented in our modelling of CCSL in Section 6.2.2.e.

We then define the membership in an interval:

```
4   _∈ᵢ_ : Support → Interval → Set                                4
5   v ∈ᵢ ⟦ α -∞⟦ = α ≼ v                                           5
6   v ∈ᵢ ⟦ α - β ⟧ = α ≼ v × ¬ (β ≺ v)                             6
```

The membership is a predicate over two parameters, defined as a mix-fix operator (the underscores indicate where the operands will be provided). It gives two expressions on how an instant can be considered belonging to an interval which is either unbounded on the right or bounded on both ends.

We define an operator that retrieves the lower bound of an interval, which always exists:

```
7   inf : Interval → Support                                       7
8   inf ⟦ α - _ ⟧ = α                                              8
9   inf ⟦ α -∞⟦ = α                                                9
```

From the membership definition, we can also define interval inclusion, which states that `I` is included in `J` when every instant in `I` is also an instant of `J`. It is defined from the notion of predicate inclusion `_⊆_` that exists in the standard library.

```
10  _⊂ᵢ_ : Rel Interval _                                          10
11  I ⊂ᵢ J = (_∈ᵢ I) ⊆ (_∈ᵢ J)                                    11
```

These definitions, as always in our approach, are to be verified or at least given properties on how they must behave. The following property states that two coincident instants are members of the same intervals. The proof revolves around the underlying properties of the instants being partially ordered.

```
12  ≈∈ : ∀ {i₁ i₂ I} → i₁ ≈ i₂ → i₁ ∈ᵢ I → i₂ ∈ᵢ I                                12
13  ≈∈ {I = ⟦ α - β ⟧} i₁≈i₂ (α≼i₁ , ¬β≺i₁)                                     13
14     = ≼-resp-≈₁ i₁≈i₂ α≼i₁ ,                                                 14
15       (λ β≺i₂ → ¬β≺i₁ (≺-resp-≈₁ (sym≈ i₁≈i₂) β≺i₂))                          15
16  ≈∈ {I = ⟦ α -∞⟦} i₁≈i₂ (inj₁ α≈i₁) = inj₁ (trans≈ α≈i₁ i₁≈i₂)              16
17  ≈∈ {I = ⟦ α -∞⟦} i₁≈i₂ (inj₂ α≺i₁) = inj₂ (≺-resp-≈₁ i₁≈i₂ α≺i₁)           17
```

The following property states that the inclusion is trivially transitive.

```
18  trans⊂ : Transitive _⊂ᵢ_                                       18
19  trans⊂ u v = v ∘ u                                             19
```

This last property states that an instant that is a member of an interval whose boundaries are coincident is coincident with these boundaries. This property will be especially useful when talking about the death of a clock in Section 6.2.2.e.

```
20  sameB : ∀ {i j k} → j ≈ k → i ∈ᵢ ⟦ j - k ⟧ → i ≈ j                          20
21  sameB j≈k = sym≈ ∘ (uncurry ≼→¬≺→≈) ∘ map₂ (_∘ (≺-resp-≈₂ j≈k))              21
```

This uses uncurry transform a function of two input into a function of a pair of inputs, and $\texttt{map}_2$ which applies a function to the second member of a couple.

6.2.1.b Subsets of instants

In order to define the clocks, which are the main bricks to express relations over event occurrences in CCSL systems, we must be able to express the subset of instants on which the clocks will tick. In type theories without subtypes, the notion of subset is not native but can be emulated and worked with. A subset is a set of values that are from a given type and that respect a given predicate. Since the predicate has a type, it is sufficient to represent the subset. By language abuse, a predicate over a type will be called a subset of this type. In the AGDA standard library, such a predicate is called a unary relation because it takes only one input, hence the unary relations are used to model subsets, similarly to binary relations which can be seen as subsets of a product type. To get familiar with subsets, we consider the example of the subset of even natural numbers.

We start by defining a predicate on natural numbers, inhabited by even numbers, as we did in Section 2.3.4.a:

```
1   data Even : ℕ → Set where                1
2     zpair : Even zero                      2
3     spair : ∀ {a} → Even a → Even (suc (suc a))   3
```

Then, we define a record named ∃ that is parametrized by a predicate and represents the subset induced by this predicate. As the notation implies, the logical correspondance of this record is the existential quantification – note that this definition can naturally be found in the standard library:

```
4   record ∃ {a b} {A : Set a} (P : A → Set b) : Set (a ⊔ b) where   4
5     constructor _,_ ; field                                         5
6       witness : A                                                   6
7       proof : P witness                                             7
```

From this definition, we can explicitly provide a definition for the subsets of naturals that are even:

```
8   Evenℕ : Set       8
9   Evenℕ = ∃ Even    9
```

Then, we give an example of a function that only takes even numbers as inputs then divides them by two, resulting in a number, which can be even or not. Note

that, as this is often the case when using dependent types, such functions have their error cases embedded in their type and are total in that regard. In this particular case, the input has to be even. To use this fact, we case split over the proof element of the input – the proof that the number is even – rather than over the number itself. There are two possible cases for the form of the proof, and AGDA deduces the form of the related number accordingly – we can see here that the number has been deduced to be either `zero` or a succession of at least two `suc` constructors.

```
10  [_/2] : Evenℕ → ℕ                                          10
11  [ .zero , zpair /2] = zero                                 11
12  [ .(suc (suc _)) , spair proof /2] = suc [ _ , proof /2]   12
```

Then, we define an equality between two elements of the previously defined record. Two elements are here declared equal when they embed the same witness, regardless of the proof they are coupled with. In our example, what interests us is that a given number is even, not that there exists several proofs of it. However, we will prove that any number can only have a single proof that it is even, which is usually convenient for such proof elements. This is the purpose of the remainder of the example.

```
13  _≈_ : ∀ {a b} {A : Set a} {P : A → Set b} → Rel (∃ P) _   13
14  _≈_ = _≡_ on ∃.witness                                     14
```

We proceed by proving that, in this specific case, there cannot exist two different proofs that a given number is even. This is often the case when defining such data structure as it is usually considered malformed when several successions of constructors can result in an element of the same type. But sometimes such ill-formed cases cannot be avoided, in which case such a proof could not be done.

```
15  uniqueEven : ∀ {n} (p₁ p₂ : Even n) → p₁ ≡ p₂                        15
16  uniqueEven zpair zpair = refl                                         16
17  uniqueEven (spair p₁) (spair p₂) = cong spair (uniqueEven p₁ p₂)      17
```

Finally, we prove that our equality between pairs implies the structural equality between these pairs when the predicate is our even predicate. The proof would be exactly the same whenever the underlying predicate satisfies the previous property. This concludes our proof that the witnesses alone are sufficient to prove equality as well as our example over even numbers. On the following development on CCSL, we will consider many subsets, and many such relations which concern witnesses and ignore the associated proof elements.

```
18  equality : ∀ {e₁ e₂ : ∃ Even} → e₁ ≈ e₂ → e₁ ≡ e₂                    18
19  equality {_ , p} {._ , p₁} refl rewrite uniqueEven p p₁ = refl        19
```

6.2.2 Clocks

6.2.2.a Informal definition

In CCSL, a clock is an entity that tracks the occurrences of a specific event in a given system. A clock ticks whenever (i.e. at every instant) the event it represents occurs. A system is represented by a set of clocks corresponding to any possible event that can occur during its execution. Each clock usually ticks an infinite number of times – can be both $\aleph_0$ (countable clocks representing discrete or dense time) or $\aleph_1$ (uncountable clocks representing only dense time) – and is partially represented in a timeline such as Figure 6.9. Discrete time means that, between two ordered instants, there always exists only a finite number of other instants. Dense time means that, between two ordered instants, there can exist an infinite number of other instants. For instance, $\mathbb{N}$ is countable and could be used to model discrete time, while $\mathbb{R}$ is uncountable and could model dense time. Note that $\mathbb{Q}$ is countable but would however be used to model dense time as well. In this example, the clock called `c` ticks three times during the portion of time depicted in the diagram. The ticks are separated by a given amount of time, unspecified – there is no scale on the diagram – because such a system is usually asynchronous. Thus, the only relevant information depicted in this diagram is that the event tracked by `c` occurred at least three times throughout the lifetime of the system. This is however very poor information which must be completed with the addition of other clocks and constraints between them to provide more data regarding the system behaviour.

Figure 6.9: An example of a clock c

6.2.2.b Formal definition

Formally, a clock is a type specification, which means an element of type $\texttt{Set}_1$ in AGDA. A clock is a set of instants which are totally ordered. This means that if we call `I` the set of instants and `C` the set of clocks then we have `C ⊆ P(I)` where `P(I)` is the powerset of `I`. What determines if a subset of instants is a clock is whether or not the instants it contains are totally ordered regarding the partial order existing on `I`. Using these observations, the notion of subsets in AGDA which was detailed in Section 6.2.1.b, the relations `_≡'_` and `_≺'_` which are presented in the Appendices in Section A.3.1 and which correspond respectively to the propositional equality and the precedence on the first member of a couple of values, and finally the notion of `Trichotomous` which corresponds to a total ordering of the instants, the specification of the clocks is as follows:

```
1  Clock₀ : Set₁                                                   1
2  Clock₀ = ∃ \Ticks → Trichotomous {A = ∃ Ticks} _≡'_ _≺'_        2
```

While this definition represents the clocks in their purest form, we do not use it directly in our work, for the simple reason that we want to have access to a specific constructor and specific field names for the constitutive elements of a clock, instead of the regular ones of the product type, namely $\texttt{proj}_1$, $\texttt{proj}_2$ and `_,_` which $\texttt{Clock}_0$ provides. In that purpose, we define a new specification of clocks which is strictly equivalent to $\texttt{Clock}_0$ while taking the form of a record. This definition is also more readable. The `Clock` record contains a constructor and two fields: a subset of instants (the ones on which the clock ticks), `Ticks`, and the proof that these instants are always comparable, `TTot`:

```
3  record Clock : Set₁ where                                   3
4    constructor _⧖_ ; field                                    4
5      Ticks : Pred Support _                                   5
6      TTot : Trichotomous {A = ∃ Ticks} _≡'_ _<'_              6
```

The syntactical correspondence between these specifications can be shown:

```
7  1⇒2 : Clock₀ → Clock                      7
8  1⇒2 (Ticks , TTot) = Ticks ⧖ TTot         8
```

```
9   2⇒1 : Clock → Clock₀                     9
10  2⇒1 (Ticks ⧖ TTot) = Ticks , TTot        10
```

As a first property about clocks, the definition imposes that two coincident ticks of the same clock are necessarily propositionally equal. Proving this statement is achieved by using the total order over the ticks of any clock. If one tick strictly precedes the others, there is a contradiction with the hypothesis, which is easily expressed using function composition `_∘_` and the constant function `const`.

```
11  ≈→≡ : ∀ {c} {i j : ∃ (Ticks c)} → i ≈' j → i ≡' j           11
12  ≈→≡ {c} {i} {j} with TTot c i j                              12
13  ≈→≡ | tri< a _ _ = ⊥-elim ∘ (<→¬≈ (inj₁ a))                  13
14  ≈→≡ | tri≈ _ b _ = const b                                   14
15  ≈→≡ | tri> _ _ c = ⊥-elim ∘ (<→¬≈ (inj₂ c))                  15
```

6.2.2.c Active and passive clocks

From the formal definition of the clock, we can derive predicates as to whether a clock ticks at least once or not, which we call `passive` and `active`. As `Empty` is defined in the standard library as a predicate which never holds, a passive clock is a clock whose subset of ticks is empty, while an active clock is the opposite.

```
16  passive : Pred Clock _          16
17  passive = Empty ∘ Ticks         17
```

```
18  active : Pred Clock _           18
19  active = ¬_ ∘ passive           19
```

These definitions are bricks on which more advanced ones can be expressed, starting with the notion of filtering.

6.2.2.d Filtering

We propose a way of filtering a clock through a given predicate, which creates a new clock that ticks on instants when both the original clock ticks and the predicate holds, thus ticking on their intersection if seen as subsets. This will be used for instance when considering properties over clocks on a specific interval inside the lifetime of the clock. The filtering is defined as follows:

```
20  _filterBy_ : Clock → (Pred _ _) → Clock                                  20
21  ((Ticks ⧖ _) filterBy P) .Ticks = Ticks ∩ P                               21
22  ((_ ⧖ TTot) filterBy _) .TTot (x , y , _) (x₁ , y₁ , _)                   22
23    = TTot (x , y) (x₁ , y₁)                                                23
```

Since this creates a new clock, it is required to compute the proof that the ticks remain comparable after the filtering, which is done on line 23. To define this filtering, we chose to rely on the notion of copatterns provided by AGDA. The dotted patterns – `.Ticks` and `.TTot` – are actually copatterns, which means they correspond to a subpart of the clock (yet another dot, different from inference and irrelevance from Section 2.3.5.e). The union of the copatterns must define the whole clock. This function is equivalent to the following one, without the use of copatterns:

```
24  _filterBy'_ : Clock → (Pred _ _) → Clock                                          24
25  (Ticks ⧖ TTot) filterBy' P                                                          25
26    = (Ticks ∩ P) ⧖ λ {(x , y , _) (x₁ , y₁ , _) → TTot (x , y) (x₁ , y₁)}            26
```

Copatterns are useful as a visual way of separating the constituents of the output of a function, as well as the bricks that are useful in building these constituents. For instant, the quantity `.Ticks` only needs `Ticks` and `P` to be built, which allows us to hide `TTot` using an underscore. This enforces a clearer view as to how each of the constituents of a clock are deduced from the context. However, using copattern usually induces a less concise definition. This is mostly a matter of taste and aesthetic views but it has other upsides which will be discussed in later uses.

Now that the filtering has been defined, we can formulate and prove properties regarding activity or passivity of filtered clocks:

- Filtering a passive clock can only produce a passive clock.
- If the filtering of a clock is active, the original clock was necessarily active.

```
27  passiveFilter : ∀ {c P} → passive c → passive (c filterBy P)      27
28  passiveFilter p x = p x ∘ proj₁                                   28
29  -                                                                 29
30  activeFilter : ∀ {c P} → active (c filterBy P) → active c         30
31  activeFilter {c} = _∘ (passiveFilter {c})                         31
```

As mentionned, filtering clocks is mostly used to restrict their lifetime to a specific interval, therefore we provide a special filtering which takes an interval as input. This filtering, called $_|_i_$ is defined as follows:

```
32  _|i_ : Clock → Interval → Clock                                       32
33  c |i I = c filterBy (_∈i I)                                            33
```

Pursuing the goal of specifying behaviour on specific interval, we define activity and passivity relatively to an interval. A clock `c` is said to be active (resp. passiv) on a given interval `I` when `c` $|_i$ `I` is active (resp. passive) as follows:

```
34  _ticksIn_ : Clock → Interval → Set                                     34
35  c ticksIn I = active (c |i I)                                          35
36  -                                                                      36
37  _idlesIn_ : Clock → Interval → Set                                     37
38  c idlesIn I = passive (c |i I)                                         38
```

A clock that is active in a specific interval `I` should be active in any interval containing `I`. Reciprocally, a clock that is passive in an interval `I` should be passive in any interval contained in `I`. This is expressed and proved as follows:

```
39  presIdles : ∀ {c I0 I1} → c idlesIn I1 → I0 ⊂i I1 → c idlesIn I0       39
40  presIdles passcI1 I0⊂iI1 i (ti , i∈I0) = passcI1 i (ti , I0⊂iI1 i∈I0)  40
41  -                                                                      41
42  presTicks : ∀ {c I0 I1} → c ticksIn I0 → I0 ⊂i I1 → c ticksIn I1       42
43  presTicks {c} p I0⊂iI1 passcI1 = p (presIdles {c} passcI1 I0⊂iI1)      43
```

6.2.2.e Lifetime of a clock

In the paper version of the denotational semantics of CCSL, the authors rely for each clock on two particular instants, the birth and the death. These instants are meant to represent the lifetime of a clock and are not represented as a constituent of the clocks in our semantic. Rather, they are represented as a predicate since we try to be as relational as possible. This means that, as it is customary in our work, a notion that was once represented as a specific element will now be expressed relationally. The most important predicate is `diesIn` which states if a clock dies in a given interval. For a clock to be considered as dying in a interval, this clock should both tick in this interval, and have all its subsequent ticks remain in this interval.

```
44  _diesIn_ : REL Clock Interval _                                        44
45  c diesIn I = ∃ λ i → i ∈i I × Ticks c i × (∀ j → Ticks c j → i ≼ j → j ∈i I)  45
```

From this definition, we can deduce a death instant. It is important to note that, according to this definition, a specific death instant might not exist, especially in the case of dense clocks. An instant is considered as the death instant of a clock if this clock dies in the interval reduced to this instant, as shown as follows:

```
46  _diesOn_ : REL Clock Support _                                        46
47  c diesOn i = c diesIn ⟦ i - i ⟧                                       47
```

According to the notion of death, all instants on which a specific clock dies should be coincident. This can be specified and proved as follows:

```
48  dies≈ : ∀ {i j c} → c diesOn i → c diesOn j → i ≈ j                  48
49  dies≈ {c = c} (_ , o∈ii , tco , _) (_ , u∈jj , tcu , _)               49
50    with sameB refl≈ o∈ii | sameB refl≈ u∈jj | TTot c (_ , tco) (_ , tcu)  50
51  dies≈ _ _ | o≈i | u≈j | tri≈ _ refl _ =                               51
52    trans≈ (sym≈ o≈i) (trans≈ refl≈ u≈j)                                52
53  dies≈ (_ , _ , _ , po) (u , _ , tcu , _) | o≈i | _ | tri< o≺u _ _ =   53
54    contradiction                                                       54
55      (sym≈ (trans≈ (sameB refl≈ (po u tcu (inj₂ o≺u))) (sym≈ o≈i)))    55
56      (≺→¬≈ (inj₁ o≺u))                                                 56
57  dies≈ (o , _ , tco , _) (_ , _ , _ , pu) | _ | u≈j | tri> _ _ u≺o =   57
58    contradiction                                                       58
59      (trans≈ (sameB refl≈ (pu o tco (inj₂ u≺o))) (sym≈ u≈j))           59
60      (≺→¬≈ (inj₂ u≺o))                                                 60
```

In order to conduct this proof, we start by retrieving additional information from the context on – line 50:

- The instant `o` on which `c` ticks in its death interval coincides with the unique boundary `i` of this interval. This means that `o ≈ i`.
- In a similar matter, we deduce that `u`, the instant on which `c` ticks in its other death interval coincides with its boundary, meaning `u ≈ j`.
- We know that there is a strict total order on the ticks of `c`. Since `c` ticks both on `o` and `u` we can compare these instants together which leads to three different cases: `o ≡ u`, `o ≺ u` and `u ≺ o`

The three branches of the proof derived from the comparison between `o` and `u` are conducted as follows:

- When `o ≡ u` – line 50 – we know that `o ≈ u` because `_≈_` is reflexive. We also know that `o ≈ i` and `u ≈ j` which leads to `i ≈ j` using the symmetry and transitivity of `_≈_`
- When `o ≺ u` – line 53 – we can exhibit a contradiction. On one side, this means that we have `¬ o ≈ u`. On the other side, using the property `po` we

know that any tick of `c` occuring after `o` is necessarily in the death interval, which is reduced to the value `i`. This means that `u` coincides with `i` and ultimately leads to `o ≈ u` using the other properties of `_≈_`.

- The case `u ≺ o` – line 57 – is conducted similarly to the previous one.

Assuming that a clock dies on an instant `d`, it should never tick in any interval starting after `d`, which can be proved as follows:

```
61  staysDead : ∀ {i d c} → c diesOn d → d ≺ i → c idlesIn ⟦ i -∞⟦                61
62  staysDead (_ , d∈xx , _ , p) d≺i j (tcj , i≼j) = proj₂                          62
63    (p j tcj (trans≼ (≼-resp-≈₂                                                    63
64    (sym≈ (sameB refl≈ d∈xx)) (inj₂ d≺i)) i≼j)) (trans≺≼ d≺i i≼j)                 64
```

While the death gives us valuable information about a single clock, we would like to introduce new notions allowing us to bind clocks with one another. This is done in CCSL by a concept called relations.

6.2.3 Relations

6.2.3.a Definition

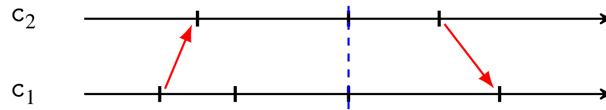


Figure 6.10: Some instants are constrained

In a complex and possibly heterogeneous system, many events – hence many clocks – can be identified. An important aspect of CCSL is that it handles complex and heterogeneous systems in a single manner (in a way, in CCSL, each system is heterogeneous compared to the atomistic description of each event it provides). Each clock taken separately does not offer much interesting information about the whole system, but bound together, they provide useful specifications about its global behaviour. This binding can be given as binary relations that constrain the execution of the system and, in our framework, is modeled as such (a binary relation is a predicate over two variables). They enforce an order between some instants by requiring some of them to be bound by precedence – red arrows – or by coincidence – dashed blue lines – as depicted in Figure 6.10. A relation holds, by default, for the lifetime of the system. The global AGDA type for CCSL relations is:

```
65  CCSLRelation : Set₁                                                              65
66  CCSLRelation = Rel Clock _                                                       66
```

`CCSLRelation` is a specification of a type – hence an element of type $\mathtt{Set}_1$ – which contains all the predicates over two clocks. This means that any binary relation over clocks (any set of couples of clocks) is a clock relation.

While relations can hold for the entire lifetime of two clocks, it is often very useful to specify a relation which only holds inside a specific interval. Considering a given interval, two clocks are related if they satisfy the relation whenever they tick inside that interval. In our framework, this is expressed this way: two clocks are considered related on a given interval when their filtered versions are related globally through the same relation. Thus, by restricting the "Ticks" predicate on our clock, we obtain a more natural version of interval constrained relations.

Let us consider the specification of a relation over a specific interval:

```
67  CCSLRelation_i : Set_1                                   67
68  CCSLRelation_i = Interval → CCSLRelation                 68
```

This allows, from a specific relation, to deduce the associated relation reduced to an interval:

```
69  toRel_i : CCSLRelation → CCSLRelation_i                   69
70  toRel_i _CR_ I = _CR_ on (_filterBy (_∈_i I))             70
```

While this definition of a relation inside a given interval is convenient, it might sometimes be useful to add an additional predicate while restricting a relation to a given interval. Indeed, while considering the filtering of a clock through the interval, we lose information about what happens outside this interval, and sometimes these pieces of information can be useful to express a more advanced property. We provide another way of building intervalled relations that accepts a predicate as input to handle these cases. Since this predicate happens to be over two clocks and an interval, it is itself of type $\mathtt{CCSLRelation}_i$.

```
71  toRel_iWithP : CCSLRelation → CCSLRelation_i → CCSLRelation_i             71
72  toRel_iWithP _CR_ CR_i I c1 c2 = toRel_i _CR_ I c1 c2 × CR_i I c1 c2      72
```

6.2.3.b Subclocking

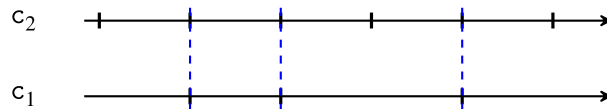


Figure 6.11: $\mathtt{c}_1$ is a subclock of $\mathtt{c}_2$

Definition The subclocking is the first and most natural relation provided by CCSL. A clock c_1 is said to be a subclock of a clock c_2 when every tick of c_1 is coincident with a tick of c_2. This is the classical sampling of synchronous systems. Figure 6.11 shows an example of subclocking.

The AGDA definition of this relation is as follows:

```
73  _⊑_ : CCSLRelation                                              73
74  (Tc₁ ⧖ _) ⊑ (Tc₂ ⧖ _) = ∀ (x₁ : ∃ Tc₁) → ∃ \(x₂ : ∃ Tc₂) → x₁ ≈' x₂   74
```

It states that whenever c_1 ticks on an instant x_1 – ∀ $(x_1 : \exists Tc_1)$ – there exists an instant x_2 on which c_2 ticks – ∃ \\$(x_2 : \exists Tc_2)$ – which coincides with x_1 – ≈' is the extension of ≈ as described in Section A.3.1.

_⊑_ and _≡_ form a preorder As a conformity property, one would expect such a subclocking to be transitive, which means that sampling a clock several times in a row would still result in a sample of the original clock. This is done as follows:

```
75  trans⊑ : Transitive _⊑_                                          75
76  trans⊑ c₁⊑c₂ _ x with c₁⊑c₂ x                                    76
77  trans⊑ _ c₂⊑c₃ _ | y , _ with c₂⊑c₃ y                            77
78  trans⊑ _ _ _ | _ , x≈y | z , y≈z = z , trans≈ x≈y y≈z            78
```

In addition, the subclocking should also be reflexive:

```
79  refl⊑ : Reflexive _⊑_                                            79
80  refl⊑ = _, refl≈                                                 80
```

Both these properties provide a convenient structure of partial ordering to the subclocking – relatively to the propositional equality. This structure can be made explicit in AGDA by providing the right elements as follows:

```
81  isPreorder≡⊑ : IsPreorder _≡_ _⊑_                                81
82  isPreorder≡⊑ = record {                                          82
83    isEquivalence = isEquivalence≡ ;                               83
84    reflexive = λ { {c} refl → refl⊑ {c}} ;                        84
85    trans = λ {c₁} {c₂} {c₃} → trans⊑ {c₁} {c₂} {c₃}}              85
```

It would be tempting to assume that these relations also exhibit a partial order structure, meaning that $\forall\ c_1\ c_2 \rightarrow c_1 \sqsubseteq c_2 \rightarrow c_2 \sqsubseteq c_1 \rightarrow c_1 \equiv c_2$. However, this does not hold because the propositional equality is too strong and the clocks c_1 and c_2 can have instants that are coincident but not equal. Another notion of equality betweens clocks will soon be described and will fulfill that role.

Relation with filtering By defining the notion of subclock, it is noticeable that we define a notion very close to filtering. In fact, the filtering of a clock is the operational equivalent of the subclocking. While the subclocking is a relation that can appear between two clocks, the filtering is an operation which creates a subclock of a given clock. While this correspondence seems natural, it is mandatory to prove that these notions are indeed two faces of the same coin.

The first step in that direction is to prove that the filtering of a clock is indeed a subclock of the original clock regarding the subclocking relation. This is very easy and natural to complete – `opToRel` stands for "operational to relational":

```
86  opToRel : ∀ {c P} → (c filterBy P) ⊑ c                    86
87  opToRel (i , tci , _) = (i , tci) , refl≈                   87
```

The second and last step is to prove that, considering two clocks in a relation of subclocking, the subclock could have been built from the original clock by filtering it through a given predicate. However, this step needs further work, for it requires us to specify a comparison between clocks. To understand where the issue comes from, let us try and formulate the property we wish to prove in natural language: "Given two clocks c_0 and c_1 such that c_0 is a subclock of c_1, there exists a predicate P such that c_1 filtered by P is in fact c_0". While reading this assertion, one can notice that the notion of "is in fact" is not quite well defined. Indeed, to prove such a property, one must be able to compare clocks one with the other. The first idea would be to use propositional equality. However, clocks are built from the `Ticks` predicate and these predicates can only be proved propositionally equal by postulating the extensional equality which we do not intend to do. Thus, there needs to be another equality between clocks, which would hold when the subsets of ticks contain the same elements. This leads to the definition of a new relation, called equality, which will ultimately lead to the proof of the reciprocal property that will be called `relToOp` (standing for relational to operational).

6.2.3.c Equality

Definition Two clocks c_1 and c_2 are equal when they only tick on coincident instants. It means that if c_1 ticks on i then there exists an instant j which coincides with i and where c_2 ticks, and reciprocally. An example of equal clocks is given in Figure 6.12.

Figure 6.12: c_1 is equal to c_2

One can notice that this definition is exactly equivalent to a double subclocking, which is a more natural way of defining it:

```
88  _∽_ : CCSLRelation
89  c₁ ∽ c₂ = c₁ ⊑ c₂ × c₂ ⊑ c₁
```

_∽_ is an equivalence relation Since this relation should be a reliable way to compare one clock to another, it should be an equivalence relation. Similarly to the preorder exhibited earlier, this is done by instantiating the right arguments inside a record which contains the fields required to be an equivalence relation.

```
90  isEq∽ : IsEquivalence _∽_
91  isEq∽ .IsEquivalence.refl = (_, refl≈) , _, refl≈
92  isEq∽ .IsEquivalence.sym = swap
93  isEq∽ .IsEquivalence.trans {i} {j} {k} (i⊑j , j⊑i) (j⊑k , k⊑j) =
94    trans⊑ {i} {j} {k} i⊑j j⊑k , trans⊑ {k} {j} {i} k⊑j j⊑i
```

_⊑_ and _∽_ form a partial order Since we exhibited a second equivalence relation between clocks – in addition to the propositional equality _≡_ – we can prove that the subclocking remains a preorder with that new equivalence relation. But, since this notion of equality is more permissive than the propositional equality, we can also exhibit a partial order. Filling these records is interesting because it allows us to grab all the predefined consequences present in the standard library implemented from the state of the art in mathematical structures. This allows us to show that copatterns can be chained.

```
95  isPartialOrder∽⊑ : IsPartialOrder _∽_ _⊑_
96  isPartialOrder∽⊑ .isPreorder .IsPreorder.isEquivalence = isEq∽
97  isPartialOrder∽⊑ .isPreorder .IsPreorder.reflexive = proj₁
98  isPartialOrder∽⊑ .isPreorder .IsPreorder.trans {c₁} {c₂} {c₃} =
99    trans⊑ {c₁} {c₂} {c₃}
100 isPartialOrder∽⊑ .antisym = _,_
```

A trivial implication is that if two clocks are equal, and one of them is a subclock of a clock c then the other is also a subclock of c. This problem can be solved in a domain manner with obvious considerations, but it can also be treated as a purely structural problem, solving it using the reflexivity and the transivity of the previous preorder.

Filtering, subclock and equality Now that we have defined a notion of equality between clocks, we can complete our relation between the operational filtering and the relation subclocking. The property which cannot be formally expressed, and which was the following informal sentence: "Given two clocks c_0 and c_1 such as

c_0 is a subclock of c_1, there exists a predicate P such as c_1 filtered by P is in fact c_0" can now be successfully expressed:

```
101  relToOp : ∀ {c₀ c₁} → c₀ ⊑ c₁ → ∃ (_∽ c₀ ∘ c₁ filterBy_)          101
```

In order to prove this property, we need to assess P. P should be a predicate such as the filtering of c_1 through P is equal to c_0. Since c_0 is a subclock of c_1, we need to filter the instants on which c_1 ticks that have a coincident instant on which c_0 ticks, which gives us P. Having expressed such property, the proof follows:

```
102  relToOp {c₀} {c₁} c₀⊑c₁ =                                                   102
103    (λ i → ∃ \j → Ticks c₀ j × i ≈ j) ,                                        103
104    (λ {(_ , _ , j , tc₀j , i≈j) → (j , tc₀j) , i≈j}) ,                        104
105    λ {(i , tc₀i) → case c₀⊑c₁ (i , tc₀i) of                                   105
106      λ {((j , tc₁j) , i≈j) → (j , tc₁j , i , tc₀i , sym≈ i≈j) , i≈j}}         106
```

This completes the relation between filtering and subclocking as the two faces of the same coin.

6.2.3.d Exclusion

Definition Two clocks are in exclusion with one another when they have no coincident ticks. An example of exclusion is given on Figure 6.13.

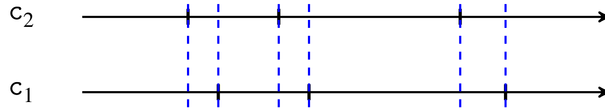


Figure 6.13: c_1 is in exclusion with c_2

The AGDA definition is the following:

```
107  _♯_ : CCSLRelation                                                            107
108  (Tc₁ ⧖ _) ♯ (Tc₂ ⧖ _) = ∀ (x : ∃ Tc₁) (y : ∃ Tc₂) → ¬ x ≈' y                 108
```

This definition consists of a predicate that for any x and y, if c_1 ticks on x and c_2 ticks on y, then x and y are not coincident.

Symmetry of _♯_ As a direct consequence to this definition and the symmetry of the coincidence, the exclusion is symmetrical.

```
109  sym♯ : Symmetric _♯_                                                          109
110  sym♯ p x y = (p y x) ∘ sym≈                                                   110
```

Relation to subclocking If two clocks are in exclusion, then sampling any or both of these clocks should retain this property.

```
111  ♯⊑ : ∀ {c₁ c₂ c₃ c₄} → c₂ ♯ c₄ → c₁ ⊑ c₂ → c₃ ⊑ c₄ → c₁ ♯ c₃              111
112  ♯⊑ c₂♯c₄ c₁⊑c₂ c₃⊑c₄ i j i≈'j with c₁⊑c₂ i | c₃⊑c₄ j                    112
113  ♯⊑ c₂♯c₄ _ _ _ _ i≈j | k , i≈k | l , j≈l =                            113
114    c₂♯c₄ k l (trans≈ (sym≈ i≈k) (trans≈ i≈j j≈l))                       114
```

Note that since the subclocking is reflexive, this property encompasses the case where only one of the clocks are sampled.

We also provide a proof that if two clocks are both in a relation of subclocking and in a relation of exclusion, then the subclock is passive.

```
115  ♯×⊑→⊥ : ∀ {c₁ c₂} → c₁ ♯ c₂ → c₁ ⊑ c₂ → passive c₁                     115
116  ♯×⊑→⊥ c₁♯c₂ c₁⊑c₂ i tc₁i with c₁⊑c₂ (i , tc₁i)                         116
117  ♯×⊑→⊥ c₁♯c₂ c₁⊑c₂ i tc₁i | j , i≈j = c₁♯c₂ (i , tc₁i) j i≈j           117
```

6.2.3.e Precedence

Informal definition A clock c_1 precedes another clock c_2 when each consecutive tick of c_2 is preceded by a distinct and consecutive tick of c_1. Note that the word "consecutive" can only refer to discrete clocks. In dense clocks, the equivalent is that every tick of c_1 placed between two mapped ticks must be mapped as well. This mapping refers to a function that binds the instants of the two clocks together so that the precedence holds. The precedence can either be strict or not, depending if two mapped instants are allowed to be coincident. Before getting to the formal definition of this relation, let us consider some examples which will clarify the notion of "consecutive" as well as the difference between strict and non-strict precedence.

Examples Figure 6.14 represents a mapping between two clocks c_1 and c_2 where each tick of c_2 is mapped to a tick of c_1 that precedes it. However, there are "unused" ticks of c_1 which should be mapped to some ticks of c_2. This example was mistakenly authorized by the paper version of CCSL denotational semantics and was corrected in our mechanization.

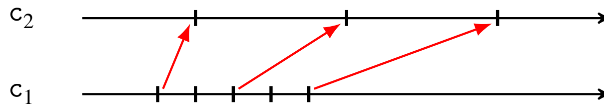


Figure 6.14: An incorrect strict precedence example

Figure 6.15 corrects the issue by mapping the remaining instants in a correct manner. One can notice that it should be possible to go from one such unsound mapping to this sound one in an operational manner by "moving" the arrows to the

left. However, this process, while possible when considering discrete clocks, is not when dealing with dense clocks where there is an infinite amount of "moving" steps to achieve, which is why we embed this constraint directly in our definition.

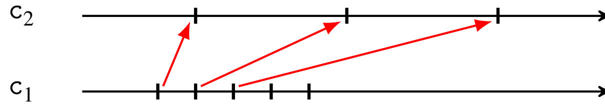


Figure 6.15: A standard strict precedence example

Figure 6.16 represents another example of precedence, where the mapping is actually bijective, which means that the faster clock (c_1) ticks as many times as the slower one (c_2) . While interesting, this is not required in the definition of precedence. However, the mapping function is always bijective when considering the subset of instants it reaches, which will be proved later on.

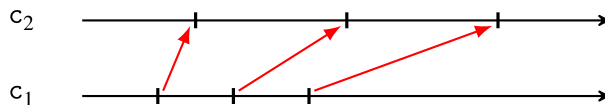


Figure 6.16: A specific strict precedence example

As a last example, Figure 6.17 shows an example of non-strict precedence, where two mapped instants are coincident. If all mapped instants are coincident, and if these mapped instants are consecutive, this is both a case of non-strict precedence, and subclocking.

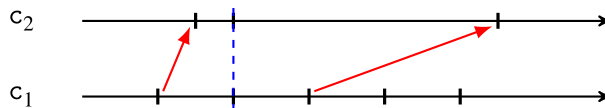


Figure 6.17: An example of non-strict precedence

Formal definition The precedence relation requires the existence of a function `h` which maps the instants of c_2 with the corresponding instants of c_1. While we chose to define it as a record to better separate its constituents, it could have equally been defined as a simple predicate. However, this allows us to add the property `preserves←` to our record which will ultimately be used as if it was a field except it is actually deduced from the fields[2]. This record is parametrized by a relation, which will later be instantiated either by the strict or non-strict precedence between instants. It is interesting to note that this relation is mostly irrelevant for the structure of this record, because it only constrains the position of the image compared to the

[2]This is possible thanks to AGDA expanding records as modules.

antecedent. This is why it appears only once in the record, in the definition of `precedes`. The other fields, which ensure that the instants are not entwined and are all mapped, do not depend on it.

```
118  record Precedence (_<_ : Rel Support _) (c₁ c₂ : Clock) : Set where
119    field
120      h : ∃ (Ticks c₂) → ∃ (Ticks c₁)
121      congruent : ∀ {i j} → i ≈' j → h i ≈' h j
122      precedes : ∀ i → proj₁ (h i) < proj₁ i
123      preserves : ∀ {i j} → i ≺' j → h i ≺' h j
124      dense : ∀ {i j} {p : ∃ (Ticks c₁)} →
125        h i ≺' p → p ≺' h j → ∃ (_≈' p ∘ h)
126    -
127    preserves← : ∀ {i j} → h i ≺' h j → i ≺' j
128    preserves← {i} {j} with TTot c₂ i j
129    preserves← | tri< a _ _ = const a
130    preserves← | tri≈ _ b _ = ⊥-elim ∘ (irrefl≈≺ ∘ congruent) (≡→≈ b)
131    preserves← | tri> _ _ c = ⊥-elim ∘ (≼→¬≺ ∘ inj₂ ∘ preserves) c
```

This definition is composed of five fields and one additional property:

- Line 120 – `h` maps the instants of the two clocks together. `h` goes from the ticks of the slower clock to the ticks of the faster clock to handle cases when the faster clock has additional ticks which should not be mapped.
- Line 121 – `congruent` ensures that two coincident ticks are mapped to coincident ticks as well. Since we know that coincidence on ticks of a given clock is equivalent to propositional equality, this means that `h` is congruent.
- Line 122 – `precedes` ensures the precedence between the two clocks. This is the main aspect of the precedence relation regarding common sense.
- Line 123 – `preserves` ensures that the mapping does not entwine instants together. Note that this property could also have been expressed using `_≼_` because the ticks are strictly ordered on a given clock.
- Line 124 – `dense` ensures that no instants are left behind between two mapped instants – regardless if the clocks are dense themselves or not. Once again, this definition relies on `_≈_` but could also have relied on `_≡_` without changing the semantics because these two are equivalent over ticks of the same clock.
- Line 127 – `preserves←` provides a proof of the reciprocal nature of `preserves` deduced from the axioms in this record.

From this definition we can extract the definition of strict and non-strict precedence, as follows:

```
132  _≺≺_ : CCSLRelation                   132
133  _≺≺_ = Precedence _≺_                133
```

```
134  _≼≼_ : CCSLRelation                   134
135  _≼≼_ = Precedence _≼_                135
```

Bijectivity of `h` The binding function `h` , which is the main part of the precedence relation, exhibits special properties. Indeed, thanks to the elements `compare`, `congruent` and `preserves`, `h` can be proven injective at first then bijective when considering its image subset of elements, that is the elements that `h` reaches. To prove such a property, it is required to work with setoids which are types coupled with an equivalence relation. These proofs are left for the appendices as explained as follows. The first step consists in proving that any injective function can be converted into a bijective function when considering its image setoid, which is displayed in Section A.4.1. The second step consists in proving that, thanks to the hypotheses present in the precedence structure, the binding function is injective after which the results of the first step can be applied on it, which is displayed in Section A.4.2.

Transitivity The precedence should be transitive, which is proven using copatterns to physically isolate the required fields of the output. This proof requires the underlying relation to be transitive.

```
136  tprec : ∀ {R} → Transitive R → (Transitive ∘ Precedence) R                        136
137  tprec _ p₁₂ p₂₃ .h = (h p₁₂) ∘ (h p₂₃)                                             137
138  tprec _ p₁₂ p₂₃ .congruent = (congruent p₁₂) ∘ (congruent p₂₃)                      138
139  tprec tr p₁₂ p₂₃ .precedes = (tr (precedes p₁₂ _)) ∘ (precedes p₂₃)                 139
140  tprec _ p₁₂ p₂₃ .preserves = (preserves p₁₂) ∘ (preserves p₂₃)                      140
141  tprec _ p₁₂ p₂₃ .dense {p = k} ≺k k≺ with dense p₁₂ {p = k} ≺k k≺                   141
142  tprec _ p₁₂ p₂₃ .dense ≺k k≺ | l , hl≈p with dense p₂₃ {p = l}                       142
143    (preserves← p₁₂ (≺-resp-≈₁ (sym≈ hl≈p) ≺k))                                       143
144    (preserves← p₁₂ (≺-resp-≈₂ (sym≈ hl≈p) k≺))                                       144
145  tprec _ p₁₂ p₂₃ .dense _ _ | m , hm≈p | n , hn≈p =                                  145
146    n , trans≈ (congruent p₁₂ hn≈p) hm≈p                                              146
```

In this proof, copatterns yield an additional upside on top of aesthetic considerations. The computation of the `dense` field requires the addition of new quantities on which to pattern match using `with` when the computation of the other fields do not. The use of copatterns allows us to do this addition specifically for this field without impacting the others as shown on lines 141 to 146. This proof would still be doable without copatterns – using the `case_of_` AGDA construct to allow in-term case-splitting – but the result would be far less readable.

While the fields `h`, `congruent`, `preserves` and `precedes` are deduced straightforwardly from the context, the proof that `dense` is preserved requires additional

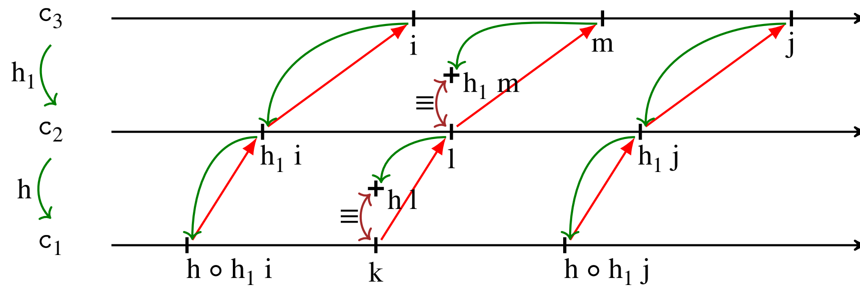


Figure 6.18: Transitivity of `dense`

work. We need to apply the `dense` properties from both inputs consecutively which provides the right instant needed for the proof to be completed. This proof is roughly represented in Figure 6.18: the green arrows depict the application of the binding functions, while the double brown arrows depict propositional equality.

Since both the precedence and the strict precedence over instants are transitive, their corresponding precedence and strict precedence over clocks embed this property.

```
147  trans≺≺ : Transitive _≺≺_      147
148  trans≺≺ = tprec trans≺          148
```

```
149  trans≼≼ : Transitive _≼≼_      149
150  trans≼≼ = tprec trans≼          150
```

Properties between _≺≺_ and _∽_ The strict precedence between clocks can instinctively be seen as a way of strictly ordering clocks with one another. However, it remains unclear under which conditions this order should hold.

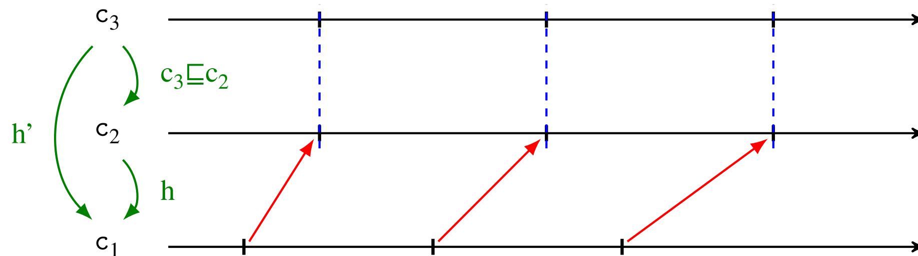


Figure 6.19: Proof that the equivalence classes are respected

This potential order is necessarily strict, because a clock should not strictly precede itself, and it is also necessarily partial, because two clocks are not necessarily in a relation of strict precedence with one another. This means that if _≺≺_ and _∽_ form an order, it is a strict partial order. Since we already know that _∽_ is an equivalence relation, and that _≺≺_ is transitive, there are two remaining properties to be proven to reach a strict partial order. These are the irreflexivity

of _≼≼_ towards _∽_ and the fact that _≼≼_ respects the equivalence classes induced by _∽_. According to the definitions, the preservation of the equivalence classes should hold instinctively by combining the mapping function with the one-one mapping of the instants of the two clocks deduced from the equality between these clocks. As shown in Figure 6.19, it means that `h' ≡ h ∘ (`$c_3 \sqsubseteq c_2$`)`.

The respect of the equivalence classes has been proven in both cases, but the proofs are too dense and are put in the appendices, they can be found in Section A.4.3. Here are their signatures:

```
151  ≺≺-resp^r-∽ : _≺≺_ Respects^r _∽_                         151
152  ≺≺-resp^l-∽ : _≺≺_ Respects^l _∽_                         152
```

However, it is noticeable that the irreflexivity cannot be proven, because it unfortunately does not hold in several cases. It is interesting to investigate these cases and try to exhibit a common factor which would explain the issue, as our instinct dictates that this should hold in "regular" cases, hence the need to rigorously define what is meant by this. A first example of non-irreflexivity is in the case of a passive clock. If the clock never ticks, then anything is definitionaly slower, including itself.

```
153  c≺≺∅ : ∀ {c e} → passive e → c ≺≺ e                                  153
154  c≺≺∅ p = record                                                       154
155    { h = λ {(i , ti) → ⊥-elim (p i ti)}                                155
156    ; congruent = λ { {i , ti} _ → ⊥-elim (p i ti)}                     156
157    ; precedes = λ {(i , ti) → ⊥-elim (p i ti)}                         157
158    ; preserves = λ { {i , ti} _ → ⊥-elim (p i ti)}                     158
159    ; dense = λ { {i , ti} _ → ⊥-elim (p i ti)}}                        159
160  -                                                                     160
161  ∅≺≺∅ : ∀ {e} → passive e → e ≺≺ e                                     161
162  ∅≺≺∅ = c≺≺∅                                                           162
```

This means that the faster clock has to be active for the precedence to be irreflexive towards equality. This makes sense because the binding function should not be empty to have a real meaning as a binder between ticks. However, this is not sufficient. Another case of non-irreflexivity can be provided when instantiating CCSL over integers. When doing so, the simplest clock of all, the one that always ticks, happens to precede itself by the simple function `x ↦ x - 1`, as shown in Figure 6.20. This examples has been developed in our framework and is presented in the appendices Section A.4.4.

The issue with this example is that the slower clock ticks infinitely often towards the past, which means there is no starting point to the precedence, which makes it unsound. Should we provide a starting point, a contradiction should arise. Indeed, given an instant `i` on which c_2 ticks for the first time of its lifetime, there exists an instant `j` which precedes `i` on which c_1 ticks, thanks to the precedence binding

these two clocks. However, c_1 and c_2 are equal, which includes c_1 being a subclock of c_2. This leads to the existence of an instant `k` on which c_2 ticks that is coincident with `j`. Since `j` strictly precedes `i` and `k` is coincident to `j`, `k` also strictly precedes `i` which is in contradiction with the fact that `i` is the first tick of c_2. We now have our condition : if the slower clock has a first tick, such ill-formed cases should not arise. This is a convenient condition because it also takes care of the empty clock unsound case since having a first tick implies having at least one tick.

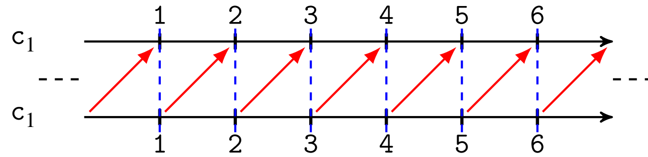


Figure 6.20: This clock strictly precedes itself

There remains the need to prove such assumption. To achieve that, we start by creating a new kind of clocks, these which have an initial instant. The initial instant is a pair of two elements : the tick instant – that is, an element of type `∃ Ticks` – and the proof that any other tick is preceded by it.

```
163  record InitialClock : Set₁ where                                          163
164    field                                                                   164
165      clock : Clock                                                         165
166      first : ∃ λ (x : ∃ (Ticks clock)) → ∀ (y : ∃ (Ticks clock)) → x ≼' y  166
```

Then, we extend our relations to initial clocks by applying the existing ones to the clock component of the initial clocks.

```
167  _∽ᵢ_ : Rel InitialClock _      167
168  _∽ᵢ_ = _∽_ on clock            168
```

```
169  _≺≺ᵢ_ : Rel InitialClock _     169
170  _≺≺ᵢ_ = _≺≺_ on clock          170
```

And finally, using the existence of this first instant, we can implement our informal proof of irreflexivity in our formal context:

```
171  ≺≺-irrefl : Irreflexive _∽ᵢ_ _≺≺ᵢ_                                    171
172  ≺≺-irrefl {y = c₂} _ _ with first c₂                                  172
173  ≺≺-irrefl c₁∽c₂ c₁≺≺c₂ | f , f≼x with proj₁ c₁∽c₂ (h c₁≺≺c₂ f)        173
174  ≺≺-irrefl _ c₁≺≺c₂ | f , f≼x | g , hf≈g =                             174
175    irrefl≈≺ hf≈g (trans≺≼ (precedes c₁≺≺c₂ f) (f≼x g))                 175
```

Note that this proof only requires the slower clock to have this first instant, but we provide it in a context where both clocks do have this instant, because it is much more convenient to work with homogeneous relations instead of heterogeneous relations.

Finally, combining all the elements that are required for a strict partial order to hold, we can build the proof that _≺≺_ and _∽_ form this kind of order on initial clocks. Note that, in these kinds of instantiations, and for a reason currently unknown to me, this field of transitivity always requires us to provide explicitly all of its implicit parameters, which is why it is so verbose.

```
176 ≺≺-ispo : IsStrictPartialOrder _∽ᵢ_ _≺≺ᵢ_                          176
177 ≺≺-ispo .isEquivalence =                                              177
178   record { refl = (_ , refl≈) , _ , refl≈ ; sym = swap ;              178
179     trans = λ { {record {clock = i}}                                  179
180       {record {clock = j}} {record {clock = k}}                       180
181       (i⊑j , j⊑i) (j⊑k , k⊑j) → trans⊑ {i} {j} {k} i⊑j j⊑k            181
182       , trans⊑ {k} {j} {i} k⊑j j⊑i} }                                 182
183 ≺≺-ispo .irrefl {i} {j} = ≺≺-irrefl {i} {j}                           183
184 ≺≺-ispo .trans = trans≺≺                                              184
185 ≺≺-ispo .<-resp-≈ = ≺≺-respʳ-∽ , ≺≺-respˡ-∽                           185
```

To summarize this part, the equality and the strict precedence between clocks have the right properties when the clocks have an initial instant, which can be interpreted as the birth instant of the clock. This is a reasonable prerequisite because it seems that in real life systems such an instant will indeed always exist, assuming the clock is not passive the entirety of its lifetime.

Properties between _≼≼_ and _∽_ Similarly to strict precedence, non-strict precedence can be seen as a way of ordering clocks one with the other. Since the precedence is by definition non-strict, the order it forms with the equality is not strict as well. In addition, since there is no reason that two random clocks would necessarily be bound through a non-strict precedence relation, this order should be partial. Thus, we aim at proving that _≼≼_ and _∽_ form a partial order.

The first step in that direction is to establish that _≼≼_ and _∽_ form a pre-order. Since the transivity of _≼≼_ and the equivalence of _∽_ have already been confirmed, it remains to prove that _≼≼_ is reflexive towards _∽_, which means that any two clocks that are equal are in a relation of non-strict precedence. This requires us to use the correspondance between the coincident instants of the two clocks as binding functions.

An interesting aspect in following proof of reflexivity is the field `dense`. It is interesting to notice that all the other fields, except this one, need the proof element associated with the variable $c_2 \sqsubseteq c_1$ to be completed but not the one associated with $c_1 \sqsubseteq c_2$. This means that, if the `dense` property was missing from the definition of precedence – which was originally the case in the paper version of the semantics – any subclock of a clock `c` – and not only any clock equal to `c` – would also non-strictly precede `c` which is not an expected behavior. The addition of `dense` ensures that no such case is possible which provides the expected semantics.

```
186  refl≼≼ : _∽_ ⇒ _≼≼_                                           186
187  refl≼≼ {c₁} (c₁⊑c₂ , c₂⊑c₁) = record {                        187
188    h = proj₁ ∘ c₂⊑c₁ ;                                          188
189    congruent =                                                  189
190      flip trans≈ (proj₂ (c₂⊑c₁ _)) ∘                            190
191      (trans≈ ((sym≈ ∘ proj₂ ∘ c₂⊑c₁) _)) ;                      191
192    precedes = inj₁ ∘ sym≈ ∘ proj₂ ∘ c₂⊑c₁ ;                     192
193    preserves = ≺-resp-≈₂ ((proj₂ ∘ c₂⊑c₁) _) ∘                  193
194      (≺-resp-≈₁ ((proj₂ ∘ c₂⊑c₁) _)) ;                          194
195    dense = λ {_} {_} {p} _ _ → proj₁ (c₁⊑c₂ p) ,                195
196      (trans≈ (sym≈                                              196
197      ((proj₂ ∘ c₂⊑c₁ ∘ proj₁ ∘ c₁⊑c₂) p))                        197
198      ((sym≈ ∘ proj₂ ∘ c₁⊑c₂) p))}                               198
```

Combining the elements proved earlier, the non-strict precedence forms a pre-order when coupled with clock equality.

```
199  isPreorder≡≼≼ : IsPreorder _∽_ _≼≼_                           199
200  isPreorder≡≼≼ = record {                                       200
201    isEquivalence = isEq∽ ; reflexive = refl≼≼ ; trans = trans≼≼ }  201
```

The next step is to assess whether this preorder can be completed to a partial order. The remaining property, to achieve the completion, is that _∽_ and _≼≼_ should be antisymmetric. This means that if a clock c_1 precedes a clock c_2 and vice-versa, these two clocks should be equal. Instinctively, this should hold and it fits the commonly accepted notion of precedence amongst CCSL experts and users. However, there are cases where this simply does not hold. As opposed to the unsound cases regarding strict precedence, there are no problems with passive clocks because the only clock that can be both faster and slower than the empty clock is another empty clock, in which case they are indeed equal. However, there are unsound cases similar to the `x → x -1` example on strict precedence, that can be expressed once again on $\mathbb{Z}$. Consider the clock c_1 that ticks on every even number, and the clock c_2 that ticks on every odd number. These clocks have no coincident instant, and are thus definitely not equal, however they both precede the other clock through the same binding function, `x → x - 1`, as shown on Figure 6.21.

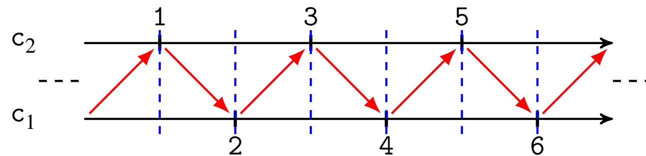


Figure 6.21: Two non-equal clocks with both precedences

This example has been implemented in our framework and is displayed in the appendices Section A.4.5. In order to prevent such a case, it is required to find a property that should be satisfied by the clocks in order to ensure the antisymmetry of `_∽_` and `_≼≼_`. A good way to assess such a property is to picture what the proof would look like and what would be the missing elements to complete it. As opposed to the irreflexivity of `_∽_` and `_≺≺_` where a contradiction had to be found, which means only considering a single instant was sufficient – the first instant – the antisymmetry requires us to build the entire proof of equality. Since two clocks are equal when all of their respective instants are coincident, the proof that has to be provided needs to reason about a potential infinity of instants. This already points us in the direction of a recursive proof over the set of instants of a clock. In order to succesfully define a recursion over an element of a type which is unspecified – where no structural recursion is possible – there needs to be a well-founded order. This leads to the conclusion that in order to establish the antisymmetry of `_∽_` and `_≼≼_`, `_≺_` needs to be a well-founded order on the subset of ticks of the clocks. From this point on, it will be possible to recurse over the ticks of the clocks to ensure that all of their ticks are indeed coincident with a tick of the other clock.

The complete proof is available in the appendices in Section A.4.6, however, here are the different steps that it contains:

1. We defined a new kind of clock which provides a proof that the precedence relation is well-founded over its ticks.

2. We extend the relations of strict precedence and equality to this new kind of clock, considering only the clock it contains.

3. We formulate a predicate which implies that the antisymmetry holds. This predicate states that any instant `i` of c_1 – c_1 being one of the two clocks involved in the double precedence – is coincident with $(h_1 \circ h)$ `i` – we call this predicate `P` of `i` – where `h` and h_1 are the binding function of the precedences. We picked an instant of c_1 to express this property but we could have equally chosen c_2 since the two clocks can be interchanged indifferently.

4. We prove that this predicate implies the antisymmetry of `_∽_` and `_≼≼_`.

5. In order to prove `P` for all ticks, we formulate a recurrence predicate which proves `P` for a given `i` provided that `P` holds for any tick that precedes `i`.

6. We use the well-founded order between the ticks to give `step` to a recursor which proves `P` for the entire subset of ticks.

7. We prove the different lemmas used to prove the step property.

8. This completes the proofs.

All of these steps allow us to prove the expected partial ordering by combining the antisymmetry with the properties that were previously expressed:

```
202 ≼≼-ipo : IsPartialOrder _∽ₒ_ _≼≼ₒ_                                   202
203 ≼≼-ipo = record {                                                  203
204   isPreorder = record {                                            204
205     isEquivalence = record {                                       205
206       refl = (_, refl≈) , _, refl≈ ;                               206
207       sym = swap ;                                                 207
208       trans = λ { {record {clock = i}} {record {clock = j}}        208
209         {record {clock = k}} (i⊑j , j⊑i) (j⊑k , k⊑j) →             209
210         trans⊑ {i} {j} {k} i⊑j j⊑k , trans⊑ {k} {j} {i} k⊑j j⊑i } ; 210
211     reflexive = refl≼≼ ;                                           211
212     trans = trans≼≼ } ;                                            212
213   antisym = λ {i} {j} → ≼≼-antisym-∽ {i} {j} }                     213
```

Application to natural numbers The natural numbers have convenient properties which allows us to have very weak conditions which are enough to ensure the existence of an initial tick and the well-foundedness of the strict precedence over its ticks. These conditions are the following :

- The clocks tick at least once.
- Their `Ticks` predicate is decidable.

Any clock that has these properties can be transformed into an element of type `InitialClock` or `WfClock`. This is proved in the appendices Section A.4.7 using the following steps:

1. Definition of a function `precedent` which, given a specific tick `i` of a clock `c` provides :
   - Either a proof that no instant preceding `i` is a tick of `c`, thus `i` is the initial instant.
   - Or an instant `j` on which `c` ticks which stricty precedes `i` with the proof that no other instant between `i` and `j` is a tick of `c`.

   This function recursively trims its input until either a new tick is found using the decidablity of the predicate, or zero is reached.

2. Proof of the well-foundedness of the strict precedence over the ticks of `c`. To prove that any tick is accessible – which is the definition of well-foundedness – we use the function `precedent` to step back and recursively build the result while proving the termination using the well-foundedness of the strict precedence on natural numbers.

3. Proof of the existence of an initial instant, applying recursively `precedent` from an existing instant which has to be provided.

4. Definition of a new kind of clocks containing a clock, a tick for this clock and the proof of decidability of its `Ticks` predicate.

5. Transformation of any clock of this type to an element of type `InitialClock` or `WfClock`.

6.2.3.f Alternation

Informal definition and examples There are some cases where precedence is not enough to express the semantics of the relation between two clocks. For instance, in Figure 6.22, the clock c_1 ticks a third time before the clock c_2 ticks a second time, which we might want to avoid.

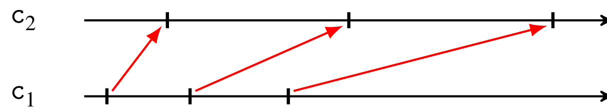


Figure 6.22: A case of precedence without alternation

There are some cases where this kind of behaviour might be unwanted and must be forbidden accordingly, forcing the clocks to be further constrained. Alternation is a stronger case of precedence, which ensures that the ticks of the two clocks alternate with one another. In other words, two clocks are said to be alternated when one precedes the other in such a way that two ticks of a clock cannot occur in between two ticks of the other one. Note that the underlying precedence has to be strict for the relation to be consistent. A non-strict precedence would lead to ill formed cases of alternation. Figure 6.23 presents a case of alternation.

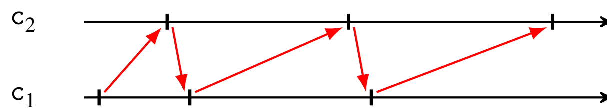


Figure 6.23: c_1 alternates with c_2

Formal definition Alternation is a case of strict precedence with the addition of a predicate that forces this alternation between instants. Formally, it is defined as a record which encapsulates the precedence while adding this predicate.

```
214 record _≪≫_ (c₁ c₂ : Clock) : Set where                          214
215   field                                                            215
216     precedence : c₁ ≺≺ c₂                                          216
217     alternate : ∀ {i j : ∃ (Ticks c₂)} → i ≺' j → i ≺' h precedence j   217
```

The predicate called `alternate` ensures the alternation by forcing, for any tick `i` of c_2, that the image of `i` by `h` precedes any subsequent tick of c_2.

6.2.4 Expressions

6.2.4.a Definition

CCSL allows the definition of new clocks from existing clocks, which is acceptable from an operational point of view. Creating new clocks usually sets an arbitrary order between the instants on which the underlying clocks are ticking, which means that instants apparently independent are getting related because a new clock is created out of them. An example of the arbitrary ordering is the union. The union of two clocks ticks whenever one of the two clocks ticks. Since a clock has a total order on its ticks, the ticks of the union must be totally ordered, which leads to a total order on the ticks of the two other clocks. In our denotational framework, all clocks are pre-existing thus we cannot create such new clocks. We assume they already exist and propose to relate them using predicates to state that a clock could be the result of such operation. To illustrate this notion, let us take the example of the addition between natural numbers. One can say that 3 is the result of the operation 1 + 2 while another point of view could be that the triplet (2,1,3) is a member of the addition. We take the second point of view to better match the denotational aspect of our work. The type of expression is thus defined as a relation between three clocks:

```
218  CCSLExpression : ∀ {a} → Set _                            218
219  CCSLExpression {a} = Clock → Clock → Clock → Set a         219
```

6.2.4.b Intersection

Informal Definition A common expression on clocks is the intersection. Informally, the clock c can be seen as the intersection of two clocks c_1 and c_2 when it only ticks on each instant where they simultaneously tick while each of its ticks has a counterpart in both of the intersected clock. This notion of intersection corresponds to the common definition of intersection over the subsets of instants.

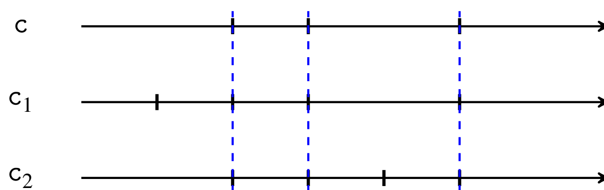


Figure 6.24: An example of intersection

Example Figure 6.24 shows an example of intersection, where the two clocks c_1 and c_2 have three coincident ticks, which are also ticks for their intersection c.

Formal definition Formally, the intersection is represented as the following CCSL expression:

```
220 _≈_⋂_ : CCSLExpression                                                    220
221 (Tc ⧖ _) ≈ Tc₁ ⧖ _ ⋂ (Tc₂ ⧖ _) =                                          221
222   (∀ (x : ∃ Tc) → ∃ λ (y : ∃ Tc₁) → ∃ λ (z : ∃ Tc₂) → x ≈' y × x ≈' z) ×  222
223   (∀ (y : ∃ Tc₁) (z : ∃ Tc₂) → y ≈' z → ∃ λ (x : ∃ Tc) → x ≈' y)          223
```

- Line 222 ensures that all ticks of c coincides with a tick of both c_1 and c_2.
- Line 223 ensures that whenever a tick of c_1 and c_2 coincide, c ticks.

Intersection and independance Intersection can be used as another way of expressing the independance between two clocks. Indeed, two clocks that are independant should have an empty intersection, established as follows:

```
224 passc≈c₁♯c₂ : ∀ {c c₁ c₂} → c ≈ c₁ ⋂ c₂ → c₁ ♯ c₂ → passive c           224
225 passc≈c₁♯c₂ (c→c₁c₂ , _) c₁♯c₂ i tci =                                    225
226   let (y , z , x≈y , x≈z) = c→c₁c₂ (i , tci) in                            226
227     c₁♯c₂ y z (trans≈ (sym≈ x≈y) x≈z)                                      227
```

Symmetry of intersection The intersection is symmetrical over its two last operands, which is natural considering its definition.

```
228 symInter : ∀ {c} → Symmetric (c ≈_⋂_)                                     228
229 symInter (c→c₁c₂ , c₁c₂→c) =                                              229
230   (λ x → case c→c₁c₂ x of λ {(y , z , x≈y , y≈z) → z , y , y≈z , x≈y}) ,  230
231   (λ y z x → case (c₁c₂→c z y) (sym≈ x) of                                231
232     λ {(t , t≈z) → t , trans≈ t≈z (sym≈ x)})                              232
```

Intersection and subclocking If a clock c is the intersection of two clocks, then c is a subclock of both of these clocks.

```
233 subInterₗ : ∀ {c c₁ c₂} → c ≈ c₁ ⋂ c₂ → c ⊑ c₁                           233
234 subInterₗ (c→c₁c₂ , _) x with c→c₁c₂ x                                    234
235 subInterₗ (_ , _) _ | y , _ , x≈y , _ = y , x≈y                           235
236 -                                                                          236
237 subInterᵣ : ∀ {c c₁ c₂} → c ≈ c₁ ⋂ c₂ → c ⊑ c₂                           237
238 subInterᵣ {c} {c₁} {c₂} =                                                  238
239   subInterₗ {c} {c₂} {c₁} ∘ symInter {c} {c₁} {c₂}                        239
```

Unicity of intersection If two clocks are the intersection of the same clocks, then one of them is necessarily a subclock of the other one.

```
240  ≈⋂→⊑ : ∀ {c₀ c c₁ c₂} → c₀ ≈ c₁ ⋂ c₂ → c ≈ c₁ ⋂ c₂ → c ⊑ c₀        240
241  ≈⋂→⊑ (_ , c₁c₂→c₀) (c→c₁c₂ , _) x =                                  241
242    let y , z , x≈y , x≈z = c→c₁c₂ x in                                  242
243    let t , t≈y = c₁c₂→c₀ y z (trans≈ (sym≈ x≈y) x≈z) in                 243
244    t , trans≈ x≈y (sym≈ t≈y)                                            244
```

Since the intersection is symmetrical, the other is also a subclock of the one, which means they are in fact equal. In other words, the intersection is unique towards clock equality.

```
245  unicityInter : ∀ {c₀ c c₁ c₂} → c₀ ≈ c₁ ⋂ c₂ → c ≈ c₁ ⋂ c₂ → c ∽ c₀        245
246  unicityInter {c₀} {c} {c₁} {c₂} p q =                                      246
247    ≈⋂→⊑ {c₀} {c} {c₁} {c₂} p q , ≈⋂→⊑ {c} {c₀} {c₁} {c₂} q p                247
```

Stability with subclocking As shown above, the intersection is particularly regular towards subclocking. Here are two other consequences of this regularity.

Firstly, if two clocks c and c_1 are in a relation of subclocking, then their intersection is c itself. Another way of seeing this property is that intersecting a clock with a clock for which it is a subclock leaves it unchanged.

```
248  ⊑→≈⋂ : ∀ {c₁ c₂} → c₁ ⊑ c₂ → c₁ ≈ c₁ ⋂ c₂                                        248
249  ⊑→≈⋂ c₁⊑c₂ =                                                                      249
250    (λ x → x , let (y , x≈y) = c₁⊑c₂ x in y , refl≈ , x≈y) , λ x _ _ → x , refl≈    250
```

Secondly, if a clock is a subclock of two clocks, then it is also a subclock of their intersection.

```
251  ⊑⊑→⊑⋂ : ∀ {c₀ c c₁ c₂} → c₀ ⊑ c₁ → c₀ ⊑ c₂ → c ≈ c₁ ⋂ c₂ → c₀ ⊑ c    251
252  ⊑⊑→⊑⋂ c₀⊑c₁ c₀⊑c₂ (_ , c₁c₂→c) x₀ =                                  252
253    let (x₁ , x₀≈x₁) = c₀⊑c₁ x₀ in                                       253
254    let (x₂ , x₀≈x₂) = c₀⊑c₂ x₀ in                                       254
255    let (x , p) = c₁c₂→c x₁ x₂ (trans≈ (sym≈ x₀≈x₁) x₀≈x₂) in            255
256      x , trans≈ x₀≈x₁ (sym≈ p)                                          256
```

6.2.4.c Union

Informal definition Another common and natural expression on clocks is the union. Informally, the clock c can be seen as the union of two clocks c_1 and c_2 when it ticks whenever either of these clocks tick, while each of its ticks has to

coincide with a tick of c_1 or a tick of c_2. This notion of union corresponds to the common notion of union over the subset of instants.

Example Figure 6.25 shows an example of union, where the two clocks c ticks five times, corresponding to the five separate ticks from either c_1 or c_2.

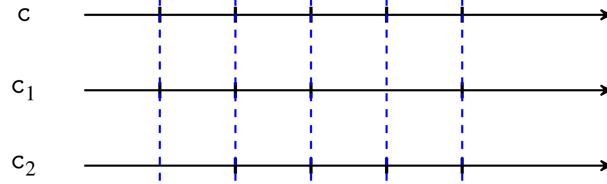


Figure 6.25: An example of union

Formal definition Union is represented by the following CCSL expression:

```
257  _≈_⋃_ : CCSLExpression                                            257
258  (Tc ⧖ _) ≈ (Tc₁ ⧖ _) ⋃ (Tc₂ ⧖ _) =                                258
259    (∀ (y : ∃ Tc) → ∃ λ (x : ∃ (Tc₁ ∪ Tc₂)) → x ≈' y) ×              259
260    (∀ (x : ∃ (Tc₁ ∪ Tc₂)) → ∃ λ (y : ∃ Tc) → x ≈' y)                260
```

- Line 261 ensures that all ticks of c coincides with at least a tick of c_1 or c_2.
- Line 262 ensures that all ticks from c_1 or c_2 have a coincident tick in c.

Symmetry of union The union is symmetrical over its two last operands, which is also natural considering the way it is defined. The proof relies on swap⊎ which swaps a sum of types (A ⊎ B → B ⊎ A).

```
261  symUnion : ∀ {c} → Symmetric (c ≈_⋃_)                              261
262  symUnion (c→c₁c₂ , c₁c₂→c) =                                       262
263    (λ x → let ((y , ty) , p) = c→c₁c₂ x                             263
264      in (y , swap⊎ ty) , p) ,                                       264
265        λ {(x , tx) → c₁c₂→c (x , swap⊎ tx)}                         265
```

Union and subclocking If a clock c is the intersection of two clocks, then both of these clocks are subclocks of c.

```
266  subUnionₗ : ∀ {c c₁ c₂} → c ≈ c₁ ⋃ c₂ → c₁ ⊑ c                     266
267  subUnionₗ (_ , c₁c₂→c) (x , Tc₁x) = c₁c₂→c (x , inj₁ Tc₁x)         267
```

```
268  subUnionᵣ : ∀ {c c₁ c₂} → c ≈ c₁ ⋃ c₂ → c₂ ⊑ c                    268
269  subUnionᵣ {c} {c₁} {c₂} =                                         269
270    subUnionₗ {c} {c₂} {c₁} ∘ symUnion {c} {c₁} {c₂}                270
```

Unicity of the union If two clocks are unions of the same clock, then one of them is necessarily a subclock of the other.

```
271  ≈⋃→⊑ : ∀ {c₀ c c₁ c₂} → c₀ ≈ c₁ ⋃ c₂ → c ≈ c₁ ⋃ c₂ → c ⊑ c₀          271
272  ≈⋃→⊑ (_ , c₁c₂→c₀) (c→c₁c₂ , _) x = let (y , x≈y) = c→c₁c₂ x in     272
273    let z , y≈z = c₁c₂→c₀ y in z , trans≈ (sym≈ x≈y) y≈z             273
```

Since the union is symmetrical, the other is also a subclock of the first one, which means they are in fact equal. In other words, the union is unique towards clock equality.

```
274  unicityUnion : ∀ {c₀ c c₁ c₂} → c₀ ≈ c₁ ⋃ c₂ → c ≈ c₁ ⋃ c₂ → c ∽ c₀    274
275  unicityUnion {c₀} {c} {c₁} {c₂} p q =                              275
276    ≈⋃→⊑ {c₀} {c} {c₁} {c₂} p q , ≈⋃→⊑ {c} {c₀} {c₁} {c₂} q p         276
```

Stability with subclocking As shown above, the union is particularly regular towards the subclocking. Here are two other consequences of this regularity.

Firstly, if two clocks c_1 and c_2 are in a relation of subclocking, then their union is c_2 itself. Another way of seeing this property is that uniting a clock with one of its subclocks leaves it unchanged.

```
277  ⊑→≈⋃ : ∀ {c₁ c₂} → c₁ ⊑ c₂ → c₂ ≈ c₁ ⋃ c₂                          277
278  ⊑→≈⋃ c₁⊑c₂ =                                                        278
279    (λ {(x , tc₁x) → (x , inj₂ tc₁x) , refl≈}) ,                       279
280    λ {(x , inj₁ tcx) → c₁⊑c₂ (x , tcx) ; (x , inj₂ tc₁x) → (x , tc₁x) , refl≈}   280
```

Secondly, if two clocks are subclocks of the same clock, then their union is a subclock of this clock.

```
281  ⊑⊑→⊑⋃ : ∀ {c₀ c c₁ c₂} → c₁ ⊑ c₀ → c₂ ⊑ c₀ → c ≈ c₁ ⋃ c₂ → c ⊑ c₀    281
282  ⊑⊑→⊑⋃ c₁⊑c₀ c₂⊑c₀ (c→c₁c₂ , _) x with c→c₁c₂ x                     282
283  ⊑⊑→⊑⋃ c₁⊑c₀ c₂⊑c₀ (c→c₁c₂ , _) x | (x₁ , inj₁ tx₁) , x₁≈x =        283
284    let (x₀ , x₁≈x₀) = c₁⊑c₀ (x₁ , tx₁) in x₀ , trans≈ (sym≈ x₁≈x) x₁≈x₀   284
285  ⊑⊑→⊑⋃ c₁⊑c₀ c₂⊑c₀ (c→c₁c₂ , _) x | (x₂ , inj₂ tx₂) , x₂≈x =        285
286    let (x₀ , x₂≈x₀) = c₂⊑c₀ (x₂ , tx₂) in x₀ , trans≈ (sym≈ x₂≈x) x₂≈x₀   286
```

Intersection and union As depicted when describing union and intersection, both are fairly similar, as one can expect. They are the two faces of the same coin, which can be formalized as a lattice. However, a mathematical lattice is composed of operators, while we deal here with a relational specification. What we can show however is that any two operators that would satisfy our specification of the union and the intersection necessarily form a lattice. Here is the specification in question:

```
287 record SpecLattice⋃⋂ (_∧_ _∨_ : Clock → Clock → Clock) : Set₁ where 287
288   field                                                                  288
289     inter∧ : ∀ {c₁ c₂} → (c₁ ∧ c₂) ≋ c₁ ⋂ c₂                             289
290     union∨ : ∀ {c₁ c₂} → (c₁ ∨ c₂) ≋ c₂ ⋃ c₁                             290
```

Finding the partial order on which these operators should be a lattice is easy: both the union and the intersection have a huge regularity regarding equality and subclocking, and we have proven that these relations form a partial order. This is the logical candidate. The proof that our specification implies that the two operators form a lattice with this partial order is given in the appendices in Section A.4.8. It consists in instantiating the lattice fields with the appropriate properties, which have already been proven in this section. The proof signature is as follows :

```
291 specToLattice⋃⋂ : ∀ {_∧_ _∨_}                                          291
292   → SpecLattice⋃⋂ _∧_ _∨_ → IsLattice _∽_ _⊑_ _∨_ _∧_                   292
```

6.2.4.d Sup and Inf

.

Informal definition The Sup c of two clocks c_1 and c_2 represents a clock that is the fastest yet slower than c_1 and c_2. This means that c is slower than both c_1 and c_2 while being faster than any clock that has this property. Similarly, the Inf c of two clocks c_1 and c_2 represents a clock that is the slowest clock yet faster than c_1 and c_2. These correspond to the common notions of greatest lower bound (meet) and least upper bound (join).

Formal definition Their formal definition is the following:

```
293 _≋_⋁_ : CCSLExpression                                  293
294 c ≋ c₁ ⋁ c₂ = c₁ ≼≼ c × c₂ ≼≼ c ×                        294
295   ∀ {c₀} → c₁ ≼≼ c₀ → c₂ ≼≼ c₀ → c ≼≼ c₀                 295
296 -                                                        296
297 _≋_⋀_ : CCSLExpression                                  297
298 c ≋ c₁ ⋀ c₂ = c ≼≼ c₁ × c ≼≼ c₂ ×                        298
299   ∀ {c₀} → c₀ ≼≼ c₁ → c₀ ≼≼ c₂ → c₀ ≼≼ c                 299
```

A lattice towards _≼≼_ and _∽_ As a direct consequence of these definitions, these two expressions form a lattice with the partial order that we established between _≼≼_ and _∽_. This lattice however is far less interesting than the previous one because the formal definitions of the sup and inf expression directly encapsulate this notion. However, it is interesting to notice that both partial orders that were established in this work have their counterpart in terms of lattices using these couples of expressions.

```
300  record SpecLattice⋁⋀ (_∧_ _∨_ : Fun₂ WFClock) : Set₁ where            300
301    field                                                                301
302      inter∧ : ∀ {c₁ c₂} → clock (c₁ ∧ c₂) ≈ clock c₁ ⋀ clock c₂         302
303      union∨ : ∀ {c₁ c₂} → clock (c₁ ∨ c₂) ≈ clock c₂ ⋁ clock c₁         303
304  -                                                                      304
305  specToLattice⋁⋀ : ∀ {_∧_ _∨_}                                          305
306    → SpecLattice⋁⋀ _∧_ _∨_ → IsLattice _∽ₒ_ _≼≼ₒ_ _∨_ _∧_              306
307  specToLattice⋁⋀ {_∧_} {_∨_}                                            307
308    record { inter∧ = inter∧ ; union∨ = union∨ } =                       308
309    record { isPartialOrder = ≼≼-ipo ;                                   309
310      supremum = λ _ _ → (proj₁ ∘ proj₂) union∨ , proj₁ union∨ ,         310
311        λ _ → flip ((proj₂ ∘ proj₂) union∨) ;                            311
312      infimum = λ _ _ → proj₁ inter∧ , (proj₁ ∘ proj₂) inter∧ ,          312
313        λ _ → (proj₂ ∘ proj₂) inter∧}                                    313
```

Assessments

This chapter presented a mechanized version of the denotational semantics of all the non-index dependent elements of CCSL. Which means it handles each construct which does not specifically require a discrete notion of time which would allow us to access events through an indexation. However, it presents a formal way of defining discrete time through an instantiation of time with natural numbers which would ultimately allow the formal definition of index dependant constructs as well. The notion of clock is presented, followed by the notion of relations and expressions. Each CCSL element that is presented has received the following treatment: it has first been adapted to our formal framework, has been reviewed for inconsistencies and has been verified through conformity properties which possibly binds it to other constructs. When the formal context was not rich enough to establish properties which should hold when using CCSL, it has been enriched accordingly, which led to the definition of enriched clocks in specific context. We advocate all clocks to respect this enriched context, in order for the precedences between clocks to be consistent. Fortunately, the required properties are reasonable and easily established when working with natural numbers. They are as follows: a clock should have a first tick and it should be possible to decide whether it ticks on a given instant.

Chapter 7

An extension of CCSL with refinement

Outline

This chapter presents the results that we obtained when combining CCSL with our notion of refinement. These results are of two sorts. The first one consists in the mechanization of the approach. The second one consists in an investigation on the preservation of CCSL constraints between the abstract model and the concrete one that refines it, either through abstraction (from the concrete model to the abstract one), or embodiment (from the abstract model to the concrete one). They are displayed and discussed using the following outline:

1. Section 7.1 presents the stakes of this approach as well as some useful information on how its development has been conducted. It explains how and why we can couple CCSL and our notion of refinement together.

2. Section 7.2 presents a first embedding of refinement in CCSL, which consists in a 1-N (one to N) refinement between clocks. We discuss the relevance of such embedding as well as the CCSL constraints it preserves. We show that this notion of refinement is very akin to preserving constraints around clock coincidence while not so much around clock precedence, which is why we propose a second notion of clock refinement.

3. Section 7.3 presents this second embedding of refinement in CCSL, which consists in a 1-1 (one to 1) refinement between clocks. This 1-1 refinement is more constrained but also preserves more constraints around clock precedence, which is discussed and established. Using this notion of refinement, we show that three of the four possible notions of clock precedence can be transferred into the other respective level of refinement.

7.1 Introduction

7.1.1 Stakes of the approach

On the combination of CCSL and refinement In Section 5.2 we introduced our approach on the modelling of refinement which proposes to handle the different layers of refinement by assigning each of them a separate partial order and ordering these orders. CCSL is itself based on partial orders which means both our notion of refinement and our mechanization of CCSL share the same formalism, which allows us to mix them together to experiment on how CCSL behaves when combined with a notion of refinement.

On preservation over instants Before starting the investigation on this conjunction, it is important to note once again that our notion of refinement is fundamentally between partial orders. In that regard, it preserves by nature any required property over these, because it has been defined in that purpose. For instance, the strict precedence between instants is preserved through embodiment using the property $\prec_{2 \to 1}$, while the coincidence between instants is preserved through abstraction as depicted by the property $\approx_{1 \to 2}$. In other words, there is no need to prove that our refinement relation preserves the right properties, because it does so by definition. Its goal is to express what properties should be preserved by refinement.

On preservation over clocks However, these preservations are natural in terms of relations between instants, yet it is not the case for relations between clocks. This means that it would be a mistake to assume that, by nature, relations between clocks should be preserved by the use of our relation of refinement. This makes the following question relevant: are there some properties between clocks which, when specified at a given level of refinement, could be transfered into another level of refinement ? This chapter aims at answering this question.

An example of what to expect Let us take an example of what to expect here: at a given level of refinement, we know that subclocking is transitive. This means that, given three clocks c_1, c_2 and c_3, if we know that $c_1 \sqsubseteq c_2$ and $c_2 \sqsubseteq c_3$ then we can deduce $c_1 \sqsubseteq c_3$. In other words, the property $c_1 \sqsubseteq c_3$ does not need to be given as an additional constraint, because it is deducible from the rest of the context. In this chapter, we try and assess such assumptions, but in various levels of refinement.

Refinement between clocks In order to establish such properties, we need to express what it means for clocks to refine one another, which is done after a short presentation of the formal context in which our work is introduced as well as some operators and naming conventions that we use throughout this chapter.

7.1.2 Formal context

In order to express refinement between clocks, we build a formal context in which two partial orders over the same set coexist and exhibit a relation of refinement. In this context, two CCSL models exist, one for each level of refinement, which we name $_1$ and $_2$ respectively for the concrete and abstract instances.

```
module CCSLRefinement (Support : Set)
  {_≈₁_ _≈₂_ _≺₁_ _≺₂_ : Rel Support _}
  (ispo₁ : IsStrictPartialOrder _≈₁_ _≺₁_)
  (ispo₂ : IsStrictPartialOrder _≈₂_ _≺₂_)
  (refines : (_≈₁_ , _≺₁_) ≺≈ (_≈₂_ , _≺₂_)) where
-
open import CCSL (record {isStrictPartialOrder = ispo₁}) as ₁
open import CCSL (record {isStrictPartialOrder = ispo₂}) as ₂
```

In the following snippets of code, the prefixes $_1$. or $_2$. will be used to represent which level of refinement is being considered. For instance, $_1$._≼_ represents the concrete non-strict precedence. Using these prefixes, we can prove for instance that the non-strict precedence is preserved through abstraction, which will be useful later on, and which is a direct consequence of the axioms of refinement:

```
≼₁→₂ : ₁._≼_ ⇒ ₂._≼_
≼₁→₂ (inj₁ i≈₁j) = inj₁ (≈₁→₂ refines i≈₁j)
≼₁→₂ (inj₂ i≺₁j) = swap⊎ (≺₁→₂ refines i≺₁j)
```

7.1.3 Useful naming convention and operators

Naming conventions In the following section, all the snippets of code will follow these conventions:

- Each index preceding an operator with a dot gives its level of abstraction, as explained in Section 7.1.2. Note that this is not really a naming convention, but can be seen as one by the reader. For instance, $_2$.`Precedence` refers to the abstract notion of parametrized precedence presented in Section 6.2.3.e.
- The first index following a clock gives its level of abstraction. For instance, c_2 refers to an abstract clock.
- The second index following clocks from the same level of refinement differentiate one from the others. For instance, c_{11} and c_{12} are two different clocks from the concrete level of abstraction.

Composition of a binary function with two unary functions In this chapter, the operator _-⟦_⟧-_ is often used. It is an operator derived from one in the standard

library which composes a function of two parameters with two functions of a single parameters, as depicted in the appendices in Section A.5.1.

A "monadic" operator of transformation Some of the results given in the following sections are built from proofs that are mostly similar in their form, which means they contain an element that can be factorized. While this factorization was originally done in the middle of the proof effort when it became apparent, we present it here in order to conveniently present the proofs using said factorization.

We define an operator called `_>[_]>_` that is similar to a monadic binding but that takes an additional parameter in the middle. This operator is convenient because its output is of the same form as its first input which means several calls can be chained, which is exactly what is done in several of the following results. Figure 7.1 presents the behaviour of this operator, while its AGDA definition is depicted in the appendices in Section A.5.1.

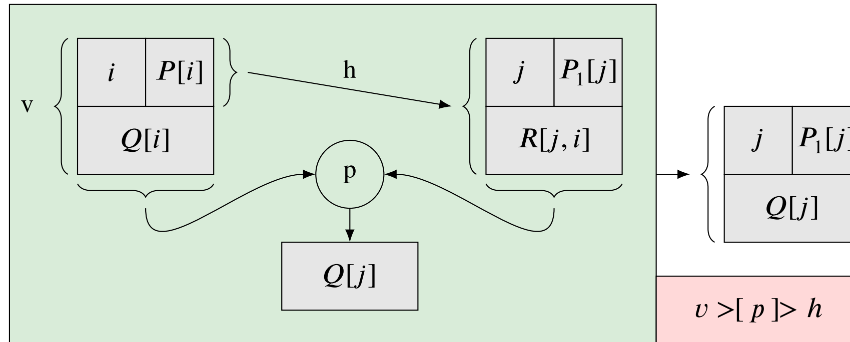


Figure 7.1: The binding operator

This operator takes a value `v` of the form `(i , P[i] , Q[i])`, a function `h` which takes an element of the form `(i , P[i])` and produces an element of the form (j , P_1[j] , R[i,j]) and a proof of preservation `p` which takes a proof `Q[i]`, a proof `R[j,i]` and produces a proof `Q[j]`. This operator uses these elements to provide a result which is of the form (j , P_1[j] , Q[j]), which is similar to the form of its input `v`.

7.2 1-N clock refinement

7.2.1 Definition of 1-N refinement

While our notion of refinement is fundamentally between partial orders, it needs a conterpart in CCSL which would allow us to express refinement between events. Since the events in CCSL are tracked by clocks that are the basic bricks on which constraints can be expressed, this counterpart has to be on clocks. A clock tracks the occurrences of an event which can possibly be refined by one or several events, which means a clock should be possibly refined by one or several clocks. Informally,

a clock refines another one when it represents a concrete event which was abstracted by the first clock. For instance, if we look back at our example in Section 5.4, the "switch on" clock is refined by several clocks, including the "compute" one.

This leads to the definition of a first notion of clock refinement: the 1-N refinement which provides no constraints on how many ticks can refine a given tick.

```
12  _≲1−n_ : REL 1.Clock 2.Clock _                                  12
13  (Ticks1 ⧖ _) ≲1−n (Ticks2 ⧖ _) =                                13
14    (∀ (y : ∃ Ticks2) → ∃ λ (x : ∃ Ticks1) → x 2.≈' y) ×          14
15    (∀ (x : ∃ Ticks1) → ∃ λ (y : ∃ Ticks2) → y 2.≈' x)            15
```

This definition is composed of two predicates as follows:

- Any tick of the abstract clock is refined by a tick of the concrete clock. These two ticks are coincident from the abstract point of view.
- Any tick of the concrete clock is a refinement of a tick of the abstract clock. These two ticks are coincident in terms of the abstract partial order.

Before starting our investigation on CCSL constructs, let us consider two clocks c_1 and c_2 such that c_1 refines c_2. Let us then take a tick `i` of c_2. We can deduce the existence of a tick `i'` of c_1 that refines `i`. Then, we can deduce the existence of a tick `i''` of c_2 which is the abstraction of `i'`. Naturally, we would expect in that scenario that `i` and `i''` are identical. In other words, this is required for our clock refinement to make sense, and can be considered as a proof obligation. While this fact does not hold definitionally, it holds propositionally using the fact that two coincident ticks of the same clock are equal:

```
16  eo∘a≡id : ∀ {c1 c2} (p : c1 ≲1−n c2) → (∀ {i : ∃ (Ticks c2)} →          16
17    i 2.≡' (proj1 ∘ proj2 p ∘ proj1 ∘ (proj1 p)) i)                         17
18  eo∘a≡id (u , _) {i} with u i                                              18
19  eo∘a≡id (_ , v) | j , j≈2i with v j                                       19
20  eo∘a≡id {c2 = c2} _ {i} | _ , j≈2i | k , k≈2j =                           20
21    2.≈→≡ {c2} {i} {k} (2.sym≈ (2.trans≈ k≈2j j≈2i))                        21
```

7.2.2 1-N refinement and coincidence

Our first area of experimentation about clock refinement is how it behaves when combined with CCSL notions related to coincidence. In other words, we investigate the preservation of CCSL notions based on coincidence when going from one level of refinement to another.

7.2.2.a Clock refinement and subclocking

We provide a proof of preservation of subclocking through abstraction. We state that subclocking is preserved when going from the lower level to the higher level

of refinement thanks to the preservation of instant coincidence through abstraction.

```
22  ⊑≲1−n : ∀ {c11 c12 c21 c22} →                                                  22
23    c11 1.⊑ c12 → c11 ≲1−n c21 → c12 ≲1−n c22 → c21 2.⊑ c22                       23
24  ⊑≲1−n o (p , _) (_ , s) i21 = (i21 , 2.refl≈)                                   24
25    >[ id -⟦ 2.trans≈ ⟧- 2.sym≈ ]> p                                              25
26    >[ id -⟦ 2.trans≈ ⟧- ≈1→2 refines ]> o                                        26
27    >[ id -⟦ 2.trans≈ ⟧- 2.sym≈ ]> s                                              27
```

7.2.2.b Clock refinement and exclusion

Another interesting property is that refining excluded clocks cannot create coincident instants, which means the refined clocks are excluded as well. This makes sense because the abstract excluded clocks have ticks that are all in different equivalence classes regarding abstract coincidence and the refined instants are still in these classes and thus cannot be coincident even from the lower point of view.

```
28  ♯≲1−n : ∀ {c11 c12 c21 c22} →                                                  28
29    c21 2.♯ c22 → c11 ≲1−n c21 → c12 ≲1−n c22 → c11 1.♯ c12                       29
30  ♯≲1−n p (q , r) (s , t) x y x≈1y with r x | t y                                 30
31  ♯≲1−n p (q , r) (s , t) x y x≈1y | rx , rx≈2x | ty , ty≈2y =                    31
32    p rx ty (2.trans≈ (2.trans≈ rx≈2x (≈1→2 refines x≈1y)) (2.sym≈ ty≈2y))        32
```

7.2.2.c Multiple clock refinement

We now investigate what happens when an abstract clock is refined by several concrete clocks, or when a concrete clock is a refinement of several abstract clocks.

Multiple concrete clocks When two clocks refine the same one, it means that these two clocks track events that are part of the events tracked by the clock being refined. Thus, it is natural to assume that the union of these clocks is still a refinement of the abstract clock.

```
33  ⋃≲1−n : ∀ {c11 c10 c12 c2} →                                                   33
34    c11 ≲1−n c2 → c12 ≲1−n c2 → c10 1.≋ c11 ⋃ c12 → c10 ≲1−n c2                   34
35  ⋃≲1−n (p , _) _ (_ , u) .proj1 x = (x , 2.refl≈)                                35
36    >[ flip 2.trans≈ ]> p                                                         36
37    >[ id -⟦ flip 2.trans≈ ⟧- (2.sym≈ ∘ ≈1→2 refines) ]> u ∘ map2 inj1            37
38  ⋃≲1−n _ _ (t , _) .proj2 x with t x                                             38
39  ⋃≲1−n (_ , q) _ _ .proj2 _ | ((i , inj1 tc11i) , i≈1x) =                        39
40    ((i , tc11i) , ≈1→2 refines i≈1x) >[ flip 2.trans≈ ]> q                       40
41  ⋃≲1−n _ (_ , s) _ .proj2 _ | ((i , inj2 tc12i) , i≈1x) =                        41
42    ((i , tc12i) , ≈1→2 refines i≈1x) >[ flip 2.trans≈ ]> s                       42
```

Multiple abstract clocks On the other hand, when a clock is a refinement of two clocks, this means that the event it tracks is deduced from two entities, leading us to assume that these entities are in fact the same. And indeed, our formalism implies such clock equality, showing that our clock refinement is not symmetrical, as expected.

```
43  ∽≲₁₋ₙ : ∀ {c₁ c₂₁ c₂₂} → c₁ ≲₁₋ₙ c₂₁ → c₁ ≲₁₋ₙ c₂₂ → c₂₁ ₂.∽ c₂₂
44  ∽≲₁₋ₙ (p , _) (_ , s) .proj₁ x = (x , ₂.refl≈)
45    >[ id -⟦ ₂.trans≈ ⟧- ₂.sym≈ ]> p
46    >[ id -⟦ ₂.trans≈ ⟧- ₂.sym≈ ]> s
47  ∽≲₁₋ₙ (_ , q) (r , _) .proj₂ x = (x , ₂.refl≈)
48    >[ id -⟦ ₂.trans≈ ⟧- ₂.sym≈ ]> r
49    >[ id -⟦ ₂.trans≈ ⟧- ₂.sym≈ ]> q
```

7.2.3 1-N refinement and precedence

While 1-N refinement preserves well CCSL notions related to coincidence, as depicted in Section 7.2.2, investigating its impact on notions based on precedence is not as fruitful in terms of preservation, the reason of which is depicted in this section.

7.2.3.a Abstraction of precedence

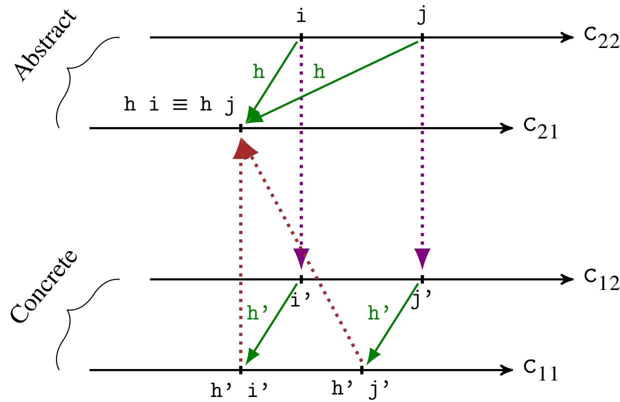


Figure 7.2: Clock precedence and abstraction

A reasonable idea would be that if two clocks precede each other in the concrete world, their respective abstracted version would also precede each other in the abstract world. Given four clocks $\mathtt{c}_{11}$ $\mathtt{c}_{12}$ $\mathtt{c}_{21}$ $\mathtt{c}_{22}$, this would be expressed as the following predicate: $\mathtt{c}_{11}\ {}_1.\preccurlyeq\preccurlyeq\ \mathtt{c}_{12} \rightarrow \mathtt{c}_{11} \lesssim_{1-n} \mathtt{c}_{21} \rightarrow \mathtt{c}_{12} \lesssim_{1-n} \mathtt{c}_{22} \rightarrow \mathtt{c}_{21}\ {}_2.\preccurlyeq\preccurlyeq\ \mathtt{c}_{22}$. However, this does not hold because the `preserves` field of the precedence is fundamentally not preserved. Note that since `preserves` does not

depend on the parameter of the `Precedence` record, this proof holds for both precedences, strict or not. Indeed, let us take two instants `i` and `j` on which c_{22} ticks such that `i` $\prec_2$ `j`. Since c_{12} refines c_{22} we can deduce the existence of two instants `i'` and `j'` on which c_{12} ticks and which are respectively coincident with `i` and `j` in the abstract level of refinement: `i` $\approx_2$ `i'` and `j` $\approx_2$ `j'`. Combining these elements, it it possible to deduce that `i'` $\prec_2$ `j'`. Applying $\prec_2 \to_1$ to this term allows us to deduce that these instants are also in a relation of strict precedence in the concrete level: `i'` $\prec_1$ `j'`. Using the fact that c_{11} precedes c_{12} we get that `h i'` $\prec_1$ `h j'`. However, what we need to achieve is `h i'` $\prec_2$ `h j'` which is not deductible from the hypotheses since the precedence abstraction only tells us that `h i'` $\preccurlyeq$ `h j'`. Indeed, it is possible that this abstraction hides the precedence of `h i'` towards `h j'` in which case the property does not hold. This is why the non-strict precedence is not preserved through abstraction. This possible case is depicted in Figure 7.2, where the dotted arrows represent the embodiment in purple and the abstraction in brown while the precedence functions are represented with green arrows.

7.2.3.b Embodiment of precedence

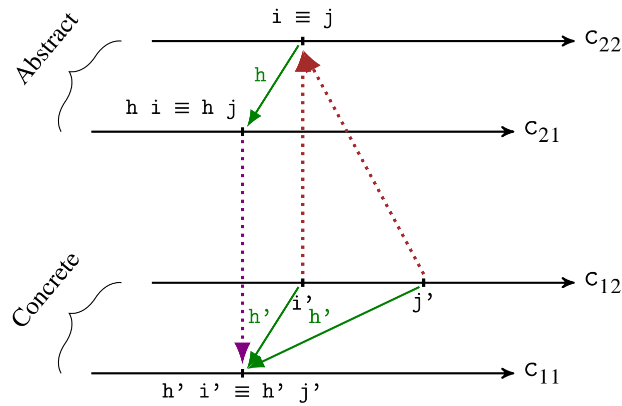


Figure 7.3: Clock precedence and embodiment

Clock precedence is fundamentally neither preserved through embodiment. This is easier to see because precedence between instants can directly be hidden through abstraction which means that if we take two instants `i'` and `j'` such that `i'` $\prec_1$ `j'` then this precedence might disappear through abstraction. Such a scenario is shown in Figure 7.3, using the same colors and symbols as in Figure 7.2.

7.2.3.c 1-N refinement and alternation

Alternation is a specific form of precedence, with an additional constraint of alternation. Since clock precedence is not preserved through 1-N refinement, alternation cannot be preserved either.

7.3 1-1 clock refinement

7.3.1 Motivation

Questioning the non-preservation of clock precedence The fact that our notion of refinement does not directly preserve precedence between clocks is to be questioned: does it fail to provide properties that should hold ? Or does it reflect the fact that precedence between clocks indeed should not be preserved by refinement ? To answer that question, we need to understand deeply what are the cases in which this preservation is not satisfied, by combining the elements depicted in Figures 7.3 and 7.2. These figures tell us that the precedence is not preserved in cases where more than one tick of the concrete clock refines the same tick of the abstract clock.

An example of non-preservation Acknowledging this fact, we provide a quick example which emphasizes them more concretely. We consider the situation where a worker has to drive nails in a plank of wood using a hammer. In the abstract level, we consider the following clocks: the worker places a nail at a given spot in the plank ($\texttt{placing}_1$) and the worker drives the nail in the plank with the hammer ($\texttt{driving}_1$).

In the concrete level, we can refine the driving of a nail as a succession of hammering steps. For instance, we can consider that, since the worker is quite strong, he needs three hammer hits to fully drive the nail inside the plank. The placing step, however, is considered unary, hence is not further refined. This leads to the following clocks at the concrete level: the worker places the nail ($\texttt{placing}_2$) and the worker makes a hammer hit on the nail ($\texttt{hitting}_2$).

In the abstract level, we can specify that $\texttt{placing}_1$ precedes $\texttt{driving}_1$ because a nail has to be placed before it can be driven. Knowing that fact, we can picture how this precedence translates in the concrete level, and notice that $\texttt{placing}_2$ does not precede $\texttt{hitting}_2$. This picturing is done in Figure 7.4.

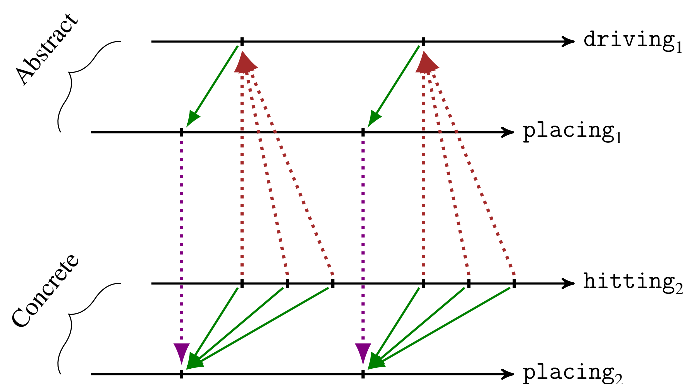


Figure 7.4: Clock precedence is not always preserved

Lessons learnt from this example As we can see in this example, the precedence does not hold because an instant of the slower clock can be bound to several instants of the faster clock in the concrete level, which invalidates the `precedes` field of the precedence which ensures the binding function to be bijective over its range. This happens when the ticks of the slower clock at the abstract level are refined by several ticks of the same concrete clock.

The question that arises is the following: if we prevent such cases for the slower abstract clock, would the precedence be preserved ? The answer is no, because if we allow the faster clock to have the same behaviour, that is to have ticks that are refined by several ticks of the concrete faster clock, the property dense would be invalidated. Indeed, each tick of the concrete slower clock would be bound to one of the corresponding ticks of the faster clock, thus leaving the others unbound.

All in all, this means that if we prevent that behaviour by not allowing any tick to be refined by several ticks of the same clock for both the slower and the faster clock, the precedence should be preserved. In other words, we define another notion of clock refinement which is a specific case of refinement which does not allow multiple refined instants of the same clock. This is the 1-1 refinement.

7.3.2 Definition of 1-1 refinement

We give another notion of clock refinement which does not allow ticks to be multiply refined. While this does not mean that a tick at the abstract level must correspond to a single tick at the concrete level, it means that these refine ticks must come from different clocks. This notion of refinement is very similar to the one presented in Section 7.2.1 except one of the predicates it contains is defined using ∃! instead of ∃ which corresponds to unique existential quantification, and which is parametrized by an underlying equality, here instantiated to the concrete coincidence. We insist on the fact that this new refinement is not meant to replace the previous one, rather a new way of expressing refinement for the users of CCSL, which is more constrained but has the upside to preserve clock precedence naturally:

```
50  _≲1-1_ : REL 1.Clock 2.Clock _                                         50
51  (Ticks1 ⧖ _) ≲1-1 (Ticks2 ⧖ _) =                                        51
52    (∀ (y : ∃ Ticks2) → ∃! (1._≈'_) λ (x : ∃ Ticks1) → x 2.≈' y) ×         52
53    (∀ (x : ∃ Ticks1) → ∃ λ (y : ∃ Ticks2) → y 2.≈' x)                    53
```

We show that a 1-1 refinement is also a 1-N refinement, which is trivial but has to be done in order to get all the preservation results that were given in the previous sections. It uses the fact that unique existence implies existence, depicted in the appendices in Section A.5.2.

```
54  ≲→≲1-1 : ∀ {c1 c2} → c1 ≲1-1 c2 → c1 ≲1-n c2                          54
55  ≲→≲1-1 (p , q) = ∃!→∃ ∘ p , q                                          55
```

7.3.3 Consequences of the definition

There are several direct consequences from this definition, which will be of great use to prove preservation properties around 1-1 refinement and CCSL relations based on precedence. 1-1 refinement forces the concrete clock to have a single tick for each tick of the abstract clocks, which means that the concrete coincidence and the abstract coincidence should be equivalent over ticks of a concrete clock which refines an abstract clock. Since by definition, we have the preservation of the coincidence through abstraction, it remains to be proved through embodiment in this specific case, which is using horizontal dots which are accepted by AGDA as a way to avoid rewriting the whole previous line:

```
≾₁₋₁→≈₂→₁ : ∀ {c₁₁ c₂₁} → c₁₁ ≾₁₋₁ c₂₁ → {k l : ∃ (Ticks c₁₁)}
  → k ₂.≈' l → k ₁.≈' l
≾₁₋₁→≈₂→₁ {c₁₁} {c₂₁} (p , q) {k} {l} k≈₂l with q k | q l
... | u , u≈₂k | v , v≈₂l with ₂.≈→≡ {c₂₁} {u} {v}
  (₂.trans≈ (₂.trans≈ u≈₂k k≈₂l) (₂.sym≈ v≈₂l))
... | refl with p u
... | j , j≈₂u , ∃! =
  ₁.trans≈ (₁.sym≈ (∃! {k} (₂.sym≈ u≈₂k))) (∃! {l} (₂.sym≈ v≈₂l))
```

As a consequence of the preservation of the coincidence, the strict precedence is also preserved through abstraction on the ticks of the concrete clock. In other words, the concrete and abstract partial orders are the same for the concrete clock:

```
≾₁₋₁→≺₁→₂ : ∀ {c₁₁ c₂₁} → c₁₁ ≾₁₋₁ c₂₁ → {k l : ∃ (Ticks c₁₁)}
  → k ₁.≺' l → k ₂.≺' l
≾₁₋₁→≺₁→₂ {c₁₁} {c₂₁} p {k} {l} k≺₁l with ≺₁→₂ refines k≺₁l
... | inj₁ k≺₂l = k≺₂l
... | inj₂ k≈₂l =
  ⊥-elim (₁.irrefl≈≺ (≾₁₋₁→≈₂→₁ {c₁₁} {c₂₁} p {k} {l} k≈₂l) k≺₁l)
```

As a last consequence of 1-1 refinement definition, we provide a proof that abstraction composed with embodiment results in the identity function. The other way around was already true for the 1-N refinement as shown in Section 7.2.1.

```
a∘e≡id : ∀ {c₁ c₂} (p : c₁ ≾₁₋₁ c₂) → (∀ {i : ∃ (Ticks c₁)} →
  i ₂.≡' (proj₁ ∘ proj₁ p ∘ proj₁ ∘ (proj₂ p)) i)
a∘e≡id (_ , v) {i} with v i
a∘e≡id (u , _) | j , j≈₂i with u j
a∘e≡id {c₁} {c₂} p {i} | j , j≈₂i | k , k≈₂j , _ =
  ₁.≈→≡ {c₁} {i} {k} (≾₁₋₁→≈₂→₁ {c₁} {c₂} p {i} {k}
    (₂.sym≈ (₂.trans≈ k≈₂j j≈₂i)))
```

7.3.4 1-1 refinement and coincidence

Since the 1-1 refinement is a special case of 1-N refinement, it inherits all the coincidence properties that were established in Section 7.2.2.

7.3.5 1-1 refinement and precedence

The main goal of the 1-1 refinement is to provide CCSL with a notion of clock refinement which naturally preserves clock precedence. In this section we investigate to what extent this preservation is guaranteed.

7.3.5.a Embodiment of precedence

We take a look at how the two precedences behave through embodiment.

Embodiment of non-strict precedence Non-strict precedence is fundamentally not preserved through embodiment, even with 1-1 refinement, and for a very concrete reason, which is that abstract coincidence can basically be transformed into any relation in the concrete level. Let us take four clocks c_{11}, c_{12}, c_{21} and c_{22} such that $c_{11} \lesssim_{1-1} c_{21}$ and $c_{12} \lesssim_{1-1} c_{22}$. If we have $c_{21} \ {}_2.\preccurlyeq\preccurlyeq\ c_{22}$ this means that for a tick i of c_{22} we have a tick j of c_{21} such that $h\ i \equiv j$ and possibly $i \approx_2 j$ which, in the concrete level, can be transformed into $j \prec_1 i$ which invalidates any attempt at preserving embodiment of non-strict precedence. Note that this is not a limitation in our ability to prove such property, but rather it simply does not hold.

Embodiment of strict precedence Strict-precedence, however, is preserved directly through embodiment. The proof is complex and is only provided in the appendices in Section A.5.3 while its signature is given here:

```
77  <<₂→<<₁ : ∀ {c₁₁ c₁₂ c₂₁ c₂₂} → c₁₁ ≲₁₋₁ c₂₁ → c₁₂ ≲₁₋₁ c₂₂        77
78      → c₂₁ ₂.<< c₂₂ → c₁₁ ₁.<< c₁₂                                    78
```

7.3.5.b Abstraction of precedence

We give an interesting result on the preservation of precedence through abstraction. Both precedences are transformed into non-strict predecence in that process. More generally, we give a proof that, given a relation `_R_` on instants such that `_R_` $\Rightarrow$ ${}_1$`._≼_` then a proof of ${}_1$`.Precedence _R_` is transformed into a proof of `_`${}_2$`.≼≼_` through abstraction. This proof is also complex and is provided in the appendices in Section A.5.4 while its signature is as follows:

```
79  prec₁→≼≼₂ : ∀ {c₁₁ c₁₂ c₂₁ c₂₂} {_R_ : Rel ₁.Support _}             79
80      → _R_ ⇒ ₁._≼_ → c₁₁ ≲₁₋₁ c₂₁ → c₁₂ ≲₁₋₁ c₂₂                       80
81      → ₁.Precedence _R_ c₁₁ c₁₂ → c₂₁ ₂.≼≼ c₂₂                          81
```

Instantiating `_R_` with `_≺₁_` we get that strict precedence becomes non-strict precedence through abstraction:

```
82  ≺≺₁→≼≼₂ : ∀ {c₁₁ c₁₂ c₂₁ c₂₂} → c₁₁ ≲₁₋₁ c₂₁ → c₁₂ ≲₁₋₁ c₂₂   82
83     → c₁₁ ₁·≺≺ c₁₂ → c₂₁ ₂·≼≼ c₂₂                              83
84  ≺≺₁→≼≼₂ = prec₁→≼≼₂ inj₂                                      84
```

Instantiating `_R_` with `_≼₁_` we get that non-strict precedence is preserved through abstraction:

```
85  ≼≼₁→≼≼₂ : ∀ {c₁₁ c₁₂ c₂₁ c₂₂} → c₁₁ ≲₁₋₁ c₂₁ → c₁₂ ≲₁₋₁ c₂₂   85
86     → c₁₁ ₁·≼≼ c₁₂ → c₂₁ ₂·≼≼ c₂₂                              86
87  ≼≼₁→≼≼₂ = prec₁→≼≼₂ id                                        87
```

7.3.5.c 1-1 refinement and alternation

Since strict precedence is transformed into non-strict precedence through abstraction, we cannot expect alternation to be preserved in the process, because it is based on strict precedence. However, we need to investigate if the additional predicate of alternation is preserved through embodiment, in which case alternation itself is preserved. Unfortunately, this is not the case, for a very specific reason. Alternation embeds in its definition the quantity `i ≺ h j`, which needs to be preserved for the alternation to be preserved. However, there is no guarantee that this will indeed be preserved because `i` and `h j` are ticks of different clocks, and 1-1 refinement further constrains relation for ticks of the same clock, as seen in Section 7.3.3, but not for ticks of different clocks. Which means that refinement alone cannot ensure that alternation is preserved from one level to the other, as shown on Figure 7.5 where we can see that `h' j'` precedes `i'`.

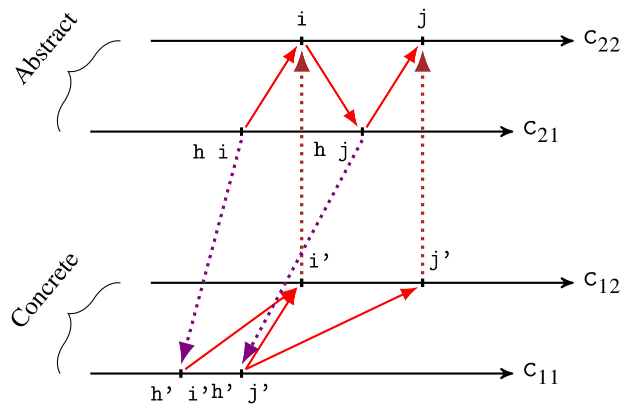


Figure 7.5: An example of non-preservation of alternation

Assessments

The semantics of CCSL has been enriched with the notion of refinement which was described in Chapter 5. Concretely, this means that we provide two new CCSL relations of refinement between two clocks: 1-1 refinement and 1-N refinement. These relations are fundamentally different from the others because they build a bridge between clocks coming from two different CCSL specifications rather than a bridge between two clocks coming from the same specification. Each of these different specifications must correspond to a given level of refinement in the sense that their partial order should comply with the relation of refinement which was defined in Chapter 5. This gives CCSL additional expressiveness: to make a CCSL specification which handles refinement, we advocate one specification be built per desired level of refinement, and subsequently binding them through clock refinement. This way, the global behaviour of the system could be verified using our notion of refinement.

In the process of adding refinement to CCSL, we investigated on how CCSL operators behave when coupled with the new relations of clock refinement. In other words, we tried to assess which properties were preserved from one level of refinement to the others. This investigation was fruitful when dealing with CCSL notions which are bound to coincidences using 1-N refinement, in the sense that refinement is fairly regular towards coincidence. It was even more fruitful when considering 1-1 refinement, which was designed to preserve properties of precedence, and which succeeded at doing so. Overall, our relation of refinement between partial orders as well as the two relations of refinement between clocks provide a formal context in which models or languages could be compared, and the relations between them assessed.

Chapter 8

Conclusion

8.1 Assessments

This work on heterogeneous systems was grounded in the GEMOC context [46], where languages can be defined using three axes: their structural aspects as an ECORE/ OCL meta-model, their atomic actions as methods associated to meta classes, and their global behaviour using CCSL to express constraints between occurrences of events of different nature. GEMOC is designed to handle heterogeneity and thus the different separations of concern it implies. In this context, we proposed several contributions which are summarized in this section. All these contributions were formally modelled and verified using AGDA.

On formal modelling of transition systems The first contribution was directed toward the depiction of systems which exhibit both structural aspects and atomic actions. In other words, this contribution was a study on the formal modelling of event-based systems: systems that are described by a state which can evolve through guarded actions. This contribution handles both these aspects through two target languages: PETRI NET and SIMPLEPDL. These languages have originally been chosen because they share a close bond: each model of process (each instance of the SIMPLEPDL language) can be transformed into a model of PETRI NET, in which case these two models are in a relation of weak bisimulation. While we did not formally prove the correctness of such transformation and relation – more on this fact is explained in Section 8.2, we provide a methodology to model such systems. This methodology involves a depiction of the states containing both the actual data and structural conformity properties about them. It also involves the depiction of the possible events through their guards, and a step-by-step evolution of the system when said guards are decided (in the sense of decidability) correct. In the process of depicting the internal state of such systems, we developed a library on notions called globally unique lists. These special lists can be instantiated into different notions, among which maps and sets (named bags in this document, because set is an AGDA keyword) are found. The depiction of this library allowed us to formulate

and illustrate a methodology on building libraries in proof assistants, which consists in coupling each definition with properties, the goal of which is to enforce a strong confidence towards the defined notions. This methodology has been used in all of our development, and will be further discussed when summarizing our work on CCSL. Mentioning CCSL leads to our next contribution. While this contribution provides a convenient methodology for the modelling of both the structural aspects of transition systems as well as their atomic actions, it does not take into account additional constraints over their traces of execution to coordinate their execution. More precisely, these constraints could be expressed relying on additional components in the state and additional constraints on the guards that enable transitions ; but this adds complexity to the atomic events by merging two concerns, local state evolution during an event and global state management during a complete execution trace. Thus, GEMOC advocates the separation of these two kinds of concerns. This choice leads to our following contribution: the modelling of logical time with the addition of a notion of trace refinement, usually referred to as instant refinement.

On formal depiction of trace refinement The second contribution of this thesis is two-fold: we provided a formal modelling of notions around time, with the addition of a relational depiction of trace refinement. We started by depicting notions which are usually used when modelling time in asynchronous executions. First, the notion of time itself, then the notion of execution traces, events and most importantly, partial orders. In asynchronous system executions, the events are usually bound by a partial order, which indicates the coincidence and precedence bindings which can be observed or assessed. In these cases, the usual context is a set of instants coupled with a partial order between these instants. While depicting this context as well as providing its formal modelling, we also took a step back on this context in order to define a relation of refinement between traces. This notion of refinement is not conceptually new, in the sense that it aims at depicting the commonly accepted notion of system refinement which corresponds to the vertical separation in a system's development. However, the depiction we made of refinement is, to our knowledge, new: instead of integrating the notion of refinement inside the usual context (set of instants and a partial order), we propose to extend this context and associate each level of refinement with a specific partial order, each of them over the same set of instants. Thus, the new context we propose is composed of a set of instants and as many partial orders as levels of refinement. Since this new context is now composed of several partial orders, we propose a relation to bind these orders, the purpose of which is to embed the usual semantics of refinement. Indeed, we cannot allow our structure to be composed of any partial orders. These partial orders must obey certain rules such that each of these is thinner than the one that follows it in the hierarchy, and wider than the one that precedes it. In that regard, this only makes sense for our relation of refinement to be expressed between partial orders. This relation states what it means for a partial order to be thinner than another. This refinement relation encompasses the usual preservation properties

about refinement: precedence is preserved through embodiment while coincidence is preserved through abstraction. This relation of refinement is a mathematical relation, which means properties can be established about it. We established the fact that this relation of refinement is a partial order between partial order, by providing an adequate notion of equivalence between partial orders. This partial ordering between levels of refinement is a requirement for refinement and it was mandatory for our relation to exhibit such a property. Overall, this contribution sets a context in which time constraints can be expressed, including constraints expressed in different levels of refinement. However, to express such constraints, a language is needed. In GEMOC, this language is CCSL which is why our next contribution is a mechanization of the denotational semantics of CCSL.

On a mechanized denotational semantics for CCSL The third contribution is the mechanization of the denotational semantics of CCSL. CCSL is the language used in GEMOC to model and express constraints between the occurrences of events coming from an heterogeneous context. Now that a formal context was provided which allows the expression of such constraints, we needed a formal mechanization of the denotational semantics of this language. Such a semantics already existed on paper, and we provided in this contribution its mechanized adaptation. This mechanization provides a formal definition of all the core elements of CCSL, more precisely, all the elements of CCSL that are not index dependent. Index dependent constructs are specific to a set of instants that is isomorphic to the natural numbers, and our approach is more generic, which is why these elements are not originally embedded in our work. However, this will be the object of a future work as depicted in Section 8.2. While embedding CCSL notions in our formal AGDA framework, we used the same methodology as advocated in Chapter 3, which means we coupled our definitions with properties of conformity. These properties of conformity have a double objective: first, they allow us to build a strong confidence in the definitions by providing proofs of their informal requirements. Second, they allow us to improve the definitions when such proofs cannot be directly built, thus improving the overall semantics of the language. In our formal development, we encountered such a case twice: first, when defining the notion of precedence, we found that an axiom had been assumed and forgotten from the paper version of the semantics: the axiom of density which forces the binding function to be bijective over its range. We added this axiom in our definition of the precedence accordingly. Second, when establishing algebraic properties about our notions of precedence, we found that some requirements were not met. This led to an investigation on which constraints should hold for such requirements to be met. Ultimately, it led to improved versions of clocks on which the precedences act according to the requirements. These semantics also allowed us to define the notion of clocks more precisely and formally. If we consider the power-set of instants, then the set of all possible clocks is the subset of this power-set such that the instants it contains are totally ordered. This gives us a formal modelling of CCSL in a context where refinement can be expressed,

which leads to our last contribution: the coupling of CCSL and refinement. In other words, the addition of a notion of refinement to CCSL.

On the addition of refinement to CCSL Our last contribution consists in the addition of refinement to CCSL. Our second contribution provides a formal context on which refinement can be described, while our third contribution provides a modelling of CCSL using the same context. However, CCSL does not currently have relations over clocks from different levels of refinement, which is the core of this contribution. We extend CCSL with two notions of refinement, which is made possible by combining the two previous contributions. The first notion of refinement is the 1-N refinement which binds clocks together, while allowing the refined clock to have multiple ticks that correspond to the same tick of the abstract clock. This refinement is the more natural conceptually. Having added a new relation to CCSL, it is natural to combine it with already existing relations through preservation properties. In other words, we made an investigation on the preservation of CCSL relations with this first notion of clock refinement. This answers the following question: in a context where several layers of refinement exist, and where several clocks exist in each layer, is it possible to translate relations expressed at a level of refinement to relations expressed at another level ? Using 1-N refinement, we were able to prove that CCSL notions which are based on coincidence between instants yield several such properties, while notions based on precedence did not. While investigating the reasons behind it, we realized that preventing ticks to be multiply refined by ticks of the same clock, we can have additional properties, which led to the definition of a second relation of refinement between clocks: 1-1 refinement. Using this second notion of refinement, we were able to establish preservation properties about CCSL notions based on precedence. All in all, we added two notions of refinement to CCSL: 1-N refinement is very permissive and preserves only a few CCSL constructs, while 1-1 refinement is more constrained and thus preserves more CCSL constructs. This contribution concludes our work on trace semantics by successfully combining CCSL and refinement in a formal context, and thus providing a formal extension of this language. As a consequence, the expressiveness of CCSL has been substantially increased: it is now possible to treat events and constraints from different levels of refinement in the same formal context, thus combining planar and vertical separations of concerns.

8.2 Limitations and Perspectives

This work provides several perspectives, which are described using a time-related categorization as follows:

Short term There are some technical aspects in our work that could be quite easily improved but could not be conducted in this PhD duration:

- AGDA's standard library has recently been enriched with a notion of maps encoded with AVLs tree. This library should be studied in order to assess if our work on event-based systems should keep relying on our maps or on these maps. There also exists some work [114] on describing graphs in HASKELL and in AGDA which could possibly be used in order to improve our representation of the states of such systems.

- Our work on event-based systems rely on the `Maybe` monad to handle error cases. Indeed, in an effort to hide the usage of proofs from the user, we use decidability to build these proofs. When the requirements do not hold, we return an error. We could improve this mechanism to use a monad which could embed some information as to what went wrong. For instance, should the user misspell a name in the building of a model, he should be informed that this misspelling is the origin of the issue, which is not currently the case. Handling these error cases was not a priority in our work, but this would definitely improve the usability of our approach.

- When combining CCSL and refinement, we introduced an operator of transformation in Section 7.1.3 to factorize some of the proofs as well as making them somewhat more readable. There are several other proofs that could be factorized in a similar manner. This would require to take a step back from the proof effort depicted in this document and provide operators which allow us to articulate proof terms in a more convenient manner. Two examples come to mind: the proofs that deal with terms of instant precedence or coincidence, and functions that deal with decidability to produce an element of type `Maybe`. These changes could improve the overall quality of our work.

Medium term There are some improvements / extensions of our current work that can be considered:

- We considered providing decidability proofs for predicates of conformity in the structural depiction of the event-based systems. This would allow us to create a model without bothering about its continuous correctness and assessing its correctness afterwards. Although we believe that a continuously correct way of building models is to be advocated, such an option could also be useful. These two alternatives are common in the dependent type community and no definite answer has been provided regarding this issue.

- Read arcs of PETRI NET are currently handled by arcs that consume and produce the same amount of token in a given place from a given transition, in which case this emulates a read arc with a weight of this number of tokens. However, should we refine the execution of our PETRI NET models such that the production and the consumption of tokens would not occur simultaneously, this could lead to inconsistent behaviours which should be handled if such a refinement arises. More precisely, this change would be mandatory

if we target time PETRI NET, where read arcs and normal arcs that give and take the same amount of tokens do not have the same semantics.

- While we did not explicitly model index dependent CCSL constructs, we gave examples of clocks with an underlying set of instants instantiated to natural numbers. In this context, we believe that the original version of the precedence (without the `dense` property) is correct because we could provide another binding function which respects this property by "moving" the bindings to the left as mentioned in Section 6.2.3.e. This is provable in our AGDA framework but it seems to be quite time consuming, as some quick attempts have shown, and we allocated such time to other aspects of this work, for instance on the improvement of our notions of clock refinement.

- An interesting improvement of our work would be to provide decidability properties over CCSL relations depending on the underlying set of instants, and on the decidability of the predicate of ticks of the clocks. Should we provide such decidability proofs, this would considerably ease the verification of traces such as the one that was done in Section 5.4 because we would not have to provide proofs for relations by hand.

- Finally, the index dependent constructs of CCSL should be modelled explicitly. This requires us to define a context on which the concept of index has a meaning. Should this context necessarily be naturals numbers ? What properties should the underlying set of instant have to allow this concept to make sense ? This remains to be investigated.

Long term There are some more open and general issues which could be tackled from our work:

- Both PETRI NET and SIMPLEPDL can be extended with time constraints on their respective events. It could be interesting to model this aspect in our framework because time constraints are common in event-based systems. This would require us to assess where these constraints should be expressed, and, most importantly, it requires the existence of a global clock which needs to be handled as well.

- As mentioned several times in this document, our two target languages of Chapter 4, PETRI NET and SIMPLEPDL are in a relation of weak bisimulation [41], or refinement, depending on the point of view. It would be interesting to ultimately be able to prove such a property in our framework. This would require us to build additional relationships between the different aspects of our work. This would also possibly require the formal definition of a transformation between a model of process and a PETRI NET. Or, instead of this transformation, this could require the definition of a relation stating if a given PETRI NET is the embodiment of a given model of process.

- A very important extension to our work on event-based systems would be to define a systematic transformation between a meta-model expressed in MOF (or an implementation of MOF like ECORE) coupled with OCL constraints and an AGDA definition of the language represented by this meta-model. We could even imagine having an automated process to do the transformation. This automation could be done in a modelling tool such as ECLIPSE and generate AGDA code which would then be given to the type checker for verification. This would allow the system engineer to build their models using DSMLs and rely on AGDA to verify them, similarly to existing approaches using COQ in [86] or WHY3 in [60].

- Concerning our work on CCSL, we have a very interesting perspective. Throughout our work, we have proven many conformity properties. As mentioned, these properties are mandatory to assess the correctness of our modelling, but they could also be exploited as means of reducing a set of CCSL constraints or proving the inconsistency of such set. For instance, if we have proven in our framework, given two constraints `A` and `B` that `A` implies `B` and if both these constraints are present in a CCSL specification, then we could safely remove `B` from said specification. An idea to achieve this goal would be to create a logic which manipulates CCSL constraints. The inference rules of this logics would be the properties that have been proven in the framework, including the properties about refinement. We would embed this logic inside AGDA using some embedding techniques (either shallow or deep) and would be able to express equivalence between CCSL specification and possibly the smallest member of each of these equivalence classes.

- We would be very much interesting in adding our two CCSL relations in the official version of CCSL, as well as in TIMESQUARE. This would require us to embed our new temporal context with several partial orders, as well as the two notions of refinement of CCSL in that context. Since instant refinement was considered by CCSL experts as a wished facility, this perspective could definitely be implemented.

- As a final perspective, we could rely on our framework to verify part of the TIMESQUARE tool-set. TIMESQUARE generates a possible finite sub-trace of the CCSL specification it handles, and our framework allows us to verify such traces regarding a specification. This verification would require us to select the best strategy to verify such a complex tool: should we encode TIMESQUARE in AGDA and prove that it only builds correct traces ? Should we, on the other hand, only make a systematic verification of the traces TIMESQUARE builds ? Such discussion is very interesting and could lead to an actual verification of TIMESQUARE, with the eventual addition of refinement.

8.3 Metrics

Line count metric

This work's line count, without comments and blank lines is as follows:

Didactic code This work contains some didactic pieces of code, in the form of tutorials and examples. These elements have the following line count:

Unicode.agda	9
Mixfix.agda	9
Addition.lagda	57
ListSize.agda	16
IfThenElse.agda	6
ListSugar.agda	17
Unif.agda	8
Associativity.agda	14
Irrelevance.agda	26
Commut.agda	12
Currying.agda	14
Tutorial.lagda	55
Total didactic	243

Helpers This work relies on a file containing helper elements, which are used in several different part of this work, and regrouped in a single file, the length of which is:

Helper.lagda	85
Total helpers	85

Globally unique lists Our library on globally unique lists metric, with its two main instantiations, has the following number of lines:

ListConform.lagda	554
ListAssoc.lagda	11
ListUnique.lagda	10
Total GULists	575

Event-based systems Our two case studies on event-based systems, PETRI NET and SIMPLEPDL have the following line count:

SimplePDL.lagda	238
Petrinet.lagda	234
Total Event-based	472

Time Our files depicting notions and proofs about time and intervals have the following length:

Instant.lagda	127
Interval.lagda	30
Unary.lagda	22
Total time	179

CCSL Our work around CCSL, including the two instantiations over natural numbers and integers, has the following size:

CCSL.lagda	439
CCSLIntegers.lagda	117
CCSLNaturals.lagda	79
Total CCSL	635

Refinement Our work on refinement, including its application to CCSL, has the following line count:

Refinement.lagda	72
RefinementExample.lagda	125
CCSLRefinement.lagda	175
Total refinement	372

Total In total, the mechanization part of this thesis contains 2571 non-empty non-comments lines of AGDA.

A step back from the line count metric

Although line count metric is provided and usually required in such a document, it is, in my opinion, quite irrelevant in assessing the complexity and the completeness of this work, especially because it was written using AGDA. AGDA is a very compact language, and most of these 2571 lines are either signatures or proof terms. In addition, the line count of a specific notion or definition is hardly equivalent to its complexity. A first example is our definition of refinement depicted in Section 5.3.1:

```
_<≈_ : ∀ {ℓ} → Rel (Rel I ℓ × Rel I ℓ) _
(≈₁ , <₁) <≈ (≈₂ , <₂) = Transform ≈₁ ≈₂ <₁ <₂
    (λ ≈₂ _ → ≈₂)
    (λ ≈₁ <₁ → ≈₁ ∪ (<₁ ∪ flip <₁))
    (flip _∪_)
    (λ _ <₁ → <₁)
```

This definition only takes six lines of code but is both the result of a long thought process which gave me the idea of relating partial orders, and the origin of another long thought process to assess its exact implications. In other words, these six lines are more impactful than significant larger pieces of code elsewhere in our work. Another example is our definition of clocks depicted in Section 6.2.2.b:

```
Clock₀ : Set₁
Clock₀ = ∃ \Ticks → Trichotomous {A = ∃ Ticks} _≡'_ _<'_
```

This definition is the culmination of a long and deep thought process and did not take that form until the redaction of this document. It results from a slow and lengthy process of gaining knowledge and understanding in AGDA which I have learned as an autodidact. In other words, although this definition is only 2 lines, it is the result of several years of learning and questioning (both about the nature of the objects defined but also on the environment in which they are defined) and can hardly be summarized as a one digit number.

As an addition, the code depicted in this document is the result of a long process of synthesis, learning and rewriting. As an example, here is a piece of code as it was originally written, on the left (without colouration, because it no longer type-checks), and as it was presented in this document on the right. As one can see, the original code is both far less elegant and double the size of its current version.

```
lemma₀ : ∀ {b l l'} {conf : Conform _} {conf' : Conform _} →
  b ∈ᵢ l → get f b from (listConf l conf)
    ≡ get f b from (listConf l' conf') → b ∈ᵢ l'
lemma₀ {b} {l} {l'} b∈ᵢl ga≡ga with dec∈ (f b) l
lemma₀ {b} {l} {l'} b∈ᵢl ga≡ga | yes p with dec∈ (f b) l'
lemma₀ {b} {l} {l'} {conf} {conf'} b∈ᵢl ga≡ga | yes p | yes q =
  case subst (λ x₁ → just x₁ ≡ just (get f b from l' when q))
    (lemma {conf = conf} {p = p} b∈ᵢl) ga≡ga of
      (λ x → lemma' {conf = conf'} {p = q}
        (sym (congjust refl refl x)))
lemma₀ {b} {l} {l'} b∈ᵢl ()    | yes _ | no _
lemma₀ {b} {l} {l'} b∈ᵢl ga≡ga | no ¬p =
  contradiction (∈ᵢ→∈ b∈ᵢl) ¬p
```

```
get≡get : ∀ {b l l₁} → GlobalUnicity l
  → get f b from l ≡ get f b from l₁ → b ∈≡ l → b ∈≡ l₁
get≡get {b} {l} _ _ with dec∈fa (f b) l
get≡get {b} {l₁ = l₁} _ _ | yes _ with dec∈fa (f b) l₁
get≡get gul eq | yes _ | yes _ =
  g≡get ∘ ((trans ∘ sym ∘ just-injective) eq) ∘ get≡g gul
get≡get _ _ | no ¬p = ⊥-elim ∘ ¬p ∘ ∈≡→∈f'
```

8.4 Publications and Seminaries

Junior workshop The initial idea about refining partial orders to model instant refinement, described in Chapter 5 was presented in the junior workshop of RTNS, JRWRTC on October the 5th, 2017 under the name "Ordering strict partial orders to model instant refinement" [115]. This consisted in a 10 minute talk coupled with a poster presentation.

National events Parts of this work have been presented twice in a french local event which unites the four laboratories working in computer science in Toulouse (ONERA, LAAS, IRT and IRIT).

- A first talk was held on April the 23rd, 2015 under the name "A formal encoding for Petri nets in AGDA " [116]. This talk described a first version of our encoding of Petri nets in AGDA which was described in Chapter 4.
- A second talk was held on April the 29th, 2017 under the name "A formal encoding of the Clock Constraint Specification Language in AGDA: Denotational semantics and Instant refinement" [117]. This talk presented the initial ideas behind the modelling of CCSL as well as ideas on how to couple it with refinement, as described in Chapters 6 and 7.

Seminaries I have been invited twice to hold seminaries around AGDA and the initial modelling of instant refinement, which led to a talk named "An AGDA introduction: Basics & Application on labelled traces". This talk was given at the following locations and dates:

- At ONERA, Toulouse, on February the 5th, 2018, invited by Claire Pagetti (ONERA) and Julien Brunel (ONERA).
- At ENS Cachan, on March the 1st, 2018, invited by Catherine Dubois (ENSIIE).

International events Finally, this work led to publications in two international conferences, at the following occasions:

- A talk was given on July the 18th, 2018 in the REFINE workshop of the FLOC conference which was held in Oxford. The talk was called "Ordering strict partial orders to model behavioural refinement" [119] and depicted the contribution about instant refinement described in Chapter 5.
- A talk was given on October the 25th, 2018 in the MEDI conference which was held in Marrakesh. The talk was named "Mechanizing the relational semantics of the Clock Constraint Specification Language" [118] and depicted the main ideas in mechanizing CCSL as described in Chapter 6.

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2), 1991.

[2] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 1996.

[3] Jean-Raymond Abrial. *Modeling in Event-B- System and Software Engineering*. Cambridge University Press, 2010.

[4] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-B. *Int. J. Softw. Tools Technol. Transf.*, 12(6):447–466, 2010.

[5] V. Alagar and K. Periyasamy. Classification of formal specification methods. January 2011.

[6] Charles André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA, 2009.

[7] Charles André and Frédéric Mallet. Clock Constraints in UML/MARTE CCSL. Research Report RR-6540, INRIA, 2008.

[8] Bui Thi Mai Anh. *Séparation des préoccupations en épidémiologie. (Separation of concerns in epidemiology)*. PhD thesis, Pierre and Marie Curie University, Paris, France, 2016.

[9] K. Appel and W. Haken. Every planar map is four colorable. part I: Discharging. *Illinois J. Math.*, 21(3):429–490, September 1977.

[10] Charles H. Applebaum and James G. Williams. PVS - design for a practical verification system. In Richard L. Muller and James J. Pottmyer, editors, *Proceedings of the 1984 ACM Annual Conference on Computer Science: The fifth generation challenge, San Francisco, CA, USA, October 1984*, pages 58–68. ACM, 1984.

[11] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The matita interactive theorem prover. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69. Springer, 2011.

[12] David Aspinall. Proof general: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000.

[13] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti : a logical framework based on the $\lambda\Pi$-calculus modulo theory. 2016.

[14] Guillaume Babin, Yamine Aït Ameur, and Marc Pantel. Formal verification of runtime compensation of web service compositions: A refinement and proof based proposal with event-B. In *2015 IEEE International Conference on Services Computing, SCC 2015, New York City, NY, USA, June 27 - July 2, 2015*, pages 98–105. IEEE Computer Society, 2015.

[15] R. J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1981.

[16] Ralph-Johan Back. Refinement calculus, part II: parallel and reactive programs. In de Bakker et al. [52], pages 67–93.

[17] Ralph-Johan Back and Joakim von Wright. Refinement calculus, part I: sequential nondeterministic programs. In de Bakker et al. [52], pages 42–66.

[18] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013.

[19] Henk Barendregt and Adrian Rezus. Semantics for classical AUTOMATH and related systems. *Information and Control*, 59(1-3):127–147, 1983.

[20] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[21] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[22] Reda Bendraou, Benoît Combemale, Xavier Crégut, and Marie-Pierre Gervais. Definition of an executable SPEM 2.0. In *14th Asia-Pacific Software Engineering Conference (APSEC 2007), 5-7 December 2007, Nagoya, Japan*, pages 390–397. IEEE Computer Society, 2007.

[23] Bernard Berthomieu and François Vernadat. Time petri nets analysis with TINA. In *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA*, pages 123–124. IEEE Computer Society, 2006.

[24] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. 2004.

[25] Frédéric Blanqui and Adam Koprowski. ColoR: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Math. Struct. Comput. Sci.*, 21(4):827–859, 2011.

[26] Frédéric Blanqui and Cody Roux. On the relation between sized-types based termination and semantic labelling, 2009.

[27] Jean-Louis Boulanger, François-Xavier Fornari, Jean-Louis Camus, and Bernard Dion. *SCADE: Language and Applications*. Wiley-IEEE Press, 1st edition, 2015.

[28] Raphael Bousso. A covariant entropy conjecture. *Journal of High Energy Physics*, 1999(07):004–004, July 1999.

[29] Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient SMT-solver. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[30] Ana Bove and Peter Dybjer. Dependent types at work. In *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, pages 57–99, 2008.

[31] Edwin Brady. IDRIS —: systems programming meets full dependent types. In Ranjit Jhala and Wouter Swierstra, editors, *Proceedings of the 5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011, Austin, TX, USA, January 29, 2011*, pages 43–54. ACM, 2011.

[32] Manfred Broy. Refinement of time. *Theor. Comput. Sci.*, 253(1):3–26, 2001.

[33] Joseph T. Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2), 1994.

[34] J. N. Buxton and B. Randell. *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. 1970.

[35] Harsh Raju Chamarthi, Peter C. Dillinger, Panagiotis Manolios, and Daron Vroon. The ACL2 sedan theorem proving system. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6605 of *Lecture Notes in Computer Science*, pages 291–295. Springer, 2011.

[36] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[37] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[38] Richard Colgren. *Basic Matlab, Simulink And Stateflow*. AIAA (American Institute of Aeronautics & Ast, 2006.

[39] Benoît Combemale, Xavier Crégut, Alain Caplain, and Bernard Coulette. Modélisation rigoureuse en SPEM de procédé de développement. In Roger Rousseau, Christelle Urtado, and Sylvain Vauttier, editors, *Actes des journéees Langages et Modèles à Objets, LMO'06. Nîmes, France, 22-24 mars*, pages 135–150. Hermès Lavoisier, 2006.

[40] Benoit Combemale, Xavier Crégut, Arnaud Dieumegard, Marc Pantel, and Faiez Zalila. Teaching MDE through the Formal Verification of Process Models. In ECEASST, editor, *7th Educators' Symposium @ MODELS 2011: Software Modeling in Education (EduSymp2011)*, Wellington, New Zealand, October 2011. Marion Brandsteidl; Andreas Winter.

[41] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on semantics definition in MDE - an instrumented approach for model verification. *JSW*, 4(9):943–958, 2009.

[42] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, Xavier Thirioux, and François Vernadat. A property-driven approach to formal verification of process models. In Joaquim Filipe, José Cordeiro, and Jorge S. Cardoso, editors, *Enterprise Information Systems, 9th International Conference, ICEIS 2007, Funchal, Madeira, Portugal, June 12-16, 2007, Revised Selected Papers*, volume 12 of *Lecture Notes in Business Information Processing*, pages 286–300. Springer, 2007.

[43] Benoît Combemale, Julien DeAntoni, Benoit Baudry, Robert B. France, Jean-Marc Jézéquel, and Jeff Gray. Globalizing modeling languages. *IEEE Computer*, 47(6), 2014.

[44] Benoît Combemale, Julien DeAntoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert B. France. Reifying concurrency for executable metamodeling. In *Software Language Engineering - 6th Intl. Conf., SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proc.*, 2013.

[45] Benoît Combemale, Pierre-Loïc Garoche, Xavier Crégut, Xavier Thirioux, and François Vernadat. Towards a formal verification of process model's properties SIMPLEPDL and TOCL case study. In Jorge S. Cardoso, José Cordeiro, and Joaquim Filipe, editors, *ICEIS 2007 - Proceedings of the Ninth International Conference on Enterprise Information Systems, Volume EIS, Funchal, Madeira, Portugal, June 12-16, 2007*, pages 80–89, 2007.

[46] Benoît Combemale, Cécile Hardebolle, Christophe Jacquet, Frédéric Boulanger, and Benoit Baudry. Bridging the chasm between executable metamodeling and models of computation. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering*, pages 184–203, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[47] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. Lightweight integration of the ergo theorem prover inside a proof assistant. *AFM'07: 2nd Workshop on Automated Formal Methods*, 11 2007.

[48] Catarina Coquand, Makoto Takeyama, and Dan Synek. An emacs-interface for type directed support constructing proofs and programs. *Electronic Notes in Theoretical Computer Science*, 2006.

[49] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.

[50] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[51] Nils Anders Danielsson and Ulf Norell. Parsing mixfix operators. In Sven-Bodo Scholz and Olaf Chitil, editors, *Implementation and Application of Functional Languages - 20th International Symposium, IFL 2008, Hatfield, UK, September 10-12, 2008. Revised Selected Papers*, volume 5836 of *Lecture Notes in Computer Science*, pages 80–99. Springer, 2008.

[52] J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors. *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop, Mook, The Netherlands, May 29 - June 2, 1989, Proceedings*, volume 430 of *Lecture Notes in Computer Science*. Springer, 1990.

[53] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[54] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015.

[55] Willem P. de Roever and Kai Engelhardt. *Data Refinement: Model-oriented Proof Theories and their Comparison*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[56] Julien Deantoni, Charles André, and Régis Gascon. CCSL denotational semantics. Research Report RR-8628, 2014.

[57] Julien DeAntoni, Papa Issa Diallo, Ciprian Teodorov, Joël Champeau, and Benoît Combemale. Towards a meta-language for the concurrency concern in DSLs. In *Proc. of the 2015 Design, Automation & Test in Europe Conf. & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, 2015.

[58] Julien Deantoni and Frédéric Mallet. TimeSquare: Treat your Models with Logical Time. In *TOOLS - 50th International Conference on Objects, Models, Components, Patterns - 2012*, 2012.

[59] Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.

[60] Arnaud Dieumegard, Andres Toom, and Marc Pantel. Formal specification of block libraries in dataflow languages. In *Embedded Real Time Software and Systems (ERTS2014)*, Toulouse, France, February 2014.

[61] Edsger W. Dijkstra. *On the Role of Scientific Thought*, pages 60–66. Springer New York, New York, NY, 1982.

[62] Albert Einstein. Zur Elektrodynamik bewegter Körper. (German) [On the electrodynamics of moving bodies]. *j-ANN-PHYS-1900-4*, 322(10):891–921, 1905.

[63] ClearSy System Engineering. *Atelier B - User Manual*. Aix-en-Provence.

[64] Daniel P. Friedman et al. *The little typer*. The MIT Press, 2018.

[65] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 1st edition, 2012.

[66] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[67] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[68] Simon Foster and Georg Struth. Integrating an automated theorem prover into agda. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011.

[69] Martin Fowler. *Domain-Specific Languages*. The Addison-Wesley signature series. Addison-Wesley, 2011.

[70] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. Keymaera X: An axiomatic tactical theorem prover for hybrid systems. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 527–538, Cham, 2015. Springer International Publishing.

[71] Manuel Garnacho, Jean-Paul Bodeveix, and Mamoun Filali-Amine. A mechanized semantic framework for real-time systems. In *Formal Modeling and Analysis of Timed Systems - 11th Intl. Conf., FORMATS 2013, Buenos Aires, Argentina, August 29-31, 2013. Proc.*, 2013.

[72] Mike Gemünde, Jens Brandt, and Klaus Schneider. Clock refinement in imperative synchronous languages. *EURASIP J. Emb. Sys.*, 2013, 2013.

[73] E. Giménez. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants.* PhD thesis, Ecole Normale Supérieure de Lyon, 1996.

[74] Kurt Godel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover Publications, 1992. Translation B. Meltzer.

[75] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013.

[76] Roger Hale, Rachel Cardell-Oliver, and John Herbert. An embedding of timed transition systems in HOL. *Formal Methods in System Design*, 3(1/2), 1993.

[77] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the kepler conjecture. *CoRR*, 2015.

[78] Cécile Hardebolle and Frédéric Boulanger. Modhel'X: A component-oriented approach to multi-formalism modeling. In *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*, 2007.

[79] Jifeng He. Process simulation and refinement. *Formal Asp. Comput.*, 1(3):229–241, 1989.

[80] Wim H. Hesselink. Simulation refinement for concurrency verification. *Sci. Comput. Program.*, 76(9):739–755, 2011.

[81] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[82] Alex Horn and Daniel Kroening. On partial order semantics for SAT/SMT-based symbolic encodings of weak memory concurrency. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble,*

France, June 2-4, 2015, Proceedings, volume 9039 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2015.

[83] William A. Howard. The formulae-as-types notion of construction. 1969.

[84] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types, 1996.

[85] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[86] Mounira Kezadri, Benoît Combemale, Marc Pantel, and Xavier Thirioux. A proof assistant based formalization of MDE components. In Farhad Arbab and Peter Csaba Ölveczky, editors, *Formal Aspects of Component Software - 8th International Symposium, FACS 2011, Oslo, Norway, September 14-16, 2011, Revised Selected Papers*, volume 7253 of *Lecture Notes in Computer Science*, pages 223–240. Springer, 2011.

[87] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009.

[88] Philip Koopman. A case study of toyota unintended acceleration and software safety, November 2014.

[89] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[90] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[91] Gérard Le Lann. An analysis of the ariane 5 flight 501 failure-a system engineering perspective. In *1997 Workshop on Engineering of Computer-Based Systems (ECBS '97), March 24-28, 1997, Monterey, CA, USA*, pages 339–246. IEEE Computer Society, 1997.

[92] Matias Ezequiel Vara Larsen, Julien DeAntoni, Benoît Combemale, and Frédéric Mallet. A behavioral coordination operator language (BCOoL). In *18th ACM/IEEE Intl. Conf. on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, 2015.

[93] Florent Latombe, Xavier Crégut, Benoît Combemale, Julien DeAntoni, and Marc Pantel. Weaving concurrency in executable domain-specific modeling

languages. In *Proc. of the 2015 ACM SIGPLAN Intl. Conf. on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, 2015.

[94] Yves Ledru, Akram Idani, Rahma Ben Ayed, Abderrahim Ait Wakrime, and Philippe Bon. A separation of concerns approach for the verified modelling of railway signalling rules. In Simon Collart Dutilleul, Thierry Lecomte, and Alexander B. Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - Third International Conference, RSSRail 2019, Lille, France, June 4-6, 2019, Proceedings*, volume 11495 of *Lecture Notes in Computer Science*, pages 173–190. Springer, 2019.

[95] Edward A. Lee. Cyber physical systems: Design challenges. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), 5-7 May 2008, Orlando, Florida, USA*, pages 363–369. IEEE Computer Society, 2008.

[96] Xavier Leroy. Formal verification of a realistic compiler. 52:107, 2009.

[97] Xavier Leroy. Programmer = démontrer ? la correspondance de curry-howard aujourd'hui, 2018.

[98] Michael Leuschel. *ProB User Manual*, 2011.

[99] Michael Leuschel and Michael J. Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer, 2003.

[100] Nancy G. Leveson and Clark Savage Turner. Investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[101] Ming Li and Paul M. B. Vitányi. *An introduction to Kolmogorov complexity and its applications*. Texts and Monographs in Computer Science. Springer, 1993.

[102] Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in agda. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2004.

[103] Zhaohui Luo. *A unifying theory of dependent types: the schematic approach*, volume 620, pages 293–304. October 2006.

[104] Gerald Lüttgen. Modeling and verification using UML statecharts. by doron drusinsky. published by newnes publishers, 2006, ISBN 0-7506-7617-5, 306 pages. *Softw. Test. Verification Reliab.*, 18(3):189–190, 2008.

[105] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[106] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations, II: timing-based systems. *Inf. Comput.*, 128(1):1–25, 1996.

[107] Jan Malakhovski. Brutal [meta]introduction to dependent types in agda. `https://web.archive.org/web/20180730183426/http://oxij.org/note/BrutalDepTypes/`. Published: 2013-03-14.

[108] Louis Mandel, Cédric Pasteur, and Marc Pouzet. Time refinement in a functional synchronous language. *Sci. Comput. Program.*, 111, 2015.

[109] P. Martin-Löf and G. Sambin. *Intuitionistic Type Theory*. Lecture notes. Bibliopolis, 1984.

[110] Raphaël Marvie. *Séparation des préoccupations et méta-modélisation pour environnements de manipulation d'architectures logicielles à base de composants. (Separation of Concerns and Metamodeling applied to Software Architecture Handling)*. PhD thesis, Lille University of Science and Technology, France, 2002.

[111] P. Matyasik and M. Szpyrka. Formal modelling and verification of concurrent systems with XCCS. In *2008 International Symposium on Parallel and Distributed Computing*, 2008.

[112] Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming, 5th International School, AFP 2004, Tartu, Estonia, August 14-21, 2004, Revised Lectures*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2004.

[113] Jan Mikac and Paul Caspi. Temporal refinement for lustre. In *Proc. of the 5th Intl. Workshop on Synchronous Languages, Applications and Programs, Edimburg, April 2005*, 2005.

[114] Andrey Mokhov. Algebraic graphs with class (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, pages 2–13, New York, NY, USA, 2017. Association for Computing Machinery.

[115] Montin and Pantel. Ordering strict partial orders to model behavioural refinement. 2017.

[116] Mathieu Montin and Marc Pantel. A formal encoding for petri nets in agda. 2015.

[117] Mathieu Montin and Marc Pantel. A formal encoding of clock constraints specification language in agda : Denotational semantics and instant refinement. 2017.

[118] Mathieu Montin and Marc Pantel. Mechanizing the denotational semantics of the clock constraint specification language. In El Hassan Abdelwahed, Ladjel Bellatreche, Matteo Golfarelli, Dominique Méry, and Carlos Ordonez, editors, *Model and Data Engineering - 8th International Conference, MEDI 2018, Marrakesh, Morocco, October 24-26, 2018, Proceedings*, volume 11163 of *Lecture Notes in Computer Science*, pages 385–400. Springer, 2018.

[119] Mathieu Montin and Marc Pantel. Ordering strict partial orders to model behavioral refinement. In John Derrick, Brijesh Dongol, and Steve Reeves, editors, *Proceedings 18th Refinement Workshop, Refine@FM 2018, Oxford, UK, 18th July 2018.*, volume 282 of *EPTCS*, pages 23–38, 2018.

[120] Márcio Ferreira Moreno, Rodrigo Costa Mesquita Santos, Guilherme Augusto Ferreira Lima, Marcelo Ferreira Moreno, and Luiz Fernando Gomes Soares. Deepening the separation of concerns in the implementation of multimedia systems. In Sascha Ossowski, editor, *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, pages 1337–1343. ACM, 2016.

[121] Carroll Morgan and Paul H. B. Gardiner. Data refinement by calculation. *Acta Inf.*, 27(6):481–503, 1990.

[122] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Not.*, 8(8):12–26, August 1973.

[123] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[124] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[125] Ulf Norell. Dependently typed programming in agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008.

[126] Ulf Norell. Programming in agda, 2014.

[127] Object Management Group (OMG). Object constraint language, February 2014.

[128] Object Management Group (OMG). Unified modeling language, December 2017.

[129] Object Management Group (OMG). UML profile for MARTE, April 2019.

[130] Christine Paulin-Mohring. Modelisation of timed automata in coq. In *Theoretical Aspects of Computer Software, 4th Intl. Symp., TACS 2001, Sendai, Japan, October 29-31, 2001, Proc.*, 2001.

[131] Lawrence C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5, 1989.

[132] Carl Adam Petri. Kommunikation mit Automaten. Dissertation, Schriften des IIM 2, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, Bonn, 1962.

[133] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977.

[134] Steve Reeves and David Streader. Comparison of data and process refinement. In Jin Song Dong and Jim Woodcock, editors, *Formal Methods and Software Engineering, 5th International Conference on Formal Engineering Methods, ICFEM 2003, Singapore, November 5-7, 2003, Proceedings*, volume 2885 of *Lecture Notes in Computer Science*, pages 266–285. Springer, 2003.

[135] Steve Reeves and David Streader. General refinement, part one: Interfaces, determinism and special refinement. *Electr. Notes Theor. Comput. Sci.*, 214:277–307, 2008.

[136] Alexandre Riazanov and Andrei Voronkov. Vampire. In Harald Ganzinger, editor, *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, volume 1632 of *Lecture Notes in Computer Science*, pages 292–296. Springer, 1999.

[137] B. Russell. *The Principles of Mathematics*. Number v. 1 in The Principles of Mathematics. University Press, 1903.

[138] John E. Savage. *Models of computation - exploring the power of computing*. Addison-Wesley, 1998.

[139] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theor. Comput. Sci.*, 121(1&2):411–440, 1993.

[140] Henny B. Sipma, Tomás E. Uribe, and Zohar Manna. Deductive model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, pages 208–219, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[141] G. M. Skii and Ye. M. Landis. An algorithm for the organization of information. 1962.

[142] Jean-Pierre Talpin, Paul Le Guernic, Sandeep K. Shukla, Frederic Doucet, and Rajesh K. Gupta. Formal refinement checking in a system-level design methodology. *Fundam. Inform.*, 62(2):243–273, 2004.

[143] Michael Tiller. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, Boston, 2001.

[144] Paul Joseph Trafford. *The use of formal methods for safety-critical systems*. PhD thesis, Kingston University, Kingston upon Thames, London, UK, 1997.

[145] Bob Travica. Mediating realities: A case of the boeing 737 MAX. *Informing Sci. Int. J. an Emerg. Transdiscipl.*, 23:25–46, 2020.

[146] Jan Tretmans. Testing concurrent systems: A formal approach. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR'99 Concurrency Theory*, pages 46–65, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[147] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[148] Hai Nguyen Van, Thibaut Balabonski, Frédéric Boulanger, Chantal Keller, Benoît Valiron, and Burkhart Wolff. A symbolic operational semantics for TESL - with an application to heterogeneous system testing. In Alessandro Abate and Gilles Geeraerts, editors, *Formal Modeling and Analysis of Timed Systems - 15th International Conference, FORMATS 2017, Berlin, Germany, September 5-7, 2017, Proceedings*, volume 10419 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 2017.

[149] Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in agda. In Ralf Hinze, editor, *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, volume 8241 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2012.

[150] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*. MIT Press, Cambridge, MA, USA, 1991.

[151] Matthias Weber. Model-based development in automotive electronics – the EAST-ADL. In Carsten Gremzow and Nico Moser, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Berlin, Germany, March 2-4, 2009*, pages 4–4. Universitätsbibliothek Berlin, Germany, 2009.

[152] Glynn Winskel. Event structures. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proc. of an Advanced Course, Bad Honnef, 8.-19. September 1986*, 1986.

[153] Niklaus Wirth. A brief history of software engineering. *IEEE Annals of the History of Computing*, 30(3):32–39, 2008.

[154] Chenchen Xi. *Separation of concerns in concurrent programs using fine grained join points*. PhD thesis, University of Manchester, UK, 2012.

[155] Luca Zamboni. *Getting Started with Simulink*. Packt Publishing, 2013.

[156] H. Zantema. Termination of term rewriting by semantic labelling. *Fundam. Inf.*, 24(1–2):89–105, April 1995.

[157] Min Zhang, Fu Song, Frédéric Mallet, and Xiaohong Chen. SMT-based bounded schedulability analysis of the clock constraint specification language. In Reiner Hähnle and Wil M. P. van der Aalst, editors, *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11424 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 2019.

Appendices

A.1 On conformity of lists

A.1.1 Decidability of the membership relation

We provide proofs of decidability for our predicates of membership, unique membership and their negations. These proofs use the function `¬?` which proves the decidability of a predicate `¬ P` from a proof of decidability of `P`.

Membership relations are decidable when `_R_` is decidable:

```
dec∈ : Decidableᵣ _R_ → Decidableᵣ _∈_
dec∈ = any ∘ _
-
dec∉ : Decidableᵣ _R_ → Decidableᵣ _∉_
dec∉ dec x = ¬? ∘ (dec∈ dec x)
```

Unique membership is decidable when `_R_` is decidable:

```
dec∈! : Decidableᵣ _R_ → Decidableᵣ _∈!_
dec∈! = decAny! ∘ _
-
dec∉! : Decidableᵣ _R_ → Decidableᵣ _∉!_
dec∉! dec x = ¬? ∘ (dec∈! dec x)
```

A.1.2 Decidability of the none or one relation

```
Dec∈∈! decR x l with dec∈! decR x l
Dec∈∈! _ _ _ | yes p = yes (λ _ → p)
Dec∈∈! decR x l | no _ with dec∈ decR x l
Dec∈∈! _ _ _ | no ¬x∈!l | yes x∈l = no (¬x∈!l ∘ (_$ x∈l))
Dec∈∈! _ _ _ | no _ | no ¬x∈l = yes (⊥-elim ∘ ¬x∈l)
```

A.1.3 Lemmas about membership of characters

```
1  open import Data.List.Membership.Setoid Data.Char.Properties.≡-setoid  1
2  -                                                                      2
3  _∈l_ : _ → _ → _                                                       3
4  c ∈l s = c ∈ toList s                                                  4
5  -                                                                      5
6  _∉l_ : _ → _ → _                                                       6
7  c ∉l s = c ∉ toList s                                                  7
8  -                                                                      8
9  z∉Alice : 'z' ∉l "Alice"                                               9
10 z∉Alice = (λ ()) ↦ (λ ()) ↦ (λ ()) ↦ (λ ()) ↦ (λ ()) ↦ ¬A[]            10
11 -                                                                      11
12 z∉Bob : 'z' ∉l "Bob"                                                   12
13 z∉Bob = (λ ()) ↦ (λ ()) ↦ (λ ()) ↦ ¬A[]                                13
14 -                                                                      14
15 z∉Judith : 'z' ∉l "Judith"                                             15
16 z∉Judith = (λ ()) ↦ (λ ()) ↦ (λ ()) ↦ (λ ()) ↦ (λ ()) ↦ (λ ()) ↦ ¬A[]  16
17 -                                                                      17
18 J∉Alice : 'J' ∉l "Alice"                                               18
19 J∉Alice = (λ ()) ↦ (λ ()) ↦ (λ ()) ↦ (λ ()) ↦ (λ ()) ↦ ¬A[]            19
20 -                                                                      20
21 J∉Bob : 'J' ∉l "Bob"                                                   21
22 J∉Bob = (λ ()) ↦ (λ ()) ↦ (λ ()) ↦ ¬A[]                                22
23 -                                                                      23
24 J∈Judith : 'J' ∈l "Judith"                                             24
25 J∈Judith = here refl                                                   25
```

A.1.4 Lemmas about global unicity and equivalence

Membership with _≡_ implies membership with reflexive relations:

```
1  ∈→∈≈ : ∀ {x l} → x ∈ l → x ∈≈ l                1
2  ∈→∈≈ (here refl) = here (≈refl isEq)          2
3  ∈→∈≈ (there x∈l) = there (∈→∈≈ x∈l)           3
```

A proof of indexes preservation:

```
4  i≡ic : ∀ {x l} (p : x ∈ l) → index p ≡ index (∈→∈≈ p)   4
5  i≡ic (here refl) = refl                                  5
6  i≡ic (there p) = cong fsuc (i≡ic p)                      6
```

Two equivalent instants appear in the same lists:

```
7   trans∈≈ : ∀ {x y l} → x ≈ y → x ∈≈ l → y ∈≈ l                          7
8   trans∈≈ x≈y (here px) = here (≈trans isEq (≈sym isEq x≈y) px)           8
9   trans∈≈ x≈y (there x∈≈l) = there (trans∈≈ x≈y x∈≈l)                     9
```

Another proof of indexes preservation:

```
10  it≡i : ∀ {x y l} (p : x ≈ y) (q : x ∈≈ l)                               10
11    → index (trans∈≈ p q) ≡ index q                                       11
12  it≡i p (here px) = refl                                                 12
13  it≡i p (there q) = cong fsuc (it≡i p q)                                 13
```

If the indexes are the same, then the elements are the same:

```
14  ≡i→≡ : ∀ {x y l} {x∈l : x ∈ l} {y∈l : y ∈ l}                            14
15    → index x∈l ≡ index y∈l → x ≡ y                                       15
16  ≡i→≡ {x∈l = here refl} {here refl} refl = refl                          16
17  ≡i→≡ {x∈l = here px} {there y∈l} ()                                     17
18  ≡i→≡ {x∈l = there x∈l} {here px} ()                                     18
19  ≡i→≡ {x∈l = there x∈l} {there y∈l} u = ≡i→≡ (fsuc-injective u)          19
```

A.1.5 Lemmas about assignment and membership

These lemmas show that assignments leave the contents of the list unchanged. They are prove by case-splitting on the proofs of memberships.

Assigning a value does not create new elements:

```
1   ∈assign : ∀ {a b l p} → a ∈ (assign b inside l when p) → a ∈ l         1
2   ∈assign {p = here p} (here q) = here (trans q p)                        2
3   ∈assign {p = here _} (there q) = there q                                3
4   ∈assign {p = there _} (here q) = here q                                 4
5   ∈assign {p = there _} (there q) = there (∈assign q)                     5
```

Assigning a value does not remove any element:

```
6   assign∈ : ∀ {a b l p} → a ∈ l → a ∈ (assign b inside l when p)         6
7   assign∈ {p = here p} (here q) = here (trans q (sym p))                  7
8   assign∈ {p = here _} (there q) = there q                                8
9   assign∈ {p = there _} (here q) = here q                                 9
10  assign∈ {p = there _} (there q) = there (assign∈ q)                     10
```

Here are the counterparts for unique membership:

```
11  assign∈! : ∀ {a b l p} → a ∈! l → a ∈! (assign b inside l when p)            11
12  assign∈! {p = here p} (here! x y) = here! (trans x (sym p)) y                  12
13  assign∈! {p = here p} (there! x y) = there! (x ∘ (flip trans) p) y            13
14  assign∈! {p = there _} (here! x y) = here! x (contraposition ∈assign y)       14
15  assign∈! {p = there _} (there! x y) = there! x (assign∈! y)                    15
16  -                                                                              16
17  ∈!assign : ∀ {a b l p} → a ∈! (assign b inside l when p) → a ∈! l            17
18  ∈!assign {p = here p} (here! x y) = here! (trans x p) y                        18
19  ∈!assign {p = here p} (there! x y) = there! (x ∘ (flip trans) (sym p)) y      19
20  ∈!assign {p = there _} (here! x y) = here! x (contraposition assign∈ y)       20
21  ∈!assign {p = there _} (there! x y) = there! x (∈!assign y)                    21
```

A.1.6 Comparison between globally unique lists

Comparison of lists Module header and definition of inclusion and equivalence:

```
1   module ListInclusion {a ℓ} {A : Set a} {_≈_ : Rel A ℓ}                        1
2     (dec≈ : Decidableᵣ _≈_) (eq≈ : IsEquivalence _≈_) where                      2
3     open Membership _≈_                                                         3
4     -                                                                           4
5     _⊑_ : Rel (List A) _                                                        5
6     _⊑_ = _⊆_ on flip _∈_                                                       6
7     -                                                                           7
8     _≋_ : Rel (List A) _                                                        8
9     _≋_ = _⊑_ -[ _×_ ]- flip _⊑_                                                9
```

A proof of preservation of membership:

```
10  conserv∈≈ : ∀ {x y l} → x ∈ l → y ≈ x → y ∈ l                                10
11  conserv∈≈ (here px) = here ∘ flip (trans≈ eq≈) px                             11
12  conserv∈≈ (there p) = there ∘ conserv∈≈ p                                     12
```

Proof of decidability of inclusion:

```
13  dec⊑ : Decidableᵣ _⊑_                                                          13
14  dec⊑ [] l₂ = yes λ()                                                           14
15  dec⊑ (x :: l₁) l₂ with dec∈ dec≈ x l₂ | dec⊑ l₁ l₂                             15
16  dec⊑ _ _ | yes p | (yes p₁) =                                                  16
17    yes (λ {(here px) → conserv∈≈ p px ; (there x₂) → p₁ x₂})                    17
18  dec⊑ _ _ | yes _ | (no ¬p ) = no (λ x₁ → ¬p (λ x₃ → x₁ (there x₃)))            18
19  dec⊑ _ _ | no ¬p | _ = no (λ x₁ → ¬p (x₁ (here (refl≈ eq≈))))                  19
```

Proof of decidability of equivalence:

```
20  dec≈ : Decidable_r _≈_                                        20
21  dec≈ l1 l2 with dec⊑ l1 l2 | dec⊑ l2 l1                        21
22  dec≈ _ _ | yes p | (yes p1) = yes (p , p1)                     22
23  dec≈ _ _ | yes p | (no ¬p ) = no (¬p ∘ proj2)                  23
24  dec≈ _ _ | no ¬p | _ = no (¬p ∘ proj1)                         24
```

Proof of equivalence:

```
25  isEq≈ : IsEquivalence _≈_                                      25
26  isEq≈ = record {                                               26
27    refl = id , id ;                                             27
28    sym = swap ;                                                 28
29    trans = zip (\f g → g ∘ f) \f g → f ∘ g }                    29
```

Comparison of GUList header of the module:

```
30  module GUCompare {a b c} {A : Set a} {B : Set b} {C : Set c}   30
31    (f : B → A) (_≟a_ : Decidable_r {A = A} _≡_) (g : B → C)      31
32    (injfg : ∀ {x y} → (f x , g x) ≡ (f y , g y) → x ≡ y)        32
33    (_≟b_ : Decidable_r {A = B} _≡_) where                       33
34    -                                                            34
35    open Commands f _≟a_ g                                       35
36    open ListInclusion _≟b_ isEquivalence                        36
37    open FunctionRelation f                                      37
38    open Membership _≡f_ renaming (_∈_ to _∈f_ ; dec∈ to dec∈f)  38
39    open Membership {A = B} _≡_                                  39
40      renaming (_∈_ to _∈≡_ ; dec∈ to dec∈≡)                     40
41    open GlobalUnicity _≡f_                                      41
42    open GUList                                                  42
```

A small macro of decidability:

```
43  dec∈fa : Decidable_r _                                         43
44  dec∈fa = (dec∈f ∘ dec≡f) _≟a_                                  44
```

First way of comparing the `GULists` using the list equality over their contents:

```
45  _≈1_ : Rel GUList _                                            45
46  _≈1_ = _≈_ on content                                          46
```

Proof of equivalence of this relation:

```
47  eq≈₁ : IsEquivalence _≈₁_
48  eq≈₁ with isEq≈
49  eq≈₁ | record { refl = refl≈ ; sym = sym≈ ; trans = trans≈ }
50    = record { refl = refl≈ ; sym = sym≈ ; trans = trans≈ }
```

Proof of decidability of this relation:

```
51  dec≈₁ : Decidableᵣ _≈₁_
52  dec≈₁ (gulist l₁ _) (gulist l₂ _) = dec≈ l₁ l₂
```

Second way of comparing `GULists` through the equality of the return values of the `get` function:

```
53  _≈₂_ : Rel GUList _
54  gu₁ ≈₂ gu₂ = ∀ {x} → get x gu₁ ≡ get x gu₂
```

Proof that this relation is an equivalence:

```
55  eq≈₂ : IsEquivalence _≈₂_
56  eq≈₂ = record
57    { refl = refl ;
58      sym = λ x → sym x ;
59      trans = λ x x₁ → trans x x₁ }
```

A proof of preservation of global unicity:

```
60  gu≡ : ∀ {b₀ b₁ l} → GlobalUnicity (b₀ :: l) → f b₁ ≡ f b₀
61    → GlobalUnicity (b₁ :: l)
62  gu≡ gulb₀ eq (here refl) =
63    here! refl (contradiction (gulb₀ (here eq)) ∘ (P→A→¬A! eq))
64  gu≡ gulb₀ eq (there q) =
65    there! (λ p → ¬A!→P→¬A
66      (gulb₀ (there q)) (trans p eq) q) (guTl gulb₀ q)
```

Lemmas about membership:

```
67  ∈≡→∈f : ∀ {b b₁ l} → f b ≡ f b₁ → b ∈≡ l → (f b₁) ∈f l
68  ∈≡→∈f px = mapAny λ {refl → sym px}
69  -
70  ∈≡→∈f' : ∀ {b l} → b ∈≡ l → (f b) ∈f l
71  ∈≡→∈f' = ∈≡→∈f refl
```

A proof about the result of `get`:

```
72  get≡g : ∀ {b l p} → GlobalUnicity l → b ∈≡ l                     72
73    → get f b from l when p ≡ g b                                   73
74  get≡g {p = here refl} _ (here refl) = refl                        74
75  get≡g {p = here px} gul (there b∈≡l) =                            75
76    contradiction (gul (here refl))                                 76
77      (P→A→¬A! refl (∈≡→∈f px b∈≡l))                                77
78  get≡g {p = there p} gul (here refl) =                             78
79    contradiction (gul (here refl))                                 79
80      (P→A→¬A! refl p)                                              80
81  get≡g {p = there p} gul (there b∈≡l) = get≡g (guTl gul) b∈≡l      81
```

The inverse proof about the result of `get`:

```
82  g≡get : ∀ {b l p} → get f b from l when p ≡ g b → b ∈≡ l          82
83  g≡get {p = here px} = here ∘ injfg ∘ (cong₂ _,_ px) ∘ sym          83
84  g≡get {p = there p} = there ∘ g≡get                                84
```

Proof of preservation of membership through `get`:

```
85  get≡get : ∀ {b l l₁} → GlobalUnicity l                            85
86    → get f b from l ≡ get f b from l₁ → b ∈≡ l → b ∈≡ l₁            86
87  get≡get {b} {l} _ _ with dec∈fa (f b) l                            87
88  get≡get {b} {l₁ = l₁} _ _ | yes _ with dec∈fa (f b) l₁             88
89  get≡get gul eq | yes _ | yes _ =                                   89
90    g≡get ∘ ((trans ∘ sym ∘ just-injective) eq) ∘ get≡g gul          90
91  get≡get _ _ | no ¬p = ⊥-elim ∘ ¬p ∘ ∈≡→∈f'                         91
```

The second comparison implies the first one:

```
92  ≈₂→≈₁ : ∀ {gu₁ gu₂} → gu₁ ≈₂ gu₂ → gu₁ ≈₁ gu₂                      92
93  ≈₂→≈₁ {gu₁} {gu₂} gu₁≈₂gu₂ =                                       93
94    let gu₂≈₂gu₁ = (IsEquivalence.sym eq≈₂) {gu₁} {gu₂} gu₁≈₂gu₂ in  94
95    (get≡get (unique gu₁) gu₁≈₂gu₂) ,                                95
96    get≡get (unique gu₂) gu₂≈₂gu₁                                    96
```

An element with the right properties can be retrieved:

```
97  retrieve : ∀ {x l} → GlobalUnicity l → (p : x ∈f l)                97
98    → ∃ λ y → y ∈≡ l × x ≡ f y × get x from l when p ≡ g y           98
99  retrieve _ (here refl) = _ , here refl , refl , refl               99
100 retrieve gul (there x∈fl) = map₂ (map₁ there) (retrieve (guTl gul) x∈fl)  100
```

The first comparison implies the second one, thus they are equivalent:

```
101  ≈₁→≈₂ : ∀ {gu₁ gu₂} → gu₁ ≈₁ gu₂ → gu₁ ≈₂ gu₂                         101
102  ≈₁→≈₂ {gulist l₁ _} {gulist l₂ _} _ {x} with dec∈fa x l₁ | dec∈fa x l₂  102
103  ≈₁→≈₂ {gulist _ gul₁} {gulist _ gul₂} (l₁⊑l₂ , _) | yes p | yes _ =     103
104    case retrieve gul₁ p of                                              104
105      λ {(_ , q , refl , ≡get) →                                         105
106        ((cong just) ∘ (trans ≡get) ∘ sym ∘ (get≡g gul₂) ∘ l₁⊑l₂) q}     106
107  ≈₁→≈₂ {gulist _ gul₁} (l₁⊑l₂ , _) | yes p | no ¬p =                    107
108    ⊥-elim (case retrieve gul₁ p of                                      108
109      λ {(_ , q , refl , _) → (¬p ∘ ∈≡→∈f' ∘ l₁⊑l₂) q})                  109
110  ≈₁→≈₂ {_} {gulist _ gul₂} (_ , l₂⊑l₁) | no ¬p | yes p =                110
111    ⊥-elim (case retrieve gul₂ p of                                      111
112      λ {(_ , q , refl , _) → (¬p ∘ ∈≡→∈f' ∘ l₂⊑l₁) q})                  112
113  ≈₁→≈₂ _ | no _ | no _ = refl                                           113
```

A proof of decidability of the second comparison using this equivalence:

```
114  dec≈₂ : Decidableᵣ _≈₂_                                                114
115  dec≈₂ gu₁ gu₂ with dec≈₁ gu₁ gu₂                                       115
116  dec≈₂ gu₁ gu₂ | yes p = yes (≈₁→≈₂ {gu₁} {gu₂} p)                      116
117  dec≈₂ gu₁ gu₂ | no ¬p = no (¬p ∘ ≈₂→≈₁ {gu₁} {gu₂})                    117
```

A.1.7 Trimming a non-empty globally unique list

```
1  trim : (gul : GUList) → ¬ (GUList.content gul) ≡ [] → GUList           1
2  trim (gulist [] unique) p = ⊥-elim (p refl)                            2
3  trim (gulist (_ :: content) unique) _ = gulist content (guTl unique)   3
```

A.2 On Petri nets and SimplePDL models

A.2.1 Lemmas to add an arc to a net

If all the elements of the map satisfy the same property, and if we add an element in this map which satisfies the same property, then all the elements of the resulting map satisfy this property:

```
1  putpres : ∀ {m₁ : Map {B = ℕ × ℕ}} {m₂ : Map {B = ℕ}} {k v}                  1
2    {¬k∈m₁ : ¬ k ∈ₗ m₁} (k∈m₂ : k ∈ₗ m₂) →                                       2
3    (∀ {x} → x ∈ₗ m₁ → x ∈ₗ m₂) →                                                3
4    (∀ {x} → x ∈ₗ (put k , v into m₁ when ¬k∈m₁) → x ∈ₗ m₂)                       4
5  putpres k∈m₂ _ (here refl) = k∈m₂                                              5
6  putpres _ m₁⊑m₂ (there x∈put) = m₁⊑m₂ x∈put                                    6
```

Retrieving an element from a list in which we assigned a value to an element e either returns e or an element that was previously in the map. This is established in two steps, the first one being an equality proof and the second one an adapted form used in the addition of an arc to a net:

```
7  prop₀ : ∀ {m₁ : Map {B = Map {B = ℕ × ℕ}}}                                     7
8    {m₂ : Map {B = ℕ × ℕ}} {k x} {k∈m₁ x∈assm₁} →                                8
9    get x from (assign k , m₂ inside m₁ if k∈m₁) if x∈assm₁ ≡                     9
10     get x from m₁ if ∈assign x∈assm₁ ⊎                                          10
11   get x from (assign k , m₂ inside m₁ if k∈m₁) if x∈assm₁ ≡ m₂                  11
12 prop₀ {k∈m₁ = here _} {here _} = inj₂ refl                                     12
13 prop₀ {k∈m₁ = here _} {there _} = inj₁ refl                                    13
14 prop₀ {k∈m₁ = there _} {here _} = inj₁ refl                                    14
15 prop₀ {m₁} {k∈m₁ = there k∈m₁} {there x∈assm₁} =                               15
16   prop₀ {m₁ = trim m₁ λ ()} {k∈m₁ = k∈m₁} {x∈assm₁}                            16
17 -                                                                               17
18 prop₂ : ∀ {m₁ : Map {B = Map {B = ℕ × ℕ}}}                                     18
19   {m₂ : Map {B = ℕ}} {m₃ : Map {B = ℕ × ℕ}} {k} {k∈m₁} →                       19
20   (∀ {x} → x ∈ₗ m₃ → x ∈ₗ m₂) →                                                20
21   (∀ {x x∈m₁ p} → p ∈ₗ (get x from m₁ if x∈m₁) → p ∈ₗ m₂) →                    21
22   ∀ {x x∈assm₁ p} →                                                            22
23   p ∈ₗ (get x from assign k , m₃ inside m₁ if k∈m₁ if x∈assm₁) →                23
24   p ∈ₗ m₂                                                                      24
25 prop₂ {m₁} {m₂} {m₃} {k} {k∈m₁} p q {x} {x∈assm₁} {r}                          25
26   with prop₀ {m₁} {m₃} {k} {x} {k∈m₁} {x∈assm₁}                                26
27 prop₂ _ q | inj₁ x₁ rewrite x₁ = q                                             27
28 prop₂ p _ | inj₂ y rewrite y = p                                               28
```

A.2.2 Exporting nets to TINA

```
1  showl : ∀ {a} {A B : Set a} f g → Fun2 String → List (A × B) → String       1
2  showl f g h = foldr ((uncurry h ∘ (map∑ f g)) -⟦ _++_ ⟧- ("\n" ++_)) ""    2
3  -                                                                          3
4  showp : List (String × ℕ) → String                                          4
5  showp = showl id showℕ (λ s0 s1 → "pl " ++ s0 ++ " (" ++ s1 ++ ")")        5
6  -                                                                          6
7  Mshowp : Map → String                                                      7
8  Mshowp = showp ∘ Map.content                                               8
9  -                                                                          9
10 aggregate : List (String × ℕ) → String                                     10
11 aggregate = foldr ((λ {(_ , zero) → "" ; (s , suc zero) → s ++ " " ;       11
12   (s , suc (suc n)) → s ++ "*"                                             12
13     ++ showℕ (suc (suc n)) ++ " "}) -⟦ _++_ ⟧- id) ""                      13
14 -                                                                          14
15 splito : List (String × (ℕ × ℕ)) → String                                  15
16 splito l = aggregate (mapl (λ {(a , b , _) → a , b}) l) ++ " -> " ++       16
17   aggregate (mapl (λ {(a , _ , c) → a , c}) l)                             17
18 -                                                                          18
19 splitom : Map → String                                                     19
20 splitom = splito ∘ Map.content                                             20
21 -                                                                          21
22 showt : List (String × Map) → String                                       22
23 showt = showl id splitom (λ s0 s1 → "tr " ++ s0 ++ " " ++ s1)              23
24 -                                                                          24
25 Mshowt : Map → String                                                      25
26 Mshowt = showt ∘ Map.content                                               26
27 -                                                                          27
28 MPshow : Petrinet → String                                                 28
29 MPshow p = Mshowp (marking p) ++ "\n" ++ Mshowt (transitions p)            29
30 -                                                                          30
31 MPshowm : Maybe Petrinet → String                                          31
32 MPshowm (just x) = MPshow x                                                32
33 MPshowm nothing = "Unsound net!"                                           33
34 -                                                                          34
35 printp : Maybe Petrinet → IO ⊤                                             35
36 printp = putStrLn ∘ toCostring ∘ MPshowm                                   36
37 -                                                                          37
38 outp : String → Maybe Petrinet → IO ⊤                                      38
39 outp name = (writeFile name) ∘ toCostring ∘ MPshowm                        39
```

A.2.3 Decidability from a list of candidates

If the Any predicate holds for the result of a get operation over a map, then this value necessarily is a member of this map, since it means that the get operation returned `just` something. Note that in this property, pattern matching on `dec∈ₗ x m` only yields the possibility `yes q` because AGDA understood that the case `no q` was no possible since it would lead to p being empty:

```
1   getp : ∀ {x} {ℓ b} {B : Set b} {P : Pred B ℓ} {m : Map {B = B}}
2     → Any P (get x m) → x ∈ₗ m
3   getp {x} {m = m} _ with dec∈ₗ x m
4   getp {x} {m = m} p | yes q = q
```

Definition of a subtraction from a proof of inferiority. Knowing that a is lower than b, we can compute the difference between b and a:

```
5   sub : ∀ {a b} → a ≤ b → ℕ
6   sub (z≤n {n}) = n
7   sub (s≤s p) = sub p
```

Being a member of a map necessarily means being a member of the keys of this map:

```
8   prop← : ∀ {ℓ} {B : Set ℓ} {m : Map {B = B}} {x}
9     → x ∈ₗ m → x ∈ (keys m)
10  prop← (here px) = here px
11  prop← {m = m} (there p) = there (prop← {m = trim m λ {()}} p)
```

A convenient way of proving the decidability of an existential quantifier when we have a list of possible witnesses and the proof that all the other elements cannot be witnesses. This requires the underlying predicate to be decidable and consists in testing all the elements in the list for decidability:

```
12  fromSample : ∀ {a ℓ} {A : Set a} {P : Pred A ℓ} (l : List A) →
13    Decidableₚ P → (∀ {x} → ¬ Anyₗ (x ≡_) l → ¬ P x) → Dec (∃ P)
14  fromSample [] _ q = no λ {(_ , px) → q (λ {()}) px}
15  fromSample (x :: _) decp _ with decp x
16  fromSample (x :: _) _ _ | yes p = yes (x , p)
17  fromSample (x :: l) decp q | no ¬p = fromSample l decp λ ¬avl pv →
18    q (λ {(here refl) → ⊥-elim (¬p pv) ; (there avl) → ¬avl avl}) pv
```

A.2.4 A detailed building of the firing predicates

Extension of _≤_ to ensure the right operand is of the form `just y`:

```
1  _≤₀_ : REL ℕ (Maybe ℕ) _                                    1
2  _≤₀_ x = Any (x ≤_)                                          2
```

Ability of an arc to be fired inside a map of places:

```
3  CanFireArc₀ : REL (String × ℕ × ℕ) Map _                      3
4  CanFireArc₀ (a , -n , _) = (-n ≤₀_) ∘ (get a)                 4
```

Ability of a list of arcs to be fired inside a map of places:

```
5  CanFireArcs₀ : REL Map (List (String × ℕ × ℕ)) _              5
6  CanFireArcs₀ m = All ( _⟨ CanFireArc₀ ⟩ m)                   6
```

Resulting alternate (but equivalent) version of `CanFireTrans`:

```
7  CanFireTrans₀ : REL String Petrinet _                        7
8  CanFireTrans₀ s [ₘ marking - transitions ₜ][ _ ] =            8
9    Any (CanFireArcs₀ marking ∘ Map.content) (get s transitions) 9
```

A.2.5 A detailed building of the decidability predicates

Decidability of $_\leq_0_$, using `dec` which transfers the decidability of a predicate `P` into the decidability of the predicate `Any P`:

```
1  dec≤₀ : Decidable _≤₀_                                       1
2  dec≤₀ x = dec (x ≤?_)                                        2
```

Decidability of the firing of an arc using the number of tokens consumed by the arc and the return value of retrieving the target place in the map:

```
3  deccfa₀ : Decidable CanFireArc₀                              3
4  deccfa₀ (a , -n , _) = (dec≤₀ -n) ∘ (get a)                   4
```

Decidability of the firing of a list of arcs using `all` which transfers the decidability of a predicate `P` into the decidability of the predicate `All P`:

```
5  deccfas₀ : Decidable CanFireArcs₀                            5
6  deccfas₀ m = all λ x → deccfa₀ x m                           6
```

Decidability of the CanFireTrans$_0$ predicate, using dec once more:

```
7  deccft₀ : Decidable CanFireTrans₀                                      7
8  deccft₀ s [ₘ marking - transitions ₜ][ _ ] =                          8
9    dec ((deccfas₀ marking) ∘ Map.content) (get s transitions)          9
```

A.2.6 Decidability of the equality between worksequences

Decidability of the equality between dependences then WorkSequences:

```
1   _≟d_ : Decidable {A = Dependence} _≡_                        1
2   toStart start ≟d toStart start = yes refl                    2
3   toStart start ≟d toStart finish = no λ ()                    3
4   toStart start ≟d toFinish _ = no λ ()                        4
5   toStart finish ≟d toStart start = no λ ()                    5
6   toStart finish ≟d toStart finish = yes refl                  6
7   toStart finish ≟d toFinish _ = no λ ()                       7
8   toFinish start ≟d toStart _ = no λ ()                        8
9   toFinish start ≟d toFinish start = yes refl                  9
10  toFinish start ≟d toFinish finish = no λ ()                  10
11  toFinish finish ≟d toStart _ = no λ ()                       11
12  toFinish finish ≟d toFinish start = no λ ()                  12
13  toFinish finish ≟d toFinish finish = yes refl                13
14  -                                                            14
15  dec≡ws : Decidable {A = WorkSequence} _≡_                    15
16  dec≡ws = ≡-dec _≟_ (≡-dec _≟d_ _≟_)                          16
```

A.2.7 An alternate definition of SimplePDL

An alternate definition of WorkSequences containing the proofs of conformity for its fields. These proofs of conformity refer to a Map of WorkSequences which is passed in parameter of the record:

```
1  record WorkSequence' (m : Map) : Set where                    1
2    field                                                       2
3      predecessor : String                                      3
4      successor : String                                        4
5      dependence : Dependence                                   5
6      .p∈m : predecessor ∈ₗ m                                   6
7      .s∈m : successor ∈ₗ m                                     7
8      .¬≡ : ¬ predecessor ≡ successor                           8
```

Proof that the equality between WorkSequences is decidable, using the decidability of the equality between string sand dependences. Note that, since the properties are declared irrelevant, it is not required to prove their equality which is very convenient in such cases:

```
9   dec≡ws' : ∀ {m} → Decidable {A = WorkSequence' m} _≡_                  9
10  dec≡ws' w1 w2 with predecessor w1 ≟ predecessor w2                       10
11  dec≡ws' w1 w2 | yes p with successor w1 ≟ successor w2                   11
12  dec≡ws' w1 w2 | yes p | yes q with dependence w1 ≟d dependence w2        12
13  dec≡ws' record { predecessor = .p ; successor = .s ; dependence = .d}    13
14    record { predecessor = p ; successor = s ; dependence = d}             14
15      | yes refl | yes refl | yes refl = yes refl                          15
16  dec≡ws' _ _ | yes _ | yes _ | no ¬p = no λ {refl → ¬p refl}              16
17  dec≡ws' _ _ | yes _ | no ¬p = no λ {refl → ¬p refl}                      17
18  dec≡ws' _ _ | no ¬p = no λ {refl → ¬p refl}                              18
```

And finally, a new version of the process model data type, as a couple of a map of WorkDefinitions and a bag of WorkSequences depending on this map. (The `LU.` is required because another instance of `ListUnique` had to be imported for this example:

```
19  SimplePDL' : Set                                                         19
20  SimplePDL' = ∃ λ x → LU.Bag (WorkSequence' x) dec≡ws'                     20
```

A.2.8 Adding a WorkSequence to a process model

A function which transforms a String into a dependence when possible, and accepts four possible String: "s2s", "s2f", "f2f" and "f2f":

```
1   toWsk : String → Maybe Dependence                                        1
2   toWsk s with s ≟ "s2s"                                                   2
3   toWsk s | no _ with s ≟ "s2f"                                            3
4   toWsk s | no _ | no _ with s ≟ "f2s"                                     4
5   toWsk s | no _ | no _ | no _ with s ≟ "f2f"                              5
6   toWsk s | no _ | no _ | no _ | no _ = nothing                            6
7   toWsk s | no _ | no _ | no _ | yes _ = just (toFinish finish)            7
8   toWsk s | no _ | no _ | yes _ = just (toStart finish)                    8
9   toWsk _ | no _ | yes _ = just (toFinish start)                           9
10  toWsk _ | yes _ = just (toStart start)                                   10
```

A function which adds a WorkSequence into a process model. This requires all the properties to safistied and builds the model accordingly, while returning `nothing` otherwise:

```
11  +ws : String → String → String → SimplePDL → Maybe SimplePDL
12  +ws _ kind _ _ with toWsk kind
13  +ws _ _ _ _ | nothing = nothing
14  +ws pred _ succ _ | just _ with pred ≟ succ
15  +ws _ _ _ _ | just _ | yes _ = nothing
16  +ws pred _ _ xpdl | just _ | no _ with dec∈l pred (wds xpdl)
17  +ws _ _ succ xpdl | just _ | no _ | yes _ with dec∈l succ (wds xpdl)
18  +ws pred _ succ xpdl | just x | no _ | yes _ | yes _
19    with dec∈g (pred , x , succ) (wss xpdl)
20  +ws pred _ succ (pdl wds wss c1 c2 c3) |
21    just x | no ¬p | yes q | yes r | no ¬p1 =
22    just (pdl wds (putg pred , x , succ into wss when ¬p1)
23    (λ {(here refl) → q ; (there x1) → c1 x1})
24    (λ {(here refl) → r ; (there x1) → c2 x1})
25    λ {(here refl) → ¬p ; (there x1) → c3 x1})
26  +ws _ _ _ _ | just _ | no _ | yes _ | yes _ | yes _ = nothing
27  +ws _ _ _ _ | just _ | no _ | yes _ | no _ = nothing
28  +ws _ _ _ _ | just _ | no _ | no _ = nothing
```

A.2.9 Decidability of the predicate of compliance

Decidability of the left compliance:

```
1  decatpf : Decidable Complies←
2  decatpf start notStarted = no (λ {(inj1 ()) ; (inj2 ())})
3  decatpf start inProgress = yes (inj1 refl)
4  decatpf start finished = yes (inj2 refl)
5  decatpf finish notStarted = no λ ()
6  decatpf finish inProgress = no λ ()
7  decatpf finish finished = yes refl
```

Decidability of the right compliance:

```
8   decatpt : Decidable Complies→
9   decatpt start notStarted = yes refl
10  decatpt start inProgress = no λ ()
11  decatpt start finished = no λ ()
12  decatpt finish notStarted = no λ ()
13  decatpt finish inProgress = yes refl
14  decatpt finish finished = no λ ()
```

Decidability of the overall compliance:

```
15  deccomp : ∀ {wds} → Decidable (Comp wds)                                15
16  deccomp {wds} _ (toStart a') with decatpf a' wds                         16
17  deccomp start (toStart _) | yes p = yes (cStart p)                       17
18  deccomp finish (toStart _) | yes _ = yes cFinishStart                    18
19  deccomp start (toStart _) | no ¬p = no (λ {(cStart x) → ¬p x})           19
20  deccomp finish (toStart _) | no _ = yes cFinishStart                     20
21  deccomp {wds} _ (toFinish a') with decatpf a' wds                        21
22  deccomp start (toFinish _) | no _ = yes cStartFinish                     22
23  deccomp finish (toFinish _) | no ¬p = no (λ {(cFinish x) → ¬p x})        23
24  deccomp start (toFinish _) | yes _ = yes cStartFinish                    24
25  deccomp finish (toFinish _) | yes p = yes (cFinish p)                    25
```

A.3 On instant refinement

A.3.1 Definition of instants

Definition of the set of instants and their main relations. The two first are directly taken from the strict partial order and the others are derived from them:

```
1   - Underlying set
2   Support : Set
3   Support = Carrier Instant
4   -
5   - Precedence
6   _≺_ : Rel Support lzero
7   _≺_ = _<_ Instant
8   -
9   - Coincidence
10  _≈_ : Rel Support lzero
11  _≈_ = _≈_ Instant
```

```
12  - Causality
13  _≼_ : Rel Support lzero
14  x ≼ y = x ≈ y ⊎ x ≺ y
15  -
16  - Dependance
17  _≭_ : Rel Support lzero
18  x ≭ y = x ≺ y ⊎ y ≺ x
19  -
20  - Independance
21  _∥_ : Rel Support lzero
22  x ∥ y = ¬ x ≭ y × ¬ x ≈ y
```

Extension of these relations to ∃ types by ignoring the proof and applying the relation to the witness. This is done usique a helper, _-⟦_⟧-_. This helper takes two functions of one parameter (its outer parameters) and a function of two parameters (its inner parameter) and returns a function of two parameters which are respectivitely applied the two out functions. It is defined elsewhere in the appendices in Section A.5.1 because it is used at several places in this work:

```
23  _≺'_ : ∀ {u} {P Q : Pred Support u} → REL (∃ P) (∃ Q) _
24  _≺'_ = proj₁ -⟦ _≺_ ⟧- proj₁
25  -
26  _≼'_ : ∀ {u} {P Q : Pred Support u} → REL (∃ P) (∃ Q) _
27  _≼'_ = proj₁ -⟦ _≼_ ⟧- proj₁
28  -
29  _≈'_ : ∀ {u} {P Q : Pred Support u} → REL (∃ P) (∃ Q) _
30  _≈'_ = proj₁ -⟦ _≈_ ⟧- proj₁
31  -
32  _≭'_ : ∀ {u} {P Q : Pred Support u} → REL (∃ P) (∃ Q) _
33  _≭'_ = proj₁ -⟦ _≭_ ⟧- proj₁
34  -
35  _∥'_ : ∀ {u} {P Q : Pred Support u} → REL (∃ P) (∃ Q) _
36  _∥'_ = proj₁ -⟦ _∥_ ⟧- proj₁
37  -
38  _≡'_ : ∀ {u} {P Q : Pred Support u} → REL (∃ P) (∃ Q) _
39  _≡'_ = proj₁ -⟦ _≡_ ⟧- proj₁
```

Shortcuts for properties of the strict partial order between _≈_ and _≺_:

```
40  private                                                   40
41    ispo : IsStrictPartialOrder _≈_ _≺_                     41
42    ispo = isStrictPartialOrder Instant                     42
43  -                                                         43
44  irrefl≈≺ : Irreflexive _≈_ _≺_                            44
45  irrefl≈≺ = irrefl ispo                                    45
46  -                                                         46
47  trans≺ : Transitive _≺_                                   47
48  trans≺ = trans ispo                                       48
49  -                                                         49
50  private                                                   50
51    ≺-resp-≈ : _≺_ Respects₂ _≈_                            51
52    ≺-resp-≈ = ≺-resp-≈ ispo                                52
53  -                                                         53
54  ≺-resp-≈₁ : ∀ {x} → (x ≺_) Respects _≈_                   54
55  ≺-resp-≈₁ = proj₁ ≺-resp-≈                                55
56  -                                                         56
57  ≺-resp-≈₂ : ∀ {x} → (_≺ x) Respects _≈_                   57
58  ≺-resp-≈₂ = proj₂ ≺-resp-≈                                58
59  -                                                         59
60  private                                                   60
61    ie : IsEquivalence _≈_                                  61
62    ie = isEquivalence ispo                                 62
63  -                                                         63
64  refl≈ : Reflexive _≈_                                     64
65  refl≈ = refl ie                                           65
66  -                                                         66
67  sym≈ : Symmetric _≈_                                      67
68  sym≈ = sym ie                                             68
69  -                                                         69
70  trans≈ : Transitive _≈_                                   70
71  trans≈ = trans ie                                         71
```

Symmetry of _≠_ and _∥_:

```
72  sym≠ : Symmetric _≠_                                      72
73  sym≠ (inj₁ x) = inj₂ x                                    73
74  sym≠ (inj₂ y) = inj₁ y                                    74
75  -                                                         75
76  sym∥ : Symmetric _∥_                                      76
77  sym∥ (u , v) = u ∘ sym≠ , v ∘ sym≈                        77
```

Definition of a setoid over a subset of instants, using the coincidence and `∃ P` as a carrier for a given `P`:

```
78  toSetoid : ∀ {u} (P : Pred Support u) → Setoid _ _                                   78
79  toSetoid P = record {                                                                 79
80    Carrier = ∃ P ; _≈_ = _≈'_ ;                                                        80
81    isEquivalence = record { refl = refl≈ ; sym = sym≈ ; trans = trans≈}}               81
```

Algebraic properties around causality:

```
82  ≺→¬≈ : ∀ {x y} → x ≠ y → ¬ x ≈ y                                                      82
83  ≺→¬≈ (inj₁ x≺y) x≈y = irrefl≈≺ x≈y x≺y                                                83
84  ≺→¬≈ (inj₂ y≺x) x≈y = irrefl≈≺ (sym≈ x≈y) y≺x                                         84
85  -                                                                                     85
86  antisym≼ : Antisymmetric _≈_ _≼_                                                      86
87  antisym≼ (inj₁ x) _ = x                                                               87
88  antisym≼ _ (inj₁ x) = sym≈ x                                                          88
89  antisym≼ {α} (inj₂ α≺β) (inj₂ β≺α) = contradiction                                   89
90    (trans≺ α≺β β≺α) (irrefl≈≺ (refl≈ {α}))                                             90
91  -                                                                                     91
92  trans≼ : Transitive _≼_                                                               92
93  trans≼ (inj₁ i≈j) (inj₁ j≈k) = inj₁ (trans≈ i≈j j≈k)                                  93
94  trans≼ (inj₁ i≈j) (inj₂ j≺k) = inj₂ (≺-resp-≈₂ (sym≈ i≈j) j≺k)                        94
95  trans≼ (inj₂ i≺j) (inj₁ j≈k) = inj₂ (≺-resp-≈₁ j≈k i≺j)                               95
96  trans≼ (inj₂ i≺j) (inj₂ j≺k) = inj₂ (trans≺ i≺j j≺k)                                  96
97  -                                                                                     97
98  refl≼ : Reflexive _≼_ ; refl≼ = inj₁ refl≈                                            98
99  -                                                                                     99
100 trans≺≼ : ∀ {α β γ} → α ≺ β → β ≼ γ → α ≺ γ                                           100
101 trans≺≼ α≺β (inj₁ β≈γ) = ≺-resp-≈₁ β≈γ α≺β                                            101
102 trans≺≼ α≺β (inj₂ β≺γ) = trans≺ α≺β β≺γ                                               102
103 -                                                                                     103
104 trans≼≺ : ∀ {α β γ} → α ≼ β → β ≺ γ → α ≺ γ                                           104
105 trans≼≺ (inj₁ α≈β) β≺γ = ≺-resp-≈₂ (sym≈ α≈β) β≺γ                                     105
106 trans≼≺ (inj₂ α≺β) β≺γ = trans≺ α≺β β≺γ                                               106
107 -                                                                                     107
108 ≼→¬≺ : ∀ {α β} → α ≼ β → ¬ β ≺ α                                                      108
109 ≼→¬≺ (inj₁ α≈β) β≺α = irrefl≈≺ (sym≈ α≈β) β≺α                                         109
110 ≼→¬≺ (inj₂ α≺β) β≺α = ≺→¬≈ (inj₂ (trans≺ β≺α α≺β)) refl≈                              110
111 -                                                                                     111
112 ≼→¬≺→≈ : ∀ {α β} → α ≼ β → ¬ α ≺ β → α ≈ β                                            112
113 ≼→¬≺→≈ (inj₁ α≈β) _ = α≈β                                                             113
114 ≼→¬≺→≈ (inj₂ α≺β) ¬α≺β = ⊥-elim (¬α≺β α≺β)                                            114
```

Additional algebraic properties:

```
115  ¬⊎→×¬ : ∀ {a} {A B : Set a} → ¬ (A ⊎ B) → ¬ A × ¬ B
116  ¬⊎→×¬ ¬a⊎b = ¬a⊎b ∘ inj₁ , ¬a⊎b ∘ inj₂
117  -
118  ∥→≈→∥₁ : ∀ {α β γ} → α ∥ β → β ≈ γ → α ∥ γ
119  ∥→≈→∥₁ {α} {β} (¬α≠β , ¬α≈β) β≈γ with ¬⊎→×¬ ¬α≠β
120  ∥→≈→∥₁ (_ , ¬α≈β) β≈γ | ¬α<β , ¬β<α
121    = (λ { (inj₁ α<γ) → ¬α<β (<-resp-≈₁ (sym≈ β≈γ) α<γ) ;
122           (inj₂ γ<α) → ¬β<α (<-resp-≈₂ (sym≈ β≈γ) γ<α)})
123      , (λ α≈γ → ¬α≈β (trans≈ α≈γ (sym≈ β≈γ)))
124  -
125  ∥→≈→∥₂ : ∀ {α β γ} → α ∥ β → α ≈ γ → γ ∥ β
126  ∥→≈→∥₂ α∥β α≈γ = sym∥ (∥→≈→∥₁ (sym∥ α∥β) α≈γ)
127  -
128  <≈≈→< : ∀ {α β γ δ} → α < β → α ≈ γ → β ≈ δ → γ < δ
129  <≈≈→< α<β α≈γ β≈δ = <-resp-≈₂ α≈γ (<-resp-≈₁ β≈δ α<β)
130  -
131  ≼≈≈→≼ : ∀ {α β γ δ} → α ≼ β → α ≈ γ → β ≈ δ → γ ≼ δ
132  ≼≈≈→≼ (inj₁ α≈β) α≈γ β≈δ =
133    inj₁ (trans≈ (sym≈ α≈γ) (trans≈ α≈β β≈δ))
134  ≼≈≈→≼ (inj₂ α<β) α≈γ β≈δ =
135    inj₂ (<≈≈→< α<β α≈γ β≈δ)
136  -
137  ≼-resp-≈₁ : ∀ {x} → (x ≼ _) Respects _≈_
138  ≼-resp-≈₁ x₁≈y (inj₁ x≈x₁) = inj₁ (trans≈ x≈x₁ x₁≈y)
139  ≼-resp-≈₁ x₁≈y (inj₂ x<x₁) = inj₂ (<-resp-≈₁ x₁≈y x<x₁)
140  -
141  ≼-resp-≈₂ : ∀ {x} → (_≼ x) Respects _≈_
142  ≼-resp-≈₂ x₁≈y (inj₁ x₁≈x) = inj₁ (trans≈ (sym≈ x₁≈y) x₁≈x)
143  ≼-resp-≈₂ x₁≈y (inj₂ x₁≼x) = inj₂ (<-resp-≈₂ x₁≈y x₁≼x)
144  -
145  ≡→≈ : ∀ {i j} → i ≡ j → i ≈ j
146  ≡→≈ _≡_.refl = refl≈
```

A.3.2 Equivalence between pairs of relations

Reflexivity of _≈≈_

```
1  refl≈≈ : ∀ {ℓ} → Reflexive (_≈≈_ {ℓ})
2  refl≈≈ = record
3    { <₁→₂ = id ; <₂→₁ = id ; ≈₁→₂ = id ; ≈₂→₁ = id }
```

Symmetry of _≈≈_

```
4  sym≈≈ : ∀ {ℓ} → Symmetric (_≈≈_ {ℓ})                                        4
5  sym≈≈ x≈≈y .≈₁→₂ = ≈₂→₁ x≈≈y                                                5
6  sym≈≈ x≈≈y .≈₂→₁ = ≈₁→₂ x≈≈y                                                6
7  sym≈≈ x≈≈y .≺₁→₂ = ≺₂→₁ x≈≈y                                                7
8  sym≈≈ x≈≈y .≺₂→₁ = ≺₁→₂ x≈≈y                                                8
```

Transitivity of _≈≈_

```
9   trans≈≈ : ∀ {ℓ} → Transitive (_≈≈_ {ℓ})                                    9
10  trans≈≈ i≈≈j j≈≈k .≈₁→₂ = (≈₁→₂ j≈≈k) ∘ ≈₁→₂ i≈≈j                         10
11  trans≈≈ i≈≈j j≈≈k .≈₂→₁ = (≈₂→₁ i≈≈j) ∘ ≈₂→₁ j≈≈k                         11
12  trans≈≈ i≈≈j j≈≈k .≺₁→₂ = (≺₁→₂ j≈≈k) ∘ ≺₁→₂ i≈≈j                         12
13  trans≈≈ i≈≈j j≈≈k .≺₂→₁ = (≺₂→₁ i≈≈j) ∘ ≺₂→₁ j≈≈k                         13
```

A.3.3 Partial ordering between pairs of relations

Antisymmety of _≺≈_ towards _≈≈_

```
1  antisym≺≈ : ∀ {ℓ} → Antisymmetric _≈≈_ (_≺≈_ {ℓ})                          1
2  antisym≺≈ i≺≈j _ .≈₁→₂ = ≈₁→₂ i≺≈j                                         2
3  antisym≺≈ _ j≺≈i .≈₂→₁ = ≈₁→₂ j≺≈i                                         3
4  antisym≺≈ _ j≺≈i .≺₁→₂ = ≺₂→₁ j≺≈i                                         4
5  antisym≺≈ i≺≈j _ .≺₂→₁ = ≺₂→₁ i≺≈j                                         5
```

Instantiation of the partial order structure.

```
6  partialOrder≺≈≈ : ∀ {ℓ} → IsPartialOrder _≈≈_ (_≺≈_ {ℓ})                   6
7  partialOrder≺≈≈ = record {                                                 7
8    isPreorder = preorder≺≈≈ ;                                               8
9    antisym = antisym≺≈ }                                                    9
```

A.3.4 Verification of the example

Definition of the module with the required imports and renames:

```
1  module RefinementExample where                                             1
2  -                                                                          2
3  open import Refinement                                                     3
4  open import Data.Sum renaming (inj₁ to l ; inj₂ to r)                      4
```

Definition of the abstract level coincidence:

```
5   _≈₂_ : _
6   _≈₂_ = _≡_ on _div 5
```

Proof that it is an equivalence:

```
7   eq≈₂ : IsEquivalence _≈₂_
8   eq≈₂ = record { refl = refl ; sym = sym ; trans = trans }
```

Definition of the abstract level precedence:

```
9   _≺₂_ : _
10  _≺₂_ = _<_ on _div 5
```

Proof that it forms a strict partial order with the abstract coincidence:

```
11  irr≺₂≈₂ : Irreflexive _≈₂_ _≺₂_
12  irr≺₂≈₂ {x} {y} with x div 5 | y div 5
13  irr≺₂≈₂ | suc _ | suc _ = <-irrefl
14  --
15  resp≺₂≈₂ : _≺₂_ Respects₂ _≈₂_
16  resp≺₂≈₂ = (λ { {x} {y} {z} → aux₁ {x} {y} {z}}) ,
17    λ { {x} {y} {z} → aux₂ {x} {y} {z}}
18    where
19      aux₁ : ∀ {x y z} → y ≈₂ z → x ≺₂ y → x ≺₂ z
20      aux₁ {x} {y} {z} with x div 5 | y div 5 | z div 5
21      aux₁ | _ | _ | _ = proj₁ (resp₂ _<_)
22      aux₂ : ∀ {x y z} → y ≈₂ z → y ≺₂ x → z ≺₂ x
23      aux₂ {x} {y} {z} with x div 5 | y div 5 | z div 5
24      aux₂ | _ | _ | _ = proj₂ (resp₂ _<_)
25  --
26  ispo≈₂≺₂ : IsStrictPartialOrder _≈₂_ _≺₂_
27  ispo≈₂≺₂ = record {
28    isEquivalence = eq≈₂ ;
29    irrefl = λ { {x} {y} → irr≺₂≈₂ {x} {y}} ;
30    trans = <-trans ;
31    <-resp-≈ = resp≺₂≈₂ }
```

Definition of the concrete level coincidence:

```
32  _≈₁_ : Rel ℕ _
33  a ≈₁ b = (a / 5) ≡ (b / 5) × (a % 5 ≡ b % 5 ⊎ a % 5 + b % 5 ≡ 1)
```

Proof that it is an equivalence:

```
34  sym≈₁ : Symmetric _≈₁_
35  sym≈₁ {a} {b} (fst , snd) with a / 5 | b / 5 | a % 5 | b % 5
36  sym≈₁ {a} {b} (fst , l x) | _ | _ | _ | _ = (sym fst) , l (sym x)
37  sym≈₁ {a} {b} (fst , r y) | _ | _ | w | w₁ =
38    (sym fst) , r (trans (+-comm w₁ w) y)
39  -
40  trans≈₁ : Transitive _≈₁_
41  trans≈₁ {a} {b} {c} _ _ with a / 5 | b / 5 | c / 5 | a % 5 | b % 5 | c % 5
42  trans≈₁ (refl , l refl) (refl , l refl) | _ | _ | _ | _ | _ | _ = refl , l refl
43  trans≈₁ (refl , l refl) (refl , r y) | _ | _ | _ | _ | _ | _ = refl , r y
44  trans≈₁ (refl , r y) (refl , l refl) | _ | _ | _ | _ | _ | _ = refl , r y
45  trans≈₁ (refl , r refl) (refl , r refl) | _ | _ | _ | zero | .1 | .0 = refl , l refl
46  trans≈₁ (refl , r refl) (refl , r refl) | _ | _ | _ | suc zero | .0 | .1 = refl , l refl
47  -
48  eq≈₁ : IsEquivalence _≈₁_
49  eq≈₁ = record {
50    refl = refl , l refl ;
51    sym = λ { {a} {b} → sym≈₁ {a} {b}};
52    trans = λ { {a} {b} {c} → trans≈₁ {a} {b} {c} }}
```

Definition of the concrete level precedence:

```
53  _≺₁_ : Rel ℕ _
54  a ≺₁ b = a / 5 < b / 5 ⊎ (a / 5 ≡ b / 5 × a % 5 < b % 5 × ¬ b % 5 ≡ 1)
```

Proofs of transitivity and irreflexivity at the concrete level:

```
55  trans≺₁ : Transitive _≺₁_
56  trans≺₁ {a} {b} {c} _ _ with a / 5 | b / 5 | c / 5 | a % 5 | b % 5 | c % 5
57  trans≺₁ (l x) (l x₁) | _ | _ | _ | _ | _ | _ = l (<-trans x x₁)
58  trans≺₁ (l x) (r (refl , _)) | _ | _ | _ | _ | _ | _ = l x
59  trans≺₁ (r (refl , _)) (l x) | _ | _ | _ | _ | _ | _ = l x
60  trans≺₁ (r (refl , u , _)) (r (refl , w , x)) | _ | _ | _ | _ | _ | _ =
61    r (refl , (<-trans u w , x))
62  -
63  irr≈₁≺₁ : Irreflexive _≈₁_ _≺₁_
64  irr≈₁≺₁ {a} {b} _ _ with a / 5 | b / 5 | a % 5 | b % 5
65  irr≈₁≺₁ (refl , snd) (l (s≤s x)) | _ | _ | _ | _ = <-irrefl refl x
66  irr≈₁≺₁ (refl , l refl) (r (refl , fst , _)) | _ | _ | _ | _ = <-irrefl refl fst
67  irr≈₁≺₁ (refl , r y₁) (r (refl , _ , snd)) | _ | _ | zero | _ = snd y₁
68  irr≈₁≺₁ (refl , r refl) (r (refl , () , snd)) | _ | _ | suc zero | _
```

Proof of strict partial ordering, starting with right compliance:

```
69  resp≺≈₁ᵣ : _≺₁_ Respectsʳ _≈₁_
70  resp≺≈₁ᵣ {a} {b} {c} _ _ with a / 5 | b / 5 | c / 5 | a % 5 | b % 5 | c % 5
71  resp≺≈₁ᵣ (refl , _) (l x) | _ | _ | _ | _ | _ | _ = l x
72  resp≺≈₁ᵣ (refl , l refl) (r y) | _ | _ | _ | _ | _ | _ = r y
73  resp≺≈₁ᵣ (refl , r refl) (r (refl , _ , snd)) | _ | _ | _ | _ | suc zero | .0 =
74    ⊥-elim (snd refl)
```

Left compliance:

```
75  resp≺≈₁ₗ : _≺₁_ Respectsˡ _≈₁_
76  resp≺≈₁ₗ {a} {b} {c} _ _ with a / 5 | b / 5 | c / 5 | a % 5 | b % 5 | c % 5
77  resp≺≈₁ₗ (refl , _) (l x) | _ | _ | _ | _ | _ | _ = l x
78  resp≺≈₁ₗ (refl , l refl) (r y) | _ | _ | _ | _ | _ | _ = r y
79  resp≺≈₁ₗ (refl , r refl) (r (refl , fst , snd)) | _ | _ | _ | suc _ | zero | .1 =
80    r (refl , ≤∧≢⇒< fst (snd ∘ sym) , snd)
81  resp≺≈₁ₗ (refl , r refl) (r (refl , _ , snd)) | _ | _ | _ | suc _ | suc zero | _ =
82    r (refl , s≤s z≤n , snd)
```

Both compliances, left and right:

```
83  resp≺₁≈₁ : _≺₁_ Respects₂ _≈₁_
84  resp≺₁≈₁ =
85    (λ { {x} {y} {z} → resp≺≈₁ᵣ {x} {y} {z}}) ,
86    λ { {x} {y} {z} → resp≺≈₁ₗ {x} {y} {z}}
```

Instantiation of the strict partial order data type:

```
87  ispo≈₁≺₁ : IsStrictPartialOrder _≈₁_ _≺₁_
88  ispo≈₁≺₁ = record {
89    isEquivalence = eq≈₁ ;
90    irrefl = λ { {x} {y} → irr≈₁≺₁ {x} {y}} ;
91    trans = λ { {x} {y} {z} → trans≺₁ {x} {y} {z}} ;
92    <-resp-≈ = resp≺₁≈₁ }
```

Proof that the concrete partial order refines the abstract partial order, starting with the proof the the third field:

```
93  aux₁ : ∀ {a b} → a ≺₁ b → a ≺₂ b ⊎ a ≈₂ b
94  aux₁ {a} {b} _ with a / 5 | b / 5 | a % 5 | b % 5
95  aux₁ (l x) | _ | _ | _ | _ = l x
96  aux₁ (r (refl , _)) | _ | _ | _ | _ = r refl
```

Proof of the second field:

```
97   aux₂ : ∀ {a b} → a ≈₂ b → a ≈₁ b ⊎ a ≺₁ b ⊎ b ≺₁ a                         97
98   aux₂ {a} {b} _ with a / 5 | b / 5 | a % 5 | b % 5                            98
99   aux₂ refl | _ | _ | zero | zero = l (refl , l refl)                          99
100  aux₂ refl | _ | _ | zero | suc zero = l (refl , r refl)                      100
101  aux₂ refl | _ | _ | zero | suc (suc _) = r (l (r (refl , s≤s z≤n , (λ ())))) 101
102  aux₂ refl | _ | _ | suc zero | zero = l (refl , r refl)                      102
103  aux₂ refl | _ | _ | suc (suc _) | zero = r (r (r (refl , s≤s z≤n , (λ ())))) 103
104  aux₂ refl | _ | _ | suc zero | suc zero = l (refl , l refl)                  104
105  aux₂ refl | _ | _ | suc zero | suc (suc _) =                                 105
106    r (l (r (refl , s≤s (s≤s z≤n) , (λ ()))))                                  106
107  aux₂ refl | _ | _ | suc (suc w) | suc zero =                                 107
108    r (r (r (refl , s≤s (s≤s z≤n) , (λ ()))))                                  108
109  aux₂ refl | _ | _ | suc (suc w) | suc (suc w₁) with <-cmp w w₁                109
110  aux₂ refl | _ | _ | suc (suc _) | suc (suc _) | tri< a _ _ =                 110
111    r (l (r (refl , s≤s (s≤s a) , (λ ()))))                                    111
112  aux₂ refl | _ | _ | suc (suc _) | suc (suc ._) | tri≈ _ refl _ =             112
113    l (refl , l refl)                                                          113
114  aux₂ refl | _ | _ | suc (suc _) | suc (suc _) | tri> _ _ c =                 114
115    r (r (r (refl , s≤s (s≤s c) , (λ ()))))                                    115
```

Instantiation of the refinement record:

```
116  refines : (_≈₁_ , _≺₁_) <≈ (_≈₂_ , _≺₂_)                                    116
117  refines = record {                                                          117
118    ≈₁→₂ = proj₁ ;                                                            118
119    ≈₂→₁ = λ { {a} {b} → aux₂ {a} {b}} ;                                       119
120    ≺₁→₂ = λ { {a} {b} → aux₁ {a} {b}} ;                                       120
121    ≺₂→₁ = l }                                                                121
```

A.4 On CCSL

A.4.1 Image of a function in a setoid

Creation of a setoid containing all the elements that have an antecedent by `f` coupled with the equivalence of the initial setoid:

```
1  imageSetoid : ∀ {a₁ a₂ b₁ b₂} {A : Setoid a₁ a₂} {B : Setoid b₁ b₂}                 1
2    (f : A ⟶ B) → Setoid (a₁ ⊔ b₁ ⊔ b₂) b₂                                             2
3  imageSetoid {B = record {                                                             3
4    Carrier = _ ; _≈_ = _≈_ ;                                                           4
5    isEquivalence = record { refl = r ; sym = s ; trans = t }}}                         5
6    record { _⟨$⟩_ = _⟨$⟩_ ; cong = cong } = record {                                   6
7      Carrier = ∃ λ y → ∃ (y ≈_ ∘ _⟨$⟩_) ;                                              7
8      _≈_ = _≈_ on proj₁ ;                                                              8
9      isEquivalence = record { refl = r ; sym = s ; trans = t } }                       9
```

Creation of a new version of the function `f` that goes to this newly created setoid of the elements within its range:

```
10 imageFunction : ∀ {a₁ a₂ b₁ b₂} {A : Setoid a₁ a₂} {B : Setoid b₁ b₂}               10
11   (f : A ⟶ B) → (A ⟶ imageSetoid f)                                                  11
12 imageFunction {B = B} record { _⟨$⟩_ = f ; cong = cong } =                           12
13   let refl≈ = IsEquivalence.refl (Setoid.isEquivalence B) in                          13
14   record {                                                                            14
15     _⟨$⟩_ = λ x → f x , x , refl≈ ;                                                   15
16     cong = cong }                                                                     16
```

This new version of `f` is surjective by construction. This means that it is bijective when `f` is injective:

```
17 injtobij : ∀ {a₁ a₂ b₁ b₂} {A : Setoid a₁ a₂} {B : Setoid b₁ b₂} {f} →               17
18   Injective {A = A} {B} f → Bijective (imageFunction f)                               18
19 injtobij {B = record { Carrier = Carrier ; _≈_ = _ ;                                 19
20   isEquivalence = record { refl = _ ; sym = s ; trans = t }}} inj = record {20
21   injective = inj ; surjective = record {                                             21
22     from = record {                                                                   22
23       _⟨$⟩_ = proj₁ ∘ proj₂ ;                                                         23
24       cong = λ { {_ , _ , y} {_ , _ , z} → inj ∘ t (s y) ∘ flip t z}} ;              24
25     right-inverse-of = s ∘ proj₂ ∘ proj₂ }}                                           25
```

A.4.2 The binding function of the precedence is bijective

The binding function extended to setoids:

```
1  h′ : (_⟶_ on (toSetoid ∘ Ticks)) c₂ c₁                          1
2  h′ = record { _⟨$⟩_ = h ; cong = congruent}                       2
```

Proof of injectivity over these setoids:

```
3  injective : Injective h′                                          3
4  injective {i} {j} with TTot c₂ i j                                4
5  injective | tri< a _ _ = (contradiction a) ∘                      5
6    (contraposition preserves) ∘ ≼→¬≺ ∘ inj₁ ∘ sym≈                 6
7  injective | tri≈ _ b _ = const (≡→≈ b)                            7
8  injective | tri> _ _ c = (contradiction c) ∘                      8
9    (contraposition preserves) ∘ ≼→¬≺ ∘ inj₁                        9
```

Proof of bijectivity of the restricted function:

```
10 bijective : Bijective (imageFunction h′)                          10
11 bijective = injtobij injective                                    11
```

A.4.3 Compliance between precedence and equality

Respect of the right side of _≺≺_:

```
1  ≺≺-respʳ-∽ (_ , c₃⊑c₂) c₁≺≺c₂ .h = (h c₁≺≺c₂) ∘ proj₁ ∘ c₃⊑c₂                 1
2  ≺≺-respʳ-∽ {_} {c₂} (_ , c₃⊑c₂) c₁≺≺c₂ .congruent =                          2
3    (congruent c₁≺≺c₂) ∘                                                      3
4    (trans≈ ((sym≈ ∘ proj₂ ∘ c₃⊑c₂) _)) ∘                                      4
5    (flip trans≈ ((proj₂ ∘ c₃⊑c₂) _))                                          5
6  ≺≺-respʳ-∽ (_ , c₃⊑c₂) c₁≺≺c₂ .precedes =                                    6
7    (flip ≺-resp-≈₁ (precedes c₁≺≺c₂ _)) ∘ sym≈ ∘ proj₂ ∘ c₃⊑c₂                7
8  ≺≺-respʳ-∽ (_ , c₃⊑c₂) c₁≺≺c₂ .preserves =                                   8
9    (preserves c₁≺≺c₂) ∘                                                      9
10   (≺-resp-≈₂ ((proj₂ ∘ c₃⊑c₂) _)) ∘                                          10
11   (≺-resp-≈₁ ((proj₂ ∘ c₃⊑c₂) _))                                            11
12 ≺≺-respʳ-∽ {_} {c₂} (c₂⊑c₃ , c₃⊑c₂) c₁≺≺c₂ .dense {p = p} u v =              12
13   ((proj₁ ∘ c₂⊑c₃ ∘ proj₁ ∘ (dense c₁≺≺c₂ {p = p} u)) v) ,                  13
14   trans≈ (congruent c₁≺≺c₂ ((sym≈ (trans≈ ((proj₂ ∘ c₂⊑c₃) _)               14
15   ((proj₂ ∘ c₃⊑c₂ ∘ proj₁ ∘ c₂⊑c₃) _)))))                                    15
16   (proj₂ (dense c₁≺≺c₂ u v))                                                16
```

Respect of the left side of $_\!<\!<\!_$:

```
17  <<-respˡ-∽ (c₂⊑c₃ , _) c₂<<c₁ .h =
18    proj₁ ∘ c₂⊑c₃ ∘ h c₂<<c₁
19  <<-respˡ-∽ {_} {_} {c₃} (c₂⊑c₃ , _) c₂<<c₁ .congruent =
20    (flip trans≈ ((proj₂ ∘ c₂⊑c₃) _)) ∘
21    (trans≈ ((sym≈ ∘ proj₂ ∘ c₂⊑c₃) _)) ∘
22    (congruent c₂<<c₁)
23  <<-respˡ-∽ (c₂⊑c₃ , _) c₂<<c₁ .precedes =
24    (≺-resp-≈₂ ((proj₂ ∘ c₂⊑c₃) _)) ∘
25    (precedes c₂<<c₁)
26  <<-respˡ-∽ (c₂⊑c₃ , _) c₂<<c₁ .preserves =
27    ≺-resp-≈₁ ((proj₂ ∘ c₂⊑c₃) _) ∘
28    ≺-resp-≈₂ ((proj₂ ∘ c₂⊑c₃) _) ∘
29    preserves c₂<<c₁
30  <<-respˡ-∽ {c₁} {c₂} {c₃} (c₂⊑c₃ , c₃⊑c₂) c₂<<c₁ .dense {p = p} u v
31    with c₃⊑c₂ p
32  <<-respˡ-∽ {c₁} {c₂} {c₃} (c₂⊑c₃ , c₃⊑c₂) c₂<<c₁ .dense {p = p} u v |
33    q , p≈q with dense c₂<<c₁ {p = q} (≺-resp-≈₁ p≈q (≺-resp-≈₂
34    ((sym≈ ∘ proj₂ ∘ c₂⊑c₃ ∘ h c₂<<c₁) _) u))
35    (≺-resp-≈₁ ((sym≈ ∘ proj₂ ∘ c₂⊑c₃ ∘ h c₂<<c₁) _)
36    (≺-resp-≈₂ p≈q v))
37  <<-respˡ-∽ {c₁} {c₂} {c₃} (c₂⊑c₃ , c₃⊑c₂) c₂<<c₁ .dense {p = p} _ _
38    | q , p≈q | r , hr≈q = r ,
39    (trans≈ ((sym≈ ∘ proj₂ ∘ c₂⊑c₃ ∘ h c₂<<c₁) r)
40    (trans≈ hr≈q (sym≈ p≈q)))
```

A.4.4 A clock can strictly precede itself on integers

Creation of a clock that ticks on every instants of $\mathbb{Z}$:

```
1  always : Clock
2  always = (λ _ → ⊤) ⧖ λ {(i , _) (j , _) → <-cmp i j}
```

Auxiliary proofs for the precedence structures, starting with the `precedes` field:

```
3  x-1<x : ∀ z → - 1ℤ + z < z
4  x-1<x (+_ zero) = -<+
5  x-1<x +[1+ zero ] = +<+ (s≤s z≤n)
6  x-1<x +[1+ ℕ.suc n ] = +<+ (n<1+n (ℕ.suc n))
7  x-1<x (-[1+ zero ] ) = -<- (s≤s z≤n)
8  x-1<x (-[1+ ℕ.suc n ]) = -<- (n<1+n (ℕ.suc n))
```

Proof of the `preserves` field:

```
preserves₀ : ∀ {i j} → i < j → - 1ℤ + i < - 1ℤ + j
preserves₀ {+_ zero} {.(+[1+ _ ])} (+<+ (s≤s m<n)) = -<+
preserves₀ {+[1+ zero ]} {.(+[1+ _ ])} (+<+ (s≤s m<n)) = +<+ m<n
preserves₀ {+[1+ ℕ.suc n ]} {.(+[1+ _ ])} (+<+ (s≤s m<n)) = +<+ m<n
preserves₀ { -[1+_] n} {.(-[1+ _ ])} (-<- n<m) = -<- (s≤s n<m)
preserves₀ { -[1+_] zero} {+_ zero} -<+ = -<- (s≤s z≤n)
preserves₀ { -[1+_] zero} {+[1+ n ]} -<+ = -<+
preserves₀ { -[1+_] (ℕ.suc n)} {+_ zero} -<+ = -<- (s≤s z≤n)
preserves₀ { -[1+_] (ℕ.suc n)} {+[1+ m ]} -<+ = -<+
```

Proof for the `dense` field. This proof actually does not need its premises because in ℤ, for any `p`, there always exists a `k` such as `1 + k ≡ p`:

```
dense₀ : ∀ {p i j} → - 1ℤ + i < p → p < - 1ℤ + j → ∃ \k → - 1ℤ + k ≡ p
dense₀ {+_ zero} {+_ n₁} {+_ n₂} u v = +[1+ zero ] , refl
dense₀ {+[1+ n ]} {+_ n₁} {+_ n₂} u v = +[1+ ℕ.suc n ] , refl
dense₀ { -[1+_] zero} {+_ n₁} {+_ n₂} u v = + zero , refl
dense₀ { -[1+_] (ℕ.suc n)} {+_ n₁} {+_ n₂} u v = -[1+ n ] , refl
dense₀ { -[1+_] zero} {+_ n₁} { -[1+_] n₂} u v = + zero , refl
dense₀ { -[1+_] (ℕ.suc n)} {+_ n₁} { -[1+_] n₂} u v = -[1+ n ] , refl
dense₀ {+_ n} { -[1+_] n₁} {+_ n₂} u v = +[1+ n ] , refl
dense₀ { -[1+_] zero} { -[1+_] n₁} {+_ n₂} u v = + zero , refl
dense₀ { -[1+_] (ℕ.suc n)} { -[1+_] n₁} {+_ n₂} u v = -[1+ n ] , refl
dense₀ { -[1+_] zero} { -[1+_] n₁} { -[1+_] n₂} u v = + zero , refl
dense₀ { -[1+_] (ℕ.suc n)} { -[1+_] n₁} { -[1+_] n₂} u v = -[1+ n ] , refl
```

Proof that the defined clock strictly precedes itself using the previous definitions:

```
all≺≺all : always ≺≺ always
all≺≺all = record
  { h = λ {(x , tt) → (- 1ℤ + x , tt)}
  ; congruent = λ {refl → refl}
  ; precedes = x-1<x ∘ proj₁
  ; preserves = preserves₀
  ; dense = λ { {i , tt} {j , tt} {p , tt} x y →
    case dense₀ {p} {i} {j} x y of
      λ {(z , p) → (z , tt) , p}}}
```

A.4.5 Two clocks can precede each other and not be equal

We start by defining the notion of odd and even for integers.

```
1   data Evenℕ : ℕ → Set where                                              1
2     even0 : Evenℕ 0                                                        2
3     evenss : ∀ {n} → Evenℕ n → Evenℕ (ℕ.suc (ℕ.suc n))                     3
4   -                                                                        4
5   Oddℕ : ℕ → Set                                                           5
6   Oddℕ = ¬_ ∘ Evenℕ                                                        6
7   -                                                                        7
8   data Even : ℤ → Set where                                                8
9     even+ : ∀ {n} → Evenℕ n → Even (+ n)                                   9
10    even-1+ : ∀ {n} → Evenℕ n → Even -[1+ ℕ.suc n ]                        10
11  -                                                                        11
12  Odd : ℤ → Set                                                            12
13  Odd = ¬_ ∘ Even                                                          13
```

Some properties about how even and odd behave when subtracting one:

```
14  prop : ∀ {n} → Evenℕ n → Oddℕ (ℕ.suc n)                                  14
15  prop {.0} even0 = λ ()                                                    15
16  prop {.(ℕ.suc (ℕ.suc _))} (evenss p) (evenss q) = prop p q                16
17  -                                                                         17
18  -1change : ∀ {z} → Even z → Odd (- 1ℤ + z)                               18
19  -1change {+[1+ .(ℕ.suc _) ]} (even+ (evenss x)) (even+ x1) = prop x x1    19
20  -1change { -[1+_] .(ℕ.suc _)} (even-1+ x) (even-1+ x1) = prop x x1        20
21  -                                                                         21
22  1-change : ∀ {z} → Odd z → Even (- 1ℤ + z)                               22
23  1-change {+_ zero} p = ⊥-elim (p (even+ even0))                           23
24  1-change {+[1+ zero ]} p = even+ even0                                    24
25  1-change {+[1+ ℕ.suc zero ]} p = ⊥-elim (p (even+ (evenss even0)))        25
26  1-change {+[1+ ℕ.suc (ℕ.suc n) ]} p                                       26
27    with 1-change {+[1+ n ]} λ {(even+ x) → p (even+ (evenss x))}           27
28  1-change {+[1+ ℕ.suc (ℕ.suc n) ]} p | even+ x = even+ (evenss x)          28
29  1-change { -[1+_] zero} p = even-1+ even0                                 29
30  1-change { -[1+_] (ℕ.suc zero)} p = ⊥-elim (p (even-1+ even0))            30
31  1-change { -[1+_] (ℕ.suc (ℕ.suc n))} p                                    31
32    with 1-change { -[1+ n ]} λ {(even-1+ x) → p (even-1+ (evenss x))}      32
33  1-change { -[1+_] (ℕ.suc (ℕ.suc n))} p | even-1+ x = even-1+ (evenss x)   33
34  -                                                                         34
35  change-1 : ∀ {z} → Odd (- 1ℤ + z) → Even z                               35
36  change-1 {+_ zero} _ = even+ even0                                        36
37  change-1 {+[1+ zero ]} x = ⊥-elim (x (even+ even0))                       37
```

```
38  change-1 {+[1+ ℕ.suc zero ]} x = even+ (evenss even0)                    38
39  change-1 {+[1+ ℕ.suc (ℕ.suc n) ]} x                                      39
40    with change-1 {+[1+ n ]} λ {(even+ y) → x (even+ (evenss y))}           40
41  change-1 {+[1+ ℕ.suc (ℕ.suc n) ]} x | even+ x₁ = even+ (evenss x₁)       41
42  change-1 { -[1+_] zero} x = ⊥-elim (x (even-1+ even0))                   42
43  change-1 { -[1+_] (ℕ.suc zero)} x = even-1+ even0                        43
44  change-1 { -[1+_] (ℕ.suc (ℕ.suc n))} x                                   44
45    with change-1 { -[1+ n ]} λ {(even-1+ y) → x (even-1+ (evenss y))}      45
46  change-1 { -[1+_] (ℕ.suc (ℕ.suc .(ℕ.suc _)))} x                          46
47    | even-1+ x₁ = even-1+ (evenss x₁)                                      47
```

Definition of the even and odd clocks:

```
48  evenClock : Clock                                                       48
49  evenClock = Even ⧖ λ {(x , _) (y , _) → <-cmp x y}                     49
50  -                                                                       50
51  oddClock : Clock                                                        51
52  oddClock = Odd ⧖ λ {(x , _) (y , _) → <-cmp x y}                       52
```

Proof that both clocks precede the other when subtracting one to their instants:

```
53  e<<o : evenClock << oddClock                                            53
54  e<<o = record { h = λ {(z , o) → (- 1ℤ + z , 1-change o)}              54
55    ; congruent = λ {refl → refl}                                         55
56    ; precedes = x-1<x ∘ proj₁                                            56
57    ; preserves = preserves₀                                              57
58    ; dense = λ { {i , oi} {j , oj} {p , op} x y → case dense₀ {p} {i} {j} x y   58
59      of λ {(z , refl) → (z , (flip -1change) op) , refl}}}              59
60  -                                                                       60
61  o<<e : oddClock << evenClock                                            61
62  o<<e = record { h = λ {(z , o) → (- 1ℤ + z , -1change o)}              62
63    ; congruent = λ {refl → refl}                                         63
64    ; precedes = x-1<x ∘ proj₁                                            64
65    ; preserves = preserves₀                                              65
66    ; dense = λ { {i , oi} {j , oj} {p , op} x y → case dense₀ {p} {i} {j} x y   66
67      of λ {(z , refl) → (z , change-1 op) , refl}}}                     67
```

Proof that these clocks are not equal, even though they both precede the other:

```
68  ¬e~o : ¬ evenClock ∽ oddClock                                           68
69  ¬e~o (e⊑o , o⊑e) with e⊑o (0ℤ , even+ even0)                            69
70  ¬e~o (e⊑o , o⊑e) | (.+0 , odd0) , refl = odd0 (even+ even0)             70
```

A.4.6 Antisymmetry of the strict precedence towards the equality

Step 1: Definition of the new clock type:

```
1  record WFClock : Set₁ where
2    field clock : Clock ; wf-≺ : WellFounded (_≺'_ {P = Ticks clock})
```

Step 2: Definition of the extended relations.

```
3  _∽ₒ_ : Rel WFClock _
4  _∽ₒ_ = _∽_ on clock
```

```
5  _≼≼ₒ_ : Rel WFClock _
6  _≼≼ₒ_ = _≼≼_ on clock
```

Step 7: Proofs of lemmas needed for the `step` property:

```
7  i≈hi : ∀ {c₁ c₂} (₁≼₂ : c₁ ≼≼ₒ c₂) (₂≼₁ : c₂ ≼≼ₒ c₁) i →
8    i ≈' ((h ₁≼₂) ∘ (h ₂≼₁)) i → i ≈' h ₂≼₁ i
9  i≈hi ₁≼₂ ₂≼₁ i p with precedes ₂≼₁ i
10 i≈hi ₁≼₂ ₂≼₁ i p | inj₁ x = sym≈ x
11 i≈hi {c₁} {c₂} ₁≼₂ ₂≼₁ i p | inj₂ y =
12   ⊥-elim ((≺→¬≈ (inj₂ (trans≼≺ (precedes ₁≼₂ (h ₂≼₁ i)) y))) p)
13 -
14 auxi≈h₁hi : ∀ {c₁ c₂} (₁≼₂ : c₁ ≼≼ₒ c₂) (₂≼₁ : c₂ ≼≼ₒ c₁) i →
15   ({x : ∃ (Ticks (clock c₁))} → x ≺' i → x ≈' (h ₁≼₂ (h ₂≼₁ x))) →
16   (h ₁≼₂ (h ₂≼₁ i)) ≺' i → i ≈' (h ₁≼₂ (h ₂≼₁ i))
17 auxi≈h₁hi {c₁} {c₂} ₁≼₂ ₂≼₁ i p h₁hi≺i = let h , h₁ = h ₂≼₁ , h ₁≼₂ in
18   let h₁hi≈h₁hh₁hi = p {(h₁ ∘ h) i} h₁hi≺i in
19   let h₁hi≈hh₁hi = i≈hi {c₁} {c₂} ₁≼₂ ₂≼₁ ((h₁ ∘ h) i) h₁hi≈h₁hh₁hi in
20   let hh₁hi≺hi = preserves ₂≼₁ {(h₁ ∘ h) i} {i} h₁hi≺i in
21   let h₁hh₁hi≺h₁hi = preserves ₁≼₂ {(h ∘ h₁ ∘ h) i} {h i} hh₁hi≺hi in
22     ⊥-elim (≼→¬≺ (inj₁ h₁hi≈h₁hh₁hi) h₁hh₁hi≺h₁hi)
```

Step 5: The step property, which ensures the heredity of the recursion:

```
23 step : ∀ {c₁ c₂} (₁≼₂ : c₁ ≼≼ₒ c₂) (₂≼₁ : c₂ ≼≼ₒ c₁) i → (∀ {x} →
24   x ≺' i → x ≈' ((h ₁≼₂) ∘ (h ₂≼₁)) x) → i ≈' ((h ₁≼₂) ∘ (h ₂≼₁)) i
25 step ₁≼₂ ₂≼₁ i p with precedes ₂≼₁ i | precedes ₁≼₂ (h ₂≼₁ i)
26 step ₁≼₂ ₂≼₁ i p | inj₁ x | inj₁ y = sym≈ (trans≈ y x)
27 step {c₁} {c₂} ₁≼₂ ₂≼₁ i p | inj₁ hi≈i | inj₂ h₁hi≺hi =
28   auxi≈h₁hi {c₁} {c₂} ₁≼₂ ₂≼₁ i p (trans≺≼ h₁hi≺hi (inj₁ hi≈i))
29 step {c₁} {c₂} ₁≼₂ ₂≼₁ i p | inj₂ hi≺i | inj₁ h₁hi≈hi =
30   auxi≈h₁hi {c₁} {c₂} ₁≼₂ ₂≼₁ i p (trans≼≺ (inj₁ h₁hi≈hi) hi≺i)
31 step {c₁} {c₂} ₁≼₂ ₂≼₁ i p | inj₂ hi≺i | inj₂ h₁hi≺hi =
32   auxi≈h₁hi {c₁} {c₂} ₁≼₂ ₂≼₁ i p (trans≺ h₁hi≺hi hi≺i)
```

Steps 3 and 6: The predicate which implies the antisymmetry and which is proven by recurring over the ticks of the clock:

```
33  i≈h₁∘hi : ∀ {c₁ c₂} (₁≼₂ : c₁ ≼≼ₒ c₂) (₂≼₁ : c₂ ≼≼ₒ c₁) i →           33
34    i ≈' ((h ₁≼₂) ∘ (h ₂≼₁)) i                                          34
35  i≈h₁∘hi {c₁} {c₂} ₁≼₂ ₂≼₁ i = let open All (wf-≺ c₁) in              35
36    wfRec _ (λ j → j ≈' ((h ₁≼₂) ∘ (h ₂≼₁)) j)                          36
37    (λ i ri → step {c₁} {c₂} ₁≼₂ ₂≼₁ i (ri _)) i                        37
```

Steps 4 and 8: Proof that the previous predicate implies the antisymmetry:

```
38  ≼≼-antisym-∽ : Antisymmetric _∽ₒ_ _≼≼ₒ_                                                38
39  ≼≼-antisym-∽ {c₁} {c₂} ₁≼₂ ₂≼₁ =                                                       39
40    (λ i → h ₂≼₁ i , i≈hi {c₁} {c₂} ₁≼₂ ₂≼₁ i (i≈h₁∘hi {c₁} {c₂} ₁≼₂ ₂≼₁ i)) ,           40
41    λ i → h ₁≼₂ i , i≈hi {c₂} {c₁} ₂≼₁ ₁≼₂ i (i≈h₁∘hi {c₂} {c₁} ₂≼₁ ₁≼₂ i)              41
```

A.4.7 Non empty decidable clocks on natural numbers

The `precedent` function, which gives back an instant preceding a given instant, or a proof this is the initial instant:

```
1   precedent : ∀ {ℓ} {P : Pred ℕ ℓ} → Decidable P → (j : ∃ P) →                 1
2     (∀ {x} → x < proj₁ j → ¬ P x) ⊎ ∃ \(i : ∃ P)                               2
3       → proj₁ i < proj₁ j × (∀ {x} → proj₁ i < x → x < proj₁ j → ¬ P x)       3
4   precedent _ (zero , _) = inj₁ \()                                            4
5   precedent dp (suc j , tj) = aux dp j (suc j) tj (n<1+n _)                    5
6     λ j<x → ⊥-elim ∘ (<-irrefl {suc j} refl) ∘ (<-transʳ j<x) where            6
7       aux : ∀ {ℓ} {P : Pred ℕ ℓ} (dp : Decidable P)                            7
8         (i j : ℕ) (pc : P j) (i<j : i < j) →                                   8
9         (∀ {x} → i < x → x < j → ¬ P x)                                        9
10        → (∀ {x} → x < j → ¬ P x) ⊎                                            10
11        ∃ \(k : ∃ P) → proj₁ k < j ×                                           11
12        (∀ {x} → proj₁ k < x → x < j → ¬ P x)                                  12
13      aux dp i j pc i<j p with dp i                                            13
14      aux dp zero j pc i<j p | no ¬p = inj₁ λ {                                14
15        {zero} x<j tx → ⊥-elim (¬p tx) ;                                       15
16        {suc _} x<j tx → p (s≤s z≤n) x<j tx}                                   16
17      aux dp (suc i) j pc i<j p | no ¬p =                                      17
18        aux dp i j pc (<-trans (n<1+n i) i<j)                                  18
19          λ i<x x<j tx → case m≤n⇒m<n∨m≡n i<x of                              19
20            λ {(inj₁ si<x) → p si<x x<j tx ; (inj₂ refl) → ¬p tx}              20
21      aux dp i j pc i<j p | yes p₁ = inj₂ ((i , p₁) , i<j , p)                 21
```

Proof that the precedence over the ticks is wellfounded. This proof uses the fact that the strict precedence over natural numbers is itself well-founded and also forms a total order with the propositional equality:

```
22  wf-<P : ∀ {ℓ} {P : Pred ℕ ℓ} → Decidable P                         22
23    → WellFounded {A = ∃ P} (_<_ on proj₁)                            23
24  wf-<P {P = P} dec x = aux x (<-wellFounded (proj₁ x))               24
25    where                                                             25
26      aux : ∀ (i : ∃ P) → Acc _<_ (proj₁ i) → Acc (_<_ on proj₁) i    26
27      aux i ai with precedent dec i                                   27
28      aux i ai | inj₁ p = acc (λ j j<i → ⊥-elim (p j<i (proj₂ j)))    28
29      aux i (acc q) | inj₂ (j , j<i , p) with aux j (q _ j<i)         29
30      aux i ai | inj₂ (j , j<i , p) | acc rs =                        30
31        acc λ y y<i → case <-cmp (proj₁ j) (proj₁ y) of λ {           31
32          (tri< a _ _) → ⊥-elim ((p a y<i) (proj₂ y)) ;               32
33          (tri≈ _ refl _) → acc rs ;                                  33
34          (tri> _ _ c) → rs y c}                                      34
```

Proof that there exists an initial tick if the clock ticks at least once. This is done by iterating over the precedent instants, starting from the existing tick of the clock, until a tick that precedes all the others is found. This tick exists because the precedence over the ticks is well-founded so there are no infinite descending chains:

```
35  initial : ∀ {ℓ} {P : Pred ℕ ℓ} → Decidable P                        35
36    → ∃ P → ∃ λ (x : ∃ P) → ∀ (y : ∃ P) → proj₁ x ≤ proj₁ y           36
37  initial {P = P} dec p = aux p (<-wellFounded _)                     37
38    where                                                             38
39      aux : (i : ∃ P) → Acc _<_ (proj₁ i) →                           39
40        ∃ λ (x : ∃ P) → ∀ (y : ∃ P) → proj₁ x ≤ proj₁ y               40
41      aux i (acc p) with precedent dec i                              41
42      aux i (acc p) | inj₁ x = i , λ y →                              42
43        case <-cmp (proj₁ i) (proj₁ y) of λ {                         43
44          (tri< a _ _) → <⇒≤ a ;                                      44
45          (tri≈ _ refl _) → ≤-refl ;                                  45
46          (tri> _ _ c) → ⊥-elim (x c (proj₂ y))}                      46
47      aux i (acc p) | inj₂ (y , z , _) = aux y (p _ z)                47
```

Instantiation of CCSL on natural numbers using the strict precedence:

```
48  open import CCSL                                                    48
49    (record {                                                         49
50      isStrictPartialOrder = <-isStrictPartialOrder })                50
51  open Clock                                                          51
```

Definition of non-empty decidable clocks, after which they can be transformed to either `InitialClock` or `WfClock`:

```
52  record Clock-¬∅-Dec : Set₁ where
53    field
54      clock : Clock
55      non-empty : ∃ (Ticks clock)
56      decidable : Decidable (Ticks clock)
57    -
58    wf-≺ : _
59    wf-≺ = wf-<P decidable
60    -
61    birth : _
62    birth = let (i , p) = initial decidable non-empty in
63      i , swap⊎ ∘ m≤n⇒m<n∨m≡n ∘ p
64    -
65    toWf : WFClock
66    toWf = record { clock = clock ; wf-≺ = wf-≺ }
67    -
68    toIn : InitialClock
69    toIn = record { clock = clock ; first = birth }
```

A.4.8 Lattice from union and intersection

Proof that our lattice specification indeed implies the existence of a lattice:

```
1   specToLattice⋃⋂ {_∧_} {_∨_}
2     record { inter∧ = inter∧ ; union∨ = union∨ } =
3     record {
4       isPartialOrder = isPartialOrder∽⊑ ;
5       supremum = λ c₁ c₂ →
6         subUnionᵣ {c₁ ∨ c₂} {c₂} {c₁} union∨ ,
7         subUnionₗ {c₁ ∨ c₂} {c₂} {c₁} union∨ ,
8         λ c₀ c₁⊑c₀ c₂⊑c₀ →
9           ⊑⊑→⊑⋃ {c₀} {c₁ ∨ c₂} {c₂} {c₁} c₂⊑c₀ c₁⊑c₀ union∨ ;
10      infimum = λ c₁ c₂ →
11        subInterₗ {c₁ ∧ c₂} {c₁} {c₂} inter∧ ,
12        subInterᵣ {c₁ ∧ c₂} {c₁} {c₂} inter∧ ,
13        λ c₀ c₀⊑c₁ c₀⊑c₂ →
14          ⊑⊑→⊑⋂ {c₀} {c₁ ∧ c₂} {c₁} {c₂} c₀⊑c₁ c₀⊑c₂ inter∧ }
```

A.5 On refinement and CCSL

A.5.1 A binding operator to ease refinement proofs

An operator to apply a function to each side of a relation, used in the definition of the instants as well as the refinement applied to CCSL:

```
1  _-⟦_⟧-_ : ∀ {a b c d e}
2    {A : Set a} {B : Set b} {C : Set c} {D : Set d} {E : Set e}
3    → (A → B) → (B → D → E) → (C → D) → (A → C → E)
4  f -⟦ _*_ ⟧- g = (λ x _ → f x) -[ _*_ ]- λ _ z → g z
5  -
6  infix 20 _-⟦_⟧-_
```

A binding operator used the same way a monadic binding would be with the addition of a proof of preservation as the inner parameter:

```
7  _>[_]>_ : ∀ {ℓ} {I : Set} {P P' Q : Pred I ℓ}
8    {R : Rel I ℓ}
9    (v : ∃ λ (j : ∃ P) → (Q ∘ proj₁) j)
10   (p : ∀ {j k} → Q j → R k j → Q k)
11   (h : ∀ (x : ∃ P) → ∃ λ (y : ∃ P') → R (proj₁ y) (proj₁ x)) →
12   ∃ λ (j : ∃ P') → (Q ∘ proj₁) j
13 _>[_]>_ (v , Qv) p h with h v
14 _>[_]>_ (v , Qv) p h | w , Rwv = w , p Qv Rwv
15 -
16 infixl 5 _>[_]>_
```

A.5.2 From unique existence to existence

If a value satisfies `P` and `Q` then it satisfies `P`:

```
1  ∃P×Q→∃P : ∀ {a b c} {A : Set a} {P : Pred A b} {Q : A → Set c}
2    → ∃ (P ∩ Q) → ∃ P
3  ∃P×Q→∃P (v , Pv , _) = v , Pv
```

Unique existence can thus be transformed into simple existence:

```
4  ∃!→∃ : ∀ {a b ℓ} {A : Set a} {P : A → Set b} {_≈_}
5    → ∃! {ℓ = ℓ} _≈_ P → ∃ P
6  ∃!→∃ = ∃P×Q→∃P
```

A.5.3 Embodiment of strict precedence

```
<<₂→<<₁ (p , _) (_ , q₁) prec .h = proj₁ ∘ p ∘ h prec ∘ proj₁ ∘ q₁
<<₂→<<₁ {c₁₁} {_} {c₂₁} (p , q) (_ , q₁) prec .congruent {i} {j} i≈₁j
  with q₁ i | q₁ j ; ... | i₁ , i₁≈₂i | j₁ , j₁≈₂j with p (h prec i₁) | p (h prec j₁)
... | i₃ , i₃≈₂i₂ , _ | j₃ , j₃≈₂j₂ , _ with ≈₁→₂ refines i≈₁j
... | i≈₂j with ₂.trans≈ (₂.trans≈ i₁≈₂i i≈₂j) (₂.sym≈ j₁≈₂j)
... | i₁≈₂j₁ with congruent prec {i₁} {j₁} i₁≈₂j₁
... | i₂≈₂j₂ with ₂.trans≈ i₃≈₂i₂ (₂.trans≈ i₂≈₂j₂ (₂.sym≈ j₃≈₂j₂))
... | i₃≈₂j₃ = ≲₁₋₁→≈₂→₁ {c₁₁} {c₂₁} (p , q) {i₃} {j₃} i₃≈₂j₃
<<₂→<<₁ (p , _) (_ , q₁) prec .precedes i with q₁ i
... | j , j≈₂i with p (h prec j)
... | _ , k≈₂hj , _ with precedes prec j
... | hj≺₂j with ₂.≺≈≈→≺ hj≺₂j (₂.sym≈ k≈₂hj) j≈₂i
... | k≺₂i = ≺₂→₁ refines k≺₂i
<<₂→<<₁ {_} {c₁₂} {_} {c₂₂} _ q _ .preserves {i} {j} i≺₁j
  with ≲₁₋₁→≺₁→₂ {c₁₂} {c₂₂} q {i} {j} i≺₁j
<<₂→<<₁ (p , _) (_ , q₁) prec .preserves {i} {j} _ | i≺₂j with q₁ i | q₁ j
... | i₁ , i₁≈i | j₁ , j₁≈j with p (h prec i₁) | p (h prec j₁)
... | _ , i₃≈₂i₂ , _ | _ , j₃≈₂j₂ , _
  with ₂.≺≈≈→≺ i≺₂j (₂.sym≈ i₁≈i) (₂.sym≈ j₁≈j)
... | i₁≺₂j₁ with preserves prec {i₁} {j₁} i₁≺₂j₁
... | i₂≺₂j₂ with ₂.≺≈≈→≺ i₂≺₂j₂ (₂.sym≈ i₃≈₂i₂) (₂.sym≈ j₃≈₂j₂)
... | i₃≺₂j₃ = ≺₂→₁ refines i₃≺₂j₃
<<₂→<<₁ {c₁₁} {c₁₂} {c₂₁} {c₂₂} (p , q) (p₁ , q₁) prec .dense
  {i₀} {j₀} {k₃} i₃≺₁k₃ k₃≺₁j₃ with q₁ i₀ | q₁ j₀ | q k₃
... | i₁ , i₁≈₂i | j₁ , j₁≈₂j | k₂ , k₂≈₂k₃
  with p (h prec i₁) | p (h prec j₁)
... | i₃ , i₃≈₂i₂ , _ | j₃ , j₃≈₂j₂ , _
  with ≲₁₋₁→≺₁→₂ {c₁₁} {c₂₁} (p , q) {i₃} {k₃} i₃≺₁k₃
... | i₃≺₂k₃ with ≲₁₋₁→≺₁→₂ {c₁₁} {c₂₁} (p , q) {k₃} {j₃} k₃≺₁j₃
... | k₃≺₂j₃ with ₂.≺≈≈→≺ i₃≺₂k₃ i₃≈₂i₂ (₂.sym≈ k₂≈₂k₃)
... | i₂≺₂k₂ with ₂.≺≈≈→≺ k₃≺₂j₃ (₂.sym≈ k₂≈₂k₃) j₃≈₂j₂
... | k₂≺₂j₂ with dense prec {i₁} {j₁} {k₂} i₂≺₂k₂ k₂≺₂j₂
... | k₁ , hk₁≈₂k₂ with p₁ k₁
... | k₀ , k₀≈₂k₁ , _ with ₂.trans≈ (proj₂ (q₁ k₀)) k₀≈₂k₁
... | qk₀≈₂k₁ with congruent prec {proj₁ (q₁ k₀)} {k₁} qk₀≈₂k₁
... | hqk₀≈₂hk₁ with ₂.trans≈ (₂.trans≈ hqk₀≈₂hk₁ hk₁≈₂k₂) k₂≈₂k₃
... | hqk₀≈₂k₃ with ₂.trans≈
  (proj₁ (proj₂ (p (h prec (proj₁ (q₁ k₀)))))) hqk₀≈₂k₃
... | phqk₀≈₂k₃ = k₀ , ≲₁₋₁→≈₂→₁ {c₁₁} {c₂₁} (p , q)
  {proj₁ (p (h prec (proj₁ (q₁ k₀))))} {k₃} phqk₀≈₂k₃
```

A.5.4 Abstraction of precedence

```
1   prec₁→≼≲₂ _ (p , q) (p₁ , q₁) prec .h = proj₁ ∘ q ∘ h prec ∘ proj₁ ∘ p₁
2   prec₁→≼≲₂ {_} {c₁₂} {_} {c₂₂} _ (p , q) (p₁ , q₁) prec .congruent
3     {i₀} {j₀} i₀≈₂j₀ with p₁ i₀ | p₁ j₀
4   ... | i₁ , i₁≈₂i₀ , _ | j₁ , j₁≈₂j₀ , _ with q (h prec i₁) | q (h prec j₁)
5   ... | i₃ , i₃≈₂hi₁ | j₃ , j₂≈₂hj₁
6     with ₂.trans≈ i₁≈₂i₀ (₂.trans≈ i₀≈₂j₀ (₂.sym≈ j₁≈₂j₀))
7   ... | i₁≈₂j₁ with ≲₁₋₁→≈₂→₁ {c₁₂} {c₂₂} (p₁ , q₁) {i₁} {j₁} i₁≈₂j₁
8   ... | i₁≈₁j₁ with ≈₁→₂ refines (congruent prec {i₁} {j₁} i₁≈₁j₁)
9   ... | hi₁≈₂hj₁ = ₂.trans≈ i₃≈₂hi₁ (₂.trans≈ hi₁≈₂hj₁ (₂.sym≈ j₂≈₂hj₁))
10  prec₁→≼≲₂ R⇒≼₁ (p , q) (p₁ , q₁) prec .precedes i₀ with p₁ i₀
11  ... | i₁ , i₁≈₂i₀ , _ with precedes prec i₁
12  ... | hi₁Ri₁ with ≼₁→₂ (R⇒≼₁ hi₁Ri₁)
13  ... | hi₁≼₂i₁ with q (h prec i₁)
14  ... | i₃ , i₃≈₂hi₁ = ₂.trans≼ (inj₁ i₃≈₂hi₁) (₂.trans≼ hi₁≼₂i₁ (inj₁ i₁≈₂i₀))
15  prec₁→≼≲₂ {c₁₁} {_} {c₂₁} _ (p , q) (p₁ , q₁) prec .preserves
16    {i₀} {j₀} i₀≺₂j₀ with p₁ i₀ | p₁ j₀
17  ... | i₁ , i₁≈₂i₀ , _ | j₁ , j₁≈₂j₀ , _ with q (h prec i₁) | q (h prec j₁)
18  ... | i₃ , i₃≈₂hi₁ | j₃ , j₂≈₂hj₁
19    with ₂.≺≈≈→≺ i₀≺₂j₀ (₂.sym≈ i₁≈₂i₀) (₂.sym≈ j₁≈₂j₀)
20  ... | i₁≺₂j₁ with ≺₂→₁ refines i₁≺₂j₁
21  ... | i₁≺₁j₁ with preserves prec {i₁} {j₁} i₁≺₁j₁
22  ... | hi₁≺₁hj₁ with
23    ≲₁₋₁→≺₁→₂ {c₁₁} {c₂₁} (p , q) {h prec i₁} {h prec j₁} hi₁≺₁hj₁
24  ... | hi₁≺₂hj₁ = ₂.≺≈≈→≺ hi₁≺₂hj₁ (₂.sym≈ i₃≈₂hi₁) (₂.sym≈ j₂≈₂hj₁)
25  prec₁→≼≲₂ {_} {c₁₂} {_} {c₂₂} _ (p , q) (p₁ , q₁) prec .dense
26    {i₀} {j₀} {k₃} i₃≺₂k₃ k₃≺₂j₃ with p₁ i₀ | p₁ j₀ | p k₃
27  ... | i₁ , i₁≈₂i₀ , _ | j₁ , j₁≈₂j₀ , _ | k₂ , k₂≈₂k₃ , _
28    with q (h prec i₁) | q (h prec j₁)
29  ... | i₃ , i₃≈₂i₂ | j₃ , j₃≈₂j₂
30    with ≺₂→₁ refines (₂.≺≈≈→≺ i₃≺₂k₃ i₃≈₂i₂ (₂.sym≈ k₂≈₂k₃))
31  ... | i₂≺₁k₂ with ≺₂→₁ refines (₂.≺≈≈→≺ k₃≺₂j₃ (₂.sym≈ k₂≈₂k₃) j₃≈₂j₂)
32  ... | k₂≺₁j₂ with dense prec {i₁} {j₁} {k₂} i₂≺₁k₂ k₂≺₁j₂
33  ... | k₁ , hk₁≈₁k₂ with q₁ k₁
34  ... | k₀ , k₀≈₂k₁ with ₂.trans≈ (proj₁ (proj₂ (p₁ k₀))) k₀≈₂k₁
35  ... | pk₀≈₂k₁
36    with ≲₁₋₁→≈₂→₁ {c₁₂} {c₂₂} (p₁ , q₁) {proj₁ (p₁ k₀)} {k₁} pk₀≈₂k₁
37  ... | pk₀≈₁k₁ with congruent prec {proj₁ (p₁ k₀)} {k₁} pk₀≈₁k₁
38  ... | hpk₀≈₁hk₁ with ₂.trans≈ (₂.trans≈ (≈₁→₂ refines hpk₀≈₁hk₁)
39    (≈₁→₂ refines hk₁≈₁k₂)) k₂≈₂k₃
40  ... | hpk₀≈₂k₃ = k₀ , ₂.trans≈ (proj₂ (q (h prec (proj₁ (p₁ k₀))))) hpk₀≈₂k₃
```