FORMALISATION OF SYSML DESIGN MODELS AND AN ANALYSIS STRATEGY USING REFINEMENT

by

Lucas Albertins de Lima

PhD Thesis

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE/2016

Universidade Federal de Pernambuco

Centro de Informática

Pós-graduação em Ciência da Computação

Lucas Albertins de Lima

FORMALISATION OF SYSML DESIGN MODELS AND AN ANALYSIS STRATEGY USING REFINEMENT

A Ph.D. Thesis presented to the Centro de Informática of Universidade Federal de Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Computer Science.

Advisor: *Juliano Manabu Iyoda*

Co-Advisor: *Augusto Cezar Alves Sampaio*

RECIFE

2016

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

L732f Lima, Lucas Albertins de.
Formalisation of SysML design models and an analysis strategy using refinement / Lucas Albertins de Lima. – 2016.
298 f.: il., fig., tab.

Orientador: Juliano Manabu Iyoda.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2016.

Inclui referências e apêndices.

1. Engenharia de software. 2. Métodos formais. 3. Modelagem de sistemas. 4. Verificação de sistemas. I. Iyoda, Juliano Manabu. (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2016-057

Lucas Albertins de Lima

FORMALISATION OF SYSML DESIGN MODELS AND AN ANALYSIS STRATEGY USING REFINEMENT

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de **Doutor** em Ciência da Computação.

Aprovado em: 03/03/2015.

Prof. Dr. Juliano Manabu Iyoda
Orientador do Trabalho de Tese

To William and Pamela.

Acknowledgements

Primeiramente gostaria de agradecer a Deus por ter me proporcionado a realização deste trabalho e por ter me guiado durante toda esta jornada. Em seguida as pessoas mais importantes da minha vida, minha esposa Pâmela e meu filho William por todo o apoio, compreensão e momentos felizes que me ajudaram a seguir em frente. Também tenho muito a agradecer a meus pais José Nivaldo e Ana Maria por sempre acreditarem que eu posso dar mais um passo adiante e por estarem sempre ao meu lado dando força sempre que preciso. A toda a minha família por sempre me incentivarem desde de que me dou por gente, especialmente ao meu querido avô Urcicino cujo exemplo e perseverança nos encorajou a perseguir nossos sonhos.

Aos meus orientadores, Juliano Iyoda e Augusto Sampaio, por toda compreensão, dedicação e aconselhamentos que tornaram possível a concretização deste trabalho. À minha orientadora durante o doutorado sanduíche, Ana Cavalcanti, cujos direcionamentos foram cruciais para a conclusão da integração das semânticas e as estratégias de análise formal, além de ter me proporcionado uma experiência acadêmica inspiradora da qual tirei lições para o resto da minha vida.

Ao colega Alvaro Miyazawa que contribuiu na definição da semântica integrada dos diferentes diagramas, além de ter definido a semântica formal de máquinas de estado que usamos neste trabalho. Aos colegas André Didier e Márcio Cornélio por terem contribuído na definição da semântica formal de diagrama de atividades.

Ao projeto COMPASS por ter permitido o desenvolvimento desta pesquisa inserida num contexto acadêmico e industrial, além do apoio financeiro através da bolsa de estudos. Ter desenvolvido esta pesquisa com colegas de diferentes universidades e empresas estrangeiras foi extremamente enriquecedor para minha carreira.

Ao CNPq e ao projeto Ciência Sem Fronteiras pelo incentivo financeiro para realização do doutorado sanduiche na Universidade de York na Inglaterra.

Finalmente, ao projeto INES que financiou a participação nas diversas conferências onde publicamos artigos e a todos os colegas do programa de pós-graduação do Centro de Informática da UFPE.

I love deadlines. I love the whooshing noise they make as they go by.

—DOUGLAS ADAMS

Resumo

O aumento da complexidade dos sistemas tem levado a um aumento na dificuldade da atividade de projeto. A abordagem padrão para desenvolvimento, baseada em tentativa e erro, com testes usados em estágios avançados para identificar erros, é custosa e leva a prazos de entrega imprevisíveis. Além disto, para sistemas críticos, para os quais segurança é um conceito chave, Verificação e Validação (V&V) com antecedência é reconhecida como uma abordagem valiosa para promover confiança. Neste contexto, nós identificamos três características importantes e desejáveis de uma técnica de V&V: (i) uma linguagem de modelagem gráfica; (ii) raciocínio formal e rigoroso, e (iii) suporte automático para modelagem e raciocínio.

Nós tratamos estes pontos com uma técnica de refinamento para SysML apoiada por ferramentas. SysML é uma linguagem baseada na UML para o projeto de sistemas. Ela tem se tornado um padrão *de facto* na área. Há uma grande disponibilidade de ferramentas de fornecedores como IBM, Atego, e Sparx Systems. Nosso trabalho se destaca de duas maneiras: ao fornecer uma semântica para refinamento e considerar uma coleção representativa de elementos do perfil UML4SysML (blocos, máquina de estados, atividades, e interações) usados de forma combinada. Nós fornecemos uma estratégia para analisar modelos de projeto especificados em SysML. Isto facilita a descoberta de problemas mais cedo durante o ciclo de vida de desenvolvimento de sistemas, reduzindo tempo e custos de produção.

Neste trabalho nós descrevemos nossa semântica a qual é definida usando uma álgebra de processo rica em estado chamada CML e implementada em uma ferramenta para geração automática de modelos formais. Nós também mostramos como esta semântica pode ser usada para análise baseada em refinamento. Nossos estudos de caso são um protocolo de eleição de líder, o qual é um componente crítico de uma aplicação industrial, e um sinal anão, o qual é um dispositivo para controlar tráfego em linhas férreas. Nossas contribuições são: um conjunto de orientações que fornecem significado para os diferentes elementos de modelagem de SysML usados durante o projeto de sistemas; as semânticas formais individuais para atividades, blocos e interações de SysML; uma semântica integrada que combina estas semânticas com outra definida para máquina de estados; e um arcabouço que usa refinamento para raciocínio de sistemas especificados por coleções de diagramas SysML.

Palavras-chave: Álgebra de processos. CML. CSP. refinamento. automação. SysML. semântica.

Abstract

The increasing complexity of systems has led to increasing difficulty in design. The standard approach to development, based on trial and error, with testing used at later stages to identify errors, is costly and leads to unpredictable delivery times. In addition, for critical systems, for which safety is a major concern, early verification and validation (V&V) is recognised as a valuable approach to promote dependability. In this context, we identify three important and desirable features of a V&V technique: (i) a graphical modelling language; (ii) formal and rigorous reasoning, and (iii) automated support for modelling and reasoning.

We address these points with a refinement technique for SysML supported by tools. SysML is a UML-based language for systems design; it has itself become a *de facto* standard in the area. There is wide availability of tool support from vendors like IBM, Atego, and Sparx Systems. Our work is distinctive in two ways: a semantics for refinement and for a representative collection of elements from the UML4SysML profile (blocks, state machines, activities, and interactions) used in combination. We provide a means to analyse design models specified using SysML. This facilitates the discovery of problems earlier in the system development lifecycle, reducing time and costs of production.

In this work we describe our semantics, which is defined using a state-rich process algebra called CML and implemented in a tool for automatic generation of formal models. We also show how the semantics can be used for refinement-based analysis and development. Our case studies are a leadership-election protocol, a critical component of an industrial application, and a dwarf signal, a device used to control rail traffic. Our contributions are: a set of guidelines that provide meaning to the different modelling elements of SysML used during the design of systems; the individual formal semantics for SysML activities, blocks and interactions; an integrated semantics that combines these semantics with another defined for state machines; and a framework for reasoning using refinement about systems specified by collections of SysML diagrams.

Keywords: Process algebra. CML. CSP. refinement. Automation. SysML. Semantics.

List of Figures

1.1 The process of our approach. 21

2.1 Overview of the relationship between UML 2 and SysML. 24
2.2 SysML diagram taxonomy. 25
2.3 A block definition diagram. 26
2.4 An example of an internal block diagram. 26
2.5 A state machine diagram. 27
2.6 Example of an activity diagram. 29
2.7 Control nodes. 30
2.8 Example of a sequence diagram. 31
2.9 Example of a sequence diagram with *interactionUse*. 32
2.10 Example of a sequence diagram with gates. 32

3.1 An illustration of a dwarf signal. 46
3.2 The proper states of a dwarf signal. 47
3.3 The BDD of a dwarf signal. 47
3.4 The state machine of the **DwarfSignal** block. 48
3.5 Abstract model: block definition diagram 49
3.6 Abstract model: state machine diagram 50
3.7 Violation of guidelines of usage. 51
3.9 Example of an activity diagram. 52

4.1 Communication channels for blocks. 61
4.2 Allowed communication patterns for a port. 63
4.3 Processes of a simple block. 73

5.1 Overview of the model of state machines. 99
5.2 Overview of the representation of activities in CML. 102
5.3 Activity Diagram: Adding an element to the Buffer 103
5.4 Activity Diagram: Removing an element of the Buffer 104
5.5 Activity Diagram: Treat Emergency Call 104
5.6 Activity Diagram: Broadcasting a Call 105

6.1 Semantic representation of sequence diagrams in CML. 120
6.2 Illustration of a lifeline. 124
6.3 Illustration of a synchronous message. 126
6.4 Illustration of an asynchronous message. 128

6.5 Illustration of an alternative combined fragment for a lifeline. 129
6.6 Illustration of a state invariant. 131

7.1 The integration points of the semantics of a single block. 137
7.2 Overview of the communication between the models of multiple blocks. 142

8.1 Artisan Studio: ACS tool bar. 151
8.2 Artisan Studio: Button for loading the CML generator. 151
8.3 Artisan Studio: Configuring the generator DLL. 152
8.4 Artisan Studio: Status of the generation process. 152
8.5 Animation in Symphony: selecting an event. 154
8.6 Animation in Symphony: executing a valid event. 154
8.7 Animation in Symphony: executing an invalid event. 155
8.8 Sequence diagram examples. 157
8.10 Work flow of the analysis of properties. 159
8.11 The ActShine activity. 161
8.12 Sequence diagram examples of the Dwarf Signal system. 162
8.14 Refinement statement with optimisations for FDR. 163
8.15 Example of refinement for inserting a private operation. 166
8.16 Example of refinement for inserting a private attribute. 167
8.17 Example of refinement for decomposing a block (part 1). 168
8.18 Block diagrams of the ManagementDepartment system v1. 169
8.20 Example of refinement for decomposing a block (part 2). 169
8.21 Example of refinement for decomposing a block (part 3). 170
8.22 Final BDD of the ManagementDepartment system. 170
8.23 Final IBD of the ManagementDepartment system. 170

B.1 Procedures for pre-processing the CML model. 239

List of Tables

3.1 Main semantic functions for SysML models. 55
3.2 SysML-CML correspondence 59

4.1 Coverage comparison. 97

5.1 Coverage comparison 117

6.1 Coverage comparison 135

7.1 Summary of related works. 147

8.1 Specifications of the computer used in the evaluation. 164
8.2 Results of the scalability evaluation. 165

B.1 Mapping between CML and CSP functions for manipulating sequences and sets. 250

Contents

1 Introduction **16**

1.1 Structure of the Thesis 21

2 Background **23**

2.1 SysML 23

2.1.1 Block-definition diagram 25

2.1.2 Internal block diagram 26

2.1.3 State-machine diagram 27

2.1.4 Activity diagram 29

2.1.5 Sequence diagram 30

2.1.6 Final Remarks on SysML models 33

2.2 CML 33

2.3 Final Remarks 39

3 Modelling Patterns **40**

3.1 Guidelines of usage 40

3.1.1 Entity definition guidelines. 41

3.1.2 Instance-level guidelines. 43

3.1.3 Action language assumptions. 44

3.1.4 Simplification assumptions. 44

3.2 Examples 46

3.2.1 Dwarf Signal 46

3.2.2 The Leadership Election Problem 49

3.2.2.1 The Abstract SysML model 49

3.2.2.2 The Concrete SysML model 50

3.3 Semantic rules metalanguage 52

3.4 Overview of the semantics of SysML models 54

3.5 Final remarks 59

4 Structural Diagrams **60**

4.1 Overview 60

4.2 Operations and signals 63

4.3 Interfaces 65

4.4 Standard ports 66

4.5 Simple blocks 70

4.5.1 Simple blocks: types 71

4.5.2 Simple blocks: channels 72
4.5.3 Simple blocks: processes 73
4.5.4 Simple blocks: controller process 85
4.6 Composite blocks 89
4.7 Related Work 96

5 Behavioural Diagrams 98
5.1 State Machine Diagram 98
5.2 Activity Diagram 101
5.2.1 Overview 101
5.2.2 Examples 103
5.2.3 Formal semantics 105
5.2.3.1 Block Activity Diagrams 105
5.2.3.2 Single Activity Diagram 110
5.2.3.3 Internal Activity Diagram 110
5.2.3.4 Main Action 114
5.3 Related work 116

6 Sequence Diagram 119
6.1 Overview 119
6.2 Formal semantics 121
6.2.1 Lifeline 123
6.2.2 Messages 125
6.2.3 Combined fragments 129
6.2.4 State invariant 131
6.2.5 Main action 132
6.3 Related work 133

7 Model Integration 136
7.1 Non-hierarchical models 137
7.2 Hierarchical models 141
7.3 Scenarios 143
7.4 Related Work 144

8 Model Analysis and Validation of the semantics 148
8.1 Mechanisation of the Translation Process 149
8.2 Consistency Checking and Animation in Symphony 152
8.3 Analysing SysML models 156
8.3.1 Analysis scalability 162
8.4 Validation of the Semantics 165

8.4.1 Inserting a private operation 166
8.4.2 Inserting a private attribute 167
8.4.3 Decomposition of a block 167
8.5 Related Work 171
8.6 Final Remarks 172

9 Conclusion **174**
9.1 Future Research 177

References **179**

Appendix **186**

A Translation rules **187**
A.1 Rules for Activity 193
A.1.1 Call Behaviour Action Parallelism 193
A.1.2 Start Activity 194
A.1.3 Interruptible Regions 196
A.1.4 Interrupt Activity Manager 197
A.1.5 Token Manager 197
A.1.6 Action Nodes 199
A.1.7 Action Types 201
A.1.8 Action Incoming Edges 201
A.1.9 Action Outgoing Edges 202
A.1.10 Call Operation action 203
A.1.11 Opaque action 205
A.1.12 Send Signal action 206
A.1.13 Value Specification action 207
A.1.14 Accept Event action 207
A.1.15 Call Behaviour action 209
A.1.16 Read Self action 210
A.1.17 Read Structural Feature action 211
A.1.18 Control Nodes 211
A.1.19 Initial Node 212
A.1.20 Flow Final Node 213
A.1.21 Activity Final Node 214
A.1.22 Decision Node 214
A.1.23 Merge Node 216
A.1.24 Fork Node 217
A.1.25 Join Node 217

A.1.26 Object Nodes 219
A.1.27 Input Pin 219
A.1.28 Activity Parameter 220
A.1.29 Output Pin 222
A.1.30 Data Store 223
A.1.31 Channel sets 224
A.2 Rules for Sequence diagram 229
A.2.1 Combined fragments 232
A.2.1.1 PAR 233
A.2.1.2 STRICT 233
A.2.1.3 OPT 233
A.2.1.4 BREAK 234
A.2.1.5 CRITICAL 234
A.2.1.6 LOOP 235
A.2.2 InteractionUse 237

B Pre-processing of the CML models 238
B.1 Procedures 238
B.1.1 Procedures for Animation 238
B.1.2 Procedures for Model Checking 241

C Semantic Validation Models 253
C.1 CSP model of the refinement for inserting a private operation 253
C.2 CSP model of the refinement for inserting a private attribute 258
C.3 CSP model of the refinement for the decomposition of a block 263

1 Introduction

One of the most critical aspects in system development consists in the difficulty to assess specification compliant products. Among the factors that influence this issue we can cite the increasing complexity of engineered systems, the effectiveness of the applied methods and budget constraints (DEBBABI et al., 2010). The increasing size and complexity of systems have led to a great difficulty to their description and specification. Features like concurrency and parallelism demand notations and techniques for reasoning about system properties. In a broader context, Systems Engineering is related to the design of whole systems through an iterative process that leads to the development and operation of a system. Overall, system engineering is an interdisciplinary approach to the development of systems (BOARD, 2006). In this case, systems are not only software-intensive but can also involve physical components.

In particular, verification and validation (V&V) tasks must evolve accordingly in order to tackle the increasing size and complexity of systems. The current methodologies based mainly on testing are not effective enough to critical systems due to the impossibility of assuring correctness. In addition, the detection of failures in the early stages of development produces several benefits and a higher return on investment because the maintenance time, effort and costs are decreased. According to Boehm (BOEHM, 1981), fixing a defect after delivery can be considerably more expensive than fixing it during the requirements and the design phases. Hence, in order to reduce risks and costs in systems engineering, activities related to V&V should be performed as early as possible, thus, increasing the quality and reliability of systems.

In some categories of systems, like critical real-time systems where safety is a major concern, early V&V is essential to support their expensive and long-term development process. These activities play a crucial role because of the danger of losing human lives due to system failures. In this context, besides testing, other approaches using formal techniques should be applied to increase the level of reliability on such systems. However, few verification approaches are supported by formal foundations.

Some desired characteristics in the V&V approach are listed below:

- Provide automated mechanisms. Automation optimises the V&V process and avoids the introduction of errors by human-related activities.

- Use formal and rigorous reasoning to reduce errors due to ambiguity.
- Use a graphical modelling language to ease the understanding of the system, hiding an underlying formal model.

Regarding the verification of systems and software, two formal approaches have been proposed to tackle this challenge, theorem proving (GOUBAULT-LARRECQ; MACKIE, 1997) and model checking (CLARKE et al., 1999). Theorem proving deals with first-order or higher-order logic, which allows the verification of infinite state systems. On the other hand, model checking works with decidable logics and is limited to deal with finite state systems. Hence, the effectiveness of model checking is reduced when the system exhibits an enormous (or infinite) number of states. This limitation is well know as the *exponential state explosion problem* (CLARKE et al., 2012). Applying model checking to a system with these characteristics may consume all memory resource of a computer and return no response at all.

Nevertheless, even with this limitation, model checking has proven to be a successful approach for verifying the behaviour of software and hardware applications (MILLER et al., 2010). The reason for this resides in the fact that this technique is fully automated. Model checkers traverse all possible states of a system specification in order to evaluate properties in an automated manner. If a property is not satisfied, a counterexample is returned, which indicates a possible flaw in the system. Such flaw can be corrected and the property analysed again, allowing an iterative process to validate systems.

According to Clarke (CLARKE et al., 1999), model checking means the automatic verification of satisfiability of the relation $M \models p$. In other words, does the model M, which is described as a transition system, satisfy the property p, which is described as a logical formula? Roscoe (ROSCOE, 1997, 1998) proposes a variation of this statement where the refinement $S_p \sqsubseteq S_m$, given that S_p and S_m are CSP (HOARE, 1985) specifications and S_p is known to exhibit the property p, provides an equivalent response to $M \models p$. However, as both specifications are CSP processes, what we are checking in fact is if the process S_p, which is defined in a lower level of abstraction than S_m, has all the behaviours of S_m. This approach is called *refinement model checking*. The model checker FDR (Failures-Divergences Refinement) (GIBSON-ROBINSON et al., 2014) and PAT (SUN et al., 2008), for instance, allow the verification of such refinements by transforming a CSP specification into labelled transition system (LTS) with the purpose of refinement checking.

Although this approach has been applied in industry, there is an issue regarding the notations used by these tools. Usually they require specifications to be written using formal notation, which is usually based on a text format and is difficult to understand for users not familiar with formal languages. Therefore, making system engineers adopt a formal notation in their daily tasks becomes a challenge. System practitioners usually work with graphical notations because they are easy to understand and communicate to other stakeholders. Among this category of languages, UML (Unified Modelling Language) (OMG, 2011) has become an established standard in both academia and industry for modelling software.

Following the success of UML, a profile for systems engineering, called SysML, has been defined (OMG, 2010a) and has become a *de facto* standard in the area. There is wide availability of basic literature (HOLT; PERRY, 2008; FRIEDENTHAL et al., 2011), and tools from vendors like IBM (IBM, 2013), Atego (ATEGO, 2013), and Sparx Systems (SYSTEM, 2013) for SysML.

Like UML, SysML has an informal semantics, with several points left loose. Flexibility of usage for capturing requirements, and for model development and evolution, is prioritised over a well-defined semantics. In systems engineering, however, there are areas of application for which rigour is essential, and the danger of misunderstanding models due to different interpretations of their meaning is not acceptable. Moreover, the use of models in automated techniques for analysis and verification requires the definition of a semantics of the diagrammatic notation in terms of desired formal syntax.

In the past years a considerable effort has been spent to fill this gap between graphical modelling languages and formal languages. This can be achieved by transformations between models, where a source model defined using a graphical language can be transformed into a target model related to a formal specification. The advantage of this approach is to reuse the tool support of the formal notation. Usually, the approaches differ in the modelling language chosen, the formal domain to represent its semantics, the amount of constructs that can be translated and the level of automation provided.

The COMPASS project [1] was the pioneer in proposing model-based methods and tools for the development and analysis of System of Systems (SoS), which is a category of systems where independent constituent systems cooperate in order to achieve a common goal (JAMSHIDI; JAMSHIDI, 2009). It was composed by research groups of five universities (Newcastle University, Aarhus University, University of York, Bremen University and Universidade Federal de Pernambuco) and three companies (Atego, Bang & Olufsen and Insiel) and was funded by the seventh framework programme (FP7) of the European Union. These groups collaborated to advance the state of the art for the development and maintenance of SoS. Among its achievements we can cite the COMPASS Modelling Language (CML), which is a formal language based on a state-rich process algebra, a tool environment for creating and analysing CML specifications, tools for deriving CML specifications from SysML models, and a process for analysing SysML models. Our research was part of the COMPASS project. We have worked in the definition of the formal semantics of SysML diagrams to allow its automatic derivation from SysML tools and we have proposed approaches for analysing SysML models.

Our work enables the use of refinement-based analysis and verification to reason about SysML models. Many have pursued formalisation of both SysML and UML: there are formalisations of individual diagrams, a few examples of which are (GRAVES; BIJAN, 2011a; RAMOS et al., 2005; STORRLE, 2004a; ABDELHALIM et al., 2010), and of models that combine

[1] http://www.compass-research.eu/

diagrams (DAVIES; CRICHTON, 2003). Typically, the approaches that tackle collections of diagrams deal with very constrained subsets of the notation of the individual diagrams. We focus on diagrams that facilitate design, namely the block definition, internal block, state machine, activity, and sequence diagrams. Our work is distinctive both in its coverage of the available notation to write each of these diagrams, and in its integrated approach, where the semantics of all diagrams are integrated into a single formal model. We have chosen not to extend previous work for several motives: the lack of integration between the SysML model elements we chose, the semantic domain were considerable distinct from CML, and, even the works that have a close relationship with CML, like the ones for CSP, do not define their semantics following a compositional approach like we do.

We define a single formal model that captures data and behavioural aspects of a SysML model that gives an overall picture of a system using a collection of diagrams. Our semantic domain (formal modelling language) is CML (WOODCOCK et al., 2012a), a state-rich process algebra for refinement. We define a semantics for SysML models via mapping into CML.

Our goal is to support reasoning at the level of the diagrammatic notation by using an underlying model described in CML together with formal method techniques and tooling. Accordingly, we describe here a semantics that can be used for automatic generation of CML models from diagrams. It takes the form of a function that maps SysML to CML and is defined by transformation rules. It is implemented in a model-generation tool based on Atego's Artisan Studio SysML tool (ATEGO, 2013).

To enable the construction of meaningful CML models, we define usage guidelines for SysML. They are important to guarantee a minimal level of concreteness of the SysML model in order to be able to reason about it. Basically, block diagrams are used to define the system and its components, internal block diagrams to define the relationship between them, each operation is defined in a state machine or in an activity diagram, but not both, and sequence diagrams define scenarios of the system. Our guidelines ensure that we can produce useful CML models that are very comprehensive and impose few restrictions to the user. We illustrate these guidelines and the use of refinement using two industrial case studies: the leadership election protocol, which is used in an industrial multimedia system of systems, and the dwarf signal, which is a hardware device used to control traffic of trains in railways.

The CML semantics for block, internal block, activity, and sequence diagrams have been presented in (MIYAZAWA et al., 2013; LIMA et al., 2013, 2014) and they are direct results of our work. Another contribution is a revised and integrated version of these semantics (LIMA et al., 2015) including the semantics of state machine diagrams, which was defined by Miyazawa (MIYAZAWA et al., 2013) and is not a contribution of this thesis but is nevertheless explained here as it is part of the integrated modelling approach proposed. The definition of these semantics in CML enables their use to analyse properties of the individual diagrams, like refinement of models, for instance, assuring that the actual version of a state machine is a refinement of an earlier version, and absence of deadlock, livelock and non-determinism in these models. In addition, we propose an analysis methodology considering the integrated model where traces and general properties of the system can be verified.

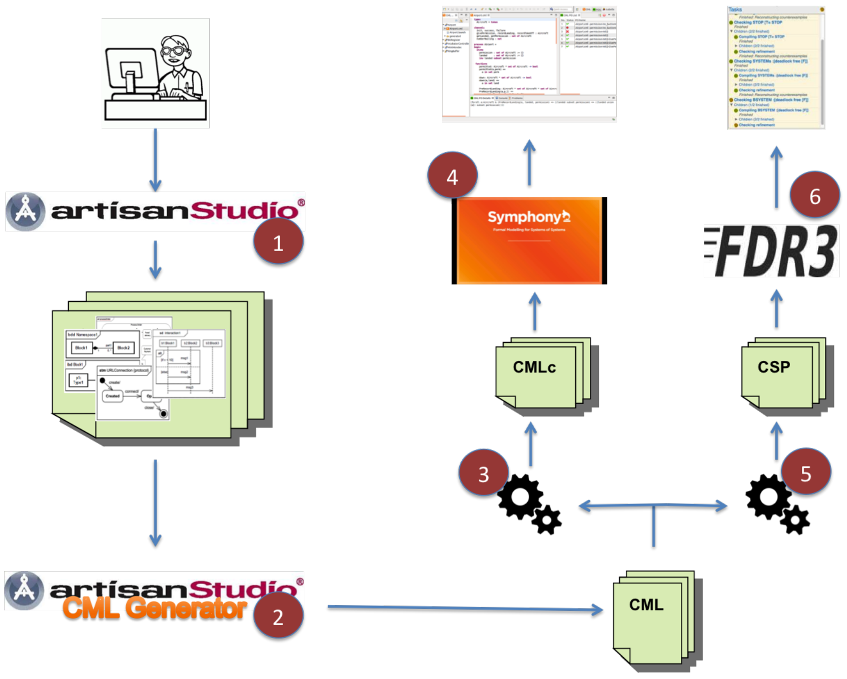
Using our CML semantics, we can use refinement to analyse properties of diagrammatic models. We propose an approach where a system model can be built using block definition and internal block diagrams to represent the structural part of the system, and state machines and activity diagrams to model the behavioural aspects. Valid scenarios, including desired properties of the system can be modelled using sequence diagrams, which we call the validation model. Finally, we can check if the system model behaves according to the validation model using refinement checking.

Figure 1.1 illustrates the process of the verification activities of our approach. In step one, system engineers build SysML models using the Artisan Studio tool. Step 2 corresponds to the automatic transformation of the SysML models into CML models. The generated CML models follow the definitions of our semantics. The translation was implemented by Atego and is not part of this work. The resultant CML models can be either animated or analysed through refinement checking. Step 3 shows the transformations of the CML model into a concrete version able to be animated. This step uses a tool implemented in the scope of this thesis. This tool outputs a new version of the CML model that can be used in the Symphony tool (COLEMAN et al., 2012) for animation in Step 4. For refinement checking analysis, we need to transform our CML models into CSP specifications in Step 5, which are analysed using the FDR model checker in step 6. Despite being a overhead transforming CML to CSP, this is minimised due to the fact that CSP is one of the baseline languages of CML, and they have similar underlying semantics both based on the Unifying Theories of Programming (HOARE; HE, 1998).

In summary, our main contributions are:

- Guidelines of usage for construction of meaningful SysML models (by meaningful, we mean models that can be translated to a formal model in CML);
- State-rich process algebraic semantics for the following SysML diagrams:
  - sequence diagram;
  - activity diagram;
  - block definition diagram;
  - internal block diagram;
- An integrated semantics that combines activity, block definition, internal block, state machine and sequence diagrams;
- An analysis approach for verifying scenarios and properties of SysML models;
- Mechanisation of the process used to transform CML models to allow their animation in the Symphony tool;

Figure 1.1: The process of our approach.



Source: Author's ownership.

- Guidelines for translating CML models into CSP;
- Development of two industrial case studies.

1.1 Structure of the Thesis

Chapter 2 describes the two baseline languages used in the proposed approach. First, the graphical modelling language SysML is described in terms of its diagrams and its relationships with its parental language UML. We also discuss the informal semantics of some constructs presented along the document. It is important to understand the SysML language because models of this language are the inputs to our strategy. Next, we describe the COMPASS Modelling Language (CML), the semantic domain used to define SysML models formally. CML is a formal language that allows the specification of models using state-rich elements and a process algebra. As the SysML constructs are mapped to CML specifications, it is important to understand the syntax and semantics of the constructs used in the CML language as well.

Chapter 3 presents how SysML models must be constructed in order to provide meaningful models, which can be represented in CML according to our semantics. The following chapters detail the isolated semantics of each one of the diagrams of SysML addressed in the thesis.

Chapter 4 presents the CML semantics provided for block diagrams (block definition and internal block diagrams), which covers the structural part of SysML models. Chapter 5 describes the CML semantics for SysML state machine and activity diagrams, both covering the behavioural part of SysML models. Chapter 6 presents the CML semantics for SysML sequence diagrams, which are used to build scenarios of the system that are used to validate the model (as described further in Chapter 8). Although the state machines semantics is not a contribution of this thesis, it is important to understand its semantics when we discuss in Chapter 7 the integrated semantics of all diagrams. We present the related work in each chapter in order to make a clearer relation with our contributions.

Chapter 8 presents some applications of our integrated semantics regarding analysis of SysML models. It describes the mechanisation of the CML models generation from SysML models using the Atego's Artisan Studio tool. Then, it describes how models can be animated, which can be used to understand the communication between the entities of the SysML model, and, as another contribution of this thesis, a strategy to analyse systems modelled in SysML based on scenarios specified in sequence diagrams. This strategy uses the refinement capability of the generated models to verify if the traces of the sequence diagrams are valid in the traces of the model resulted from the combination of the other diagrams. This chapter also provides some validations of our semantics through the exercise of refinements at the SysML level.

Finally, in Chapter 9, we consider what are the benefits of using an integrated semantics for SysML diagrams and the practical advantages and limitations of using our semantics in formal-based tools. We also present what still needs to be done as future research.

2 Background

In this chapter we describe the modelling languages used in the context of this work. Section 2.1 describes SysML, which is the diagrammatic language used as baseline of our approach, while Section 2.2 describes the semantic domain used as the formal semantics of SysML, which is described in terms of the state-rich and process algebraic language, CML.

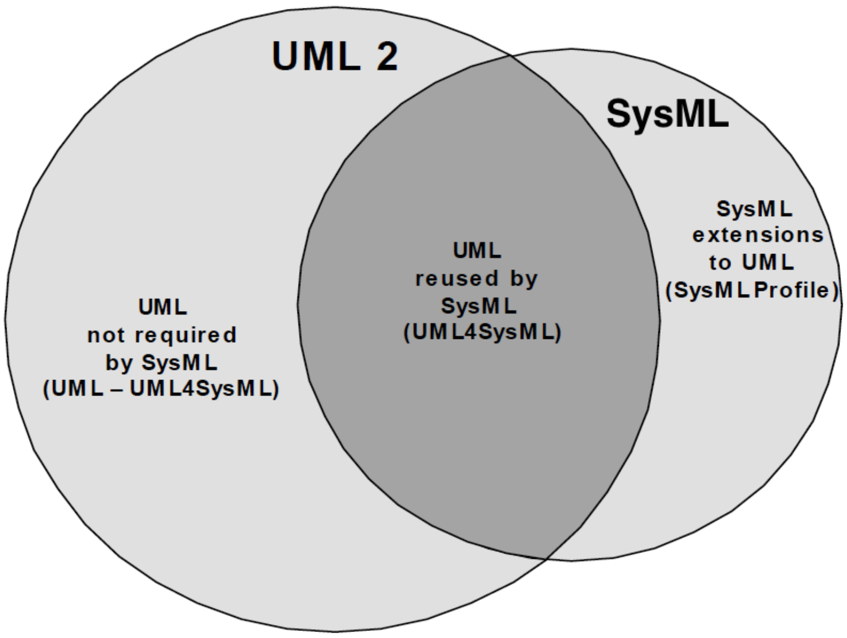
2.1 SysML

SysML, which is a standard of the Object Management Group (OMG, 2012), is a modelling notation for systems engineering, defined as a UML 2.0 profile. SysML allows the representation of systems, hardware, software, information and processes, with the objective of providing dedicated support for system-level modelling, verification and validation. Like UML, SysML provides a number of diagrams to support the description of different aspects of a system. SysML has the following prominent distinctive features that are not present in UML:

- The “classical” software-centric focus present in UML, through class diagrams and composite structure diagrams, has been moved to the system-level in SysML by the introduction of block definition diagrams and internal block diagrams (OMG, 2012, Chapter 8).
- The general UML notion of constraints has been strengthened in SysML through the introduction of constraint blocks and parametric diagrams (OMG, 2012, Chapter 10).
- A notation for requirements engineering and tracing from model elements to requirements (OMG, 2012, Chapter 16).

The Venn diagram shown in Figure 2.1 displays how the two languages, UML2 and SysML, relate to each other. The intersection between the two circles represents the constructs from UML reused by SysML, which is called the UML4SysML subset. The part of the SysML circle that is not part of the UML 2 circle indicates the new modelling constructs of SysML that are not present in UML, or that replace UML constructs. There is a part of UML that is not required in SysML, which is marked "UML not required by SysML".

Figure 2.1: Overview of the relationship between UML 2 and SysML.


UML 2
SysML
UML not required by SysML (UML – UML4SysML)
UML reused by SysML (UML4SysML)
SysML extensions to UML (SysML Profile)

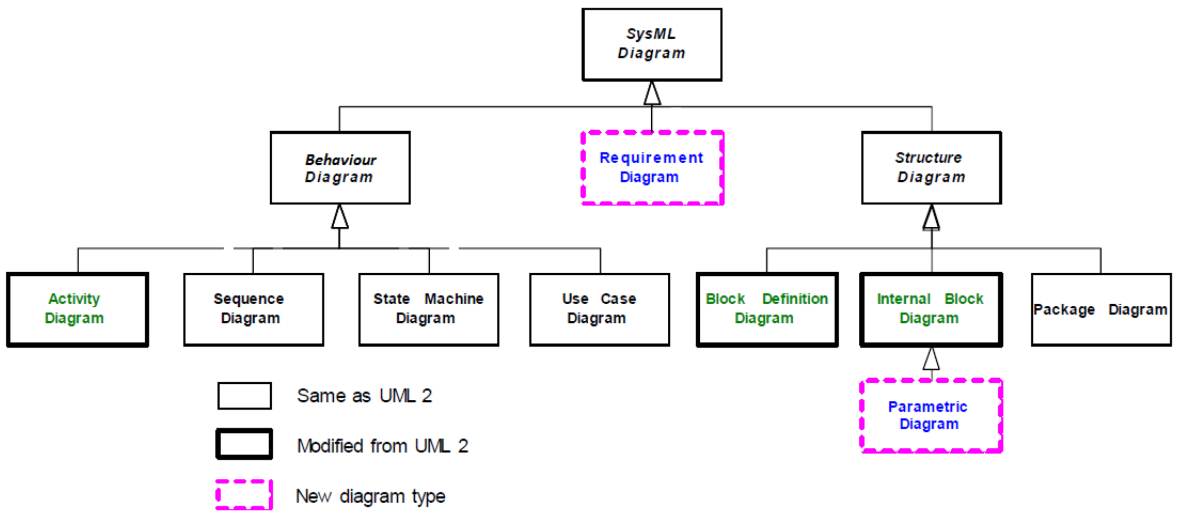
Source: (OMG, 2012)

Figure 2.2 shows a representation of the organisation of the SysML diagrams. Basically, SysML provides nine types of diagrams to model systems. Some of them are completely reused from UML, like sequence diagram, state machine diagram, use case diagram and package diagram. Some others were modified to be consistent with SysML extensions. Block definition diagram and internal block diagram are similar to UML class diagram and composite structure diagram, respectively. UML activity diagram has also been extended. There are two new types of diagrams, requirements diagram and parametric diagram. A requirement diagram provides constructs for text-based requirements, and the relationship between requirements and other model elements that satisfy or verify them. Parametric diagram describes constraints among the properties associated with blocks. Its purpose is to integrate behaviour and structure models with engineering analysis models such as performance and reliability.

The details on the informal syntax and semantics of the diagrams and their constructs is available in (OMG, 2012). Additionally, other sources of literature (HOLT; PERRY, 2008; FRIEDENTHAL et al., 2011) describe the language with the support of real word examples focusing on architectural modelling.

This thesis does not provide semantics for all SysML diagrams. We cover block definition diagram, internal block diagram, activity diagram and sequence diagram. State machine diagram is discussed here as part of the integrated approach but its semantics has been provided separately by partners of the COMPASS project (MIYAZAWA et al., 2013). However, the chosen subset provides means to describe both structure and behaviour of a system. Despite having nine types of diagrams, SysML does not require all diagrams to be used during modelling tasks. It depends

Figure 2.2: SysML diagram taxonomy.



Source: (OMG, 2012).

on what needs to be represented. System designers may only be interested in modelling the behavioural part of a system, or just the structural part, or both, and for that they may choose the diagrams to be used according to their expertise or tools that are available for use.

In this section, we shall briefly describe the parts of SysML structural and behavioural diagrams that are relevant in the context of our semantics of SysML in terms of CML.

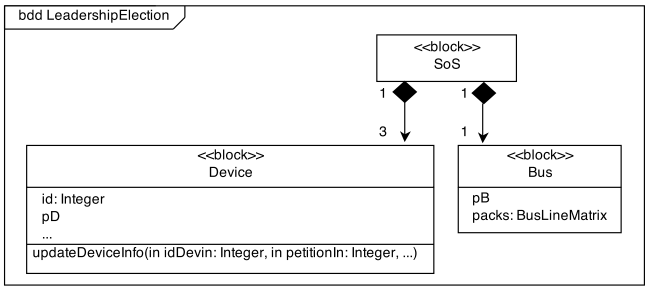
2.1.1 Block-definition diagram

A block-definition diagram depicts blocks and their relationships. A block can represent any abstract or real entity: a piece of hardware, a software, or a physical object, for instance. The whole system is also represented by a block. Figure 2.3 shows a diagram of our case study: a distributed system of systems (devices) that automatically elects a leader among them. A possible scenario of this case study is a house with a television, a cell phone and a home stereo that must provide a synchronised experience to the costumer. However, to achieve that they must elect a leader between the three devices that manage the data that allows the synchronism between them. In this diagram, there are three blocks: SoS represents the complete system (of systems), Device, three devices, and Bus, one bus.

Blocks can have attributes and operations. Attributes are properties of a block. For instance, in Figure 2.3, id is an integer attribute of the block Device, and packs is an attribute of Bus. The attributes pD and pB are ports, which are used for connecting blocks and are further discussed in the next section.

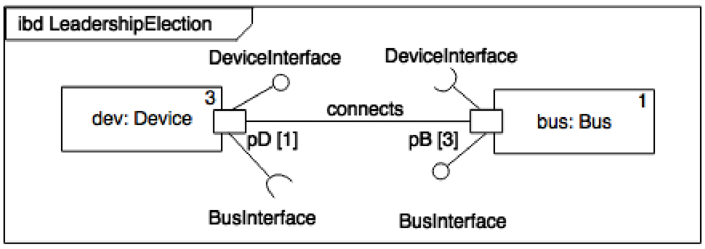
An operation captures functional behaviour provided by the block. For instance, Device is able to update its state via the operation updateDeviceInfo(). Operations are typically triggered by synchronous requests. A signal, on the other hand, does not have a specific behaviour, but may

Figure 2.3: A block definition diagram.



Source: Author's ownership.

Figure 2.4: An example of an internal block diagram.



Source: Author's ownership.

trigger behaviours in state machines and activities. It is used for asynchronous communication between blocks (or with the environment).

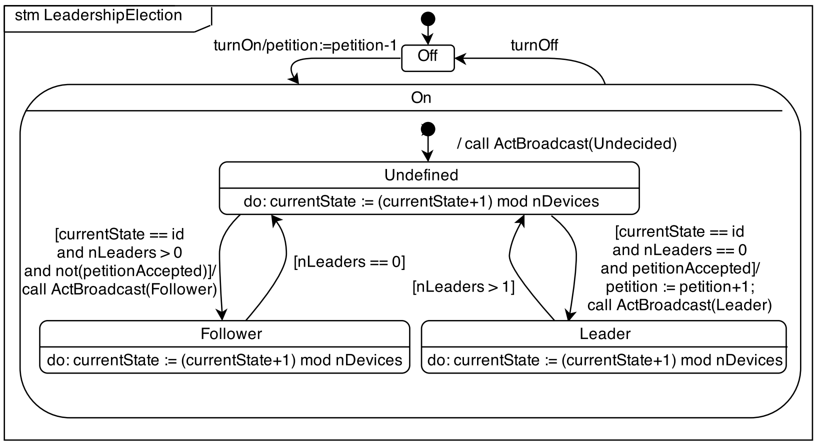
A block can be related to another by an association, which can be over different multiplicities and indicates a potential relationship between instances of the blocks. Blocks can also be related by composition, indicated by an arrow with a filled diamond at one end. A composition establishes a whole-part relationship: the block at the diamond end is the whole composite block and those at the other end are the parts. The main feature of a composition is that a part, that is, an instance owned by the composition, may belong to at most one block. In our example, SoS is a composite block; its parts are three instances of a Device and one instance of a Bus.

2.1.2 Internal block diagram

Internal block diagrams are similar to block definition diagrams, but typically show the internal connections between parts of a block. Figure 2.4 shows the parts of the leadership election protocol.

In this example, three instances of a Device are connected to one instance of a Bus. This is defined via the connector between the ports pD and pB, represented by a solid line. In fact, each connection between each port should be explicit represented in the model, however, we have grouped all connections in one solid line for simplification. A connector links two or more instances of a block either directly or via ports, like in the example, to allow them to communicate. This is in contrast with an association, which specifies that there can be links

Figure 2.5: A state machine diagram.


stm LeadershipElection
turnOn/petition:=petition-1
Off
turnOff
On
/ call ActBroadcast(Undecided)
Undefined
do: currentState := (currentState+1) mod nDevices
[currentState == id and nLeaders > 0 and not(petitionAccepted)]/ call ActBroadcast(Follower)
[nLeaders == 0]
[nLeaders > 1]
[currentState == id and nLeaders == 0 and petitionAccepted]/ petition := petition+1; call ActBroadcast(Leader)
Follower
do: currentState := (currentState+1) mod nDevices
Leader
do: currentState := (currentState+1) mod nDevices

Source: Author's ownership.

between any instances of the associated blocks, rather than that there is a link between particular instances.

If there is an association between the blocks that define the instances directly linked by a connector, we say that this connector realises the association, and that the association defines the type of the connector. Connectors that do not realise an association are not typed.

The ports pD and pB define provided and required interfaces. A provided interface is depicted as a circle and identifies a port that produces outputs to its client. A required interface is depicted as a semicircle and identifies a port that takes inputs from its client.

The number in a port defines its arity. For instance, each device has a port pD with arity one, whereas the port pB of the single instance of the bus has arity 3. This allows the connection of one bus with three devices.

2.1.3 State-machine diagram

The state-machine diagrams of both UML and SysML are compatible (OMG, 2010b, pp. 541). A state-machine diagram reacts to events from the environment, which are stored in an event pool. The order in which events in the pool are processed is unspecified. Figure 2.5 shows an example of a state machine from our case study.

States can be simple or composite. Simple states do not have substates. For instance, Off in Figure 2.5 is a simple state, while On is a composite state. Initial states are depicted as filled circles. Figure 2.5 shows two initial states: the initial state of the entire state-machine diagram and the initial state of the state On.

A state may have three types of behaviour: entry and exit actions, and do activities. Entry actions are executed when the state is activated, exit actions are executed when the state is exited, and do activities are executed when the state finishes activating. Do activities may either stop by themselves or continue indefinitely until a transition interrupts it and exits the state. Figure 2.5 shows the do activities performed by the states Undefined, Follower and Leader. These activities store in the currentState the value of the new state: currentState+1 modulo the number (nDevices) of devices.

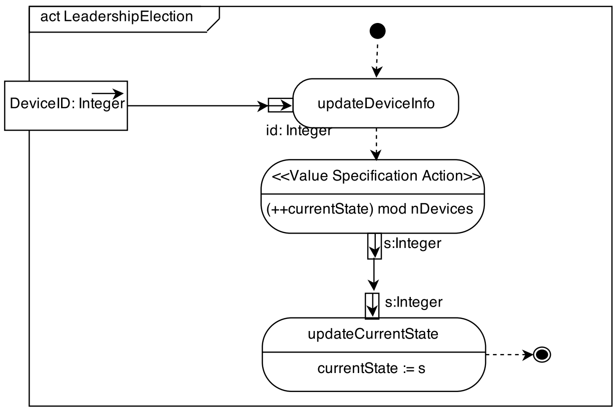
A transition connects a source to a target state; it may contain a trigger, a guard and a behaviour. The trigger and guard are separated from the behaviour by a slash (/). Guards are specified between square brackets. For example, in Figure 2.5, the transition from the state Off to the state On is triggered by a signal turnOn and it has no guard. On the other hand, the transition from Undefined to Leader takes place whenever the state Undefined is active, its do activity has finished, and the guard currentState == id and nLeaders == 0 and petitionAccepted is true (this means that a device becomes a leader whenever its petition has been accepted and there are no leaders). The system returns to Undefined whenever more than one leader is elected. Similarly, the transition from Undefined to Follower takes place whenever Undefined is active, its do activity has terminated, and the guard currentState == id and nLeaders $>$ 0 and not(petitionAccepted) evaluates to true. This means that the device becomes a follower whenever there is at least one leader and it is not itself (because its petition was not accepted). The transition from Follower to Undefined happens whenever the source state is active, the do activity has terminated and nLeaders == 0 holds, i.e. it happens whenever there are no leaders. In our semantics, we restrict behaviours to be any CML action language statement, like the assignment petition := petition + 1 displayed in Figure 2.5. Also, operation calls, signals and call to activities can be used as descriptions of behaviour by using the CML action language. The latter is illustrated in Figure 2.5 where the activity ActBroadcast is invoked in the behaviours of some transitions. The syntax of the CML action language is detailed in Chapter 3.

For the purpose of this work we divide transitions between completion and non-completion transitions. Completion transitions are triggerless transitions that are executed when the internal behaviour of the state (e.g, "do activities") terminates. Non-completion transitions are triggered by events from the event pool and can interrupt the internal events of its source states. For instance, the transition from Off to On is a non-completion transition because it is triggered by the event turnOn, while the transition from Follower to Undefined is a completion transition because it has no trigger, it only requires the do activity from Follower to terminate and the guard between the states to be true.

A transition between two states, whose trigger and guard (if any) are evaluated to true, can be executed by exiting the source state, executing the transition behaviour and entering the target state. In the general case, transitions can cross state boundaries; this requires that not only the source state is exited by the transition, but some of its ancestors too.

Finally, state machines may contain join, forks, junction and choice states. These states are used to control the flow through the states. Join and fork states (both represented by solid bars) gather and split transitions, respectively. A junction state (represented graphically by a solid black circle) provides a mechanism for choosing which transition (leaving the junction state) to follow. Transitions leaving junction states cannot have triggers, and the evaluation of the

Figure 2.6: Example of an activity diagram.



Source: Author's ownership.

guard is performed prior to the execution of the transition. An alternative is the choice state that provides a mechanism for choosing the particular transitions to follow, but, unlike junction states, the decision of which transition to follow is performed during the execution of the transition path. Therefore, the execution of the behaviours of the transitions in the path can affect the outcome of the execution of the choice state.

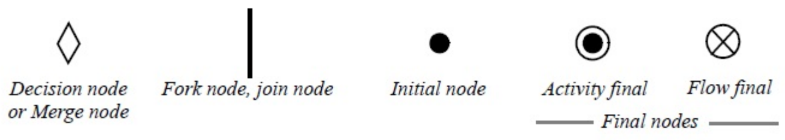
2.1.4 Activity diagram

Activity diagrams are based on classic flow charts. In contrast with the other diagrams, they are normally used for low-level modelling: detail the behaviour of an operation and describe workflows or processes.

An activity diagram has three basic elements: activity nodes, edges and regions. An activity node represents an action, a control or an object. An action node is used to represent some action. In Figure 2.6, the nodes with names updateDeviceInfo, <<Value Specification Action>> and updateCurrentState are action nodes. The node named <<Value Specification Action>> computes the value of an expression (++currentState) mod nDevices that is then sent to updateCurrentState via the output pin s.

Control nodes manipulate the flow of actions, for example, via decisions, forks and joins. Figure 2.7 illustrates the possible control nodes. A decision node chooses among its outgoing edges according to their guards or probabilities. Only one outgoing edge must be chosen. If more than one guard is true, the order of evaluation is not defined. A merge node brings together multiple flows without synchronisation. A fork node splits a flow into multiple concurrent flows. A join node synchronises multiple flows. An initial node starts the flow when the activity is invoked. An activity may have more than one initial node, then invoking an activity starts multiple flows. A flow final node destroys all tokens that arrive at it. It has no effect on other flows in the activity. An activity final node terminates the activity. An activity may have more than one activity final node. The first activity final node reached stops all flows in the activity.

Figure 2.7: Control nodes.



Source: (OMG, 2011).

Object nodes represent data used by an activity: inputs and outputs to the activity or to its nodes. Among the object nodes, the most used ones are parameters pins and datastores. For example, in Figure 2.6, DeviceID is a parameter object node of the activity and its data is taken as input by the node updateDeviceInfo via the pin id, which is also an object node. A datastore is an object node that can repeatedly provide in its outgoing edges the data received until the activity terminates or another data arrives in the node.

Edges can be of two types: control flow or object flow. A control flow defines when and in which order the actions run. Control flows are shown as dashed arrows. An object flow describes how inputs and outputs flow between actions. Object flows are depicted as solid arrows; in Figure 2.6 the DeviceId object is passed to the action updateDeviceInfo and ((++currentState) mod nDevices) is sent to updateCurrentState.

Activity diagrams use a token semantics to control the flow of execution. A node can be executed only when all its inputs have received tokens, and, once it finishes its behaviour, it provides tokens on its outputs. Some nodes create tokens (for instance, an initial node only provides tokens on its outputs), while others remove tokens (for instance, a flow final node only consumes tokens that arrive on it). An activity diagram finishes its execution when there is no token flowing through it or an activity final node is reached.

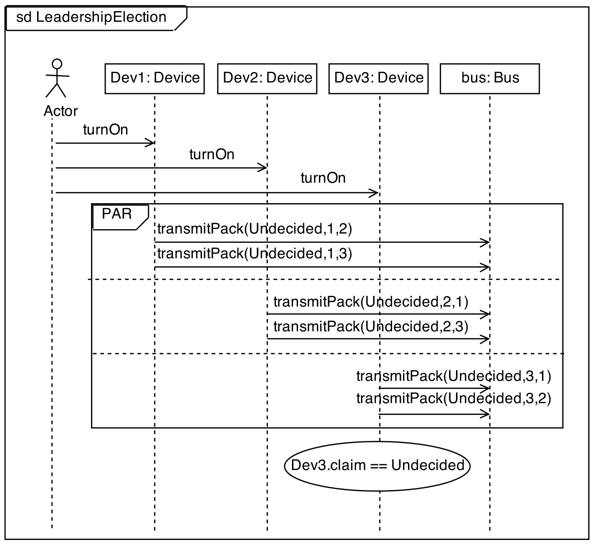
2.1.5 Sequence diagram

In UML 2.0, there are four types of diagrams to describe interactions: sequence diagrams, communication diagrams, interaction overview diagrams, and timing diagrams. Among them, the sequence diagram is the most commonly used to describe interaction aspects of systems, and therefore SysML considers only sequence diagrams to describe interactions. According to the SysML specification (OMG, 2012), communication diagram and interaction overview diagrams are excluded as they overlap in functionality without adding significant capabilities for modelling systems. Timing diagrams are not included because of their lack of maturity and suitability for systems engineering needs.

A sequence diagram describes operational scenarios of a system with an emphasis on order. This is achieved through the use of lifelines. Each participant of the diagram, typically, instances of blocks or its parts, possesses a lifeline, so that we can represent a message-exchange order.

The sequence diagram in Figure 2.8 presents a scenario of the leadership election example

Figure 2.8: Example of a sequence diagram.


sd LeadershipElection
Actor
Dev1: Device
Dev2: Device
Dev3: Device
bus: Bus
turnOn
turnOn
turnOn
PAR
transmitPack(Undecided,1,2)
transmitPack(Undecided,1,3)
transmitPack(Undecided,2,1)
transmitPack(Undecided,2,3)
transmitPack(Undecided,3,1)
transmitPack(Undecided,3,2)
Dev3.claim == Undecided

Source: Author's ownership.

where a user turns on three devices and each of them notifies the bus that the leader is undefined. The user is depicted as an actor, while the three Devices Dev1, Dev2, Dev3 are instances of block Device and bus is an instance of block Bus. A lifeline is represented by a dashed vertical line under each participant.

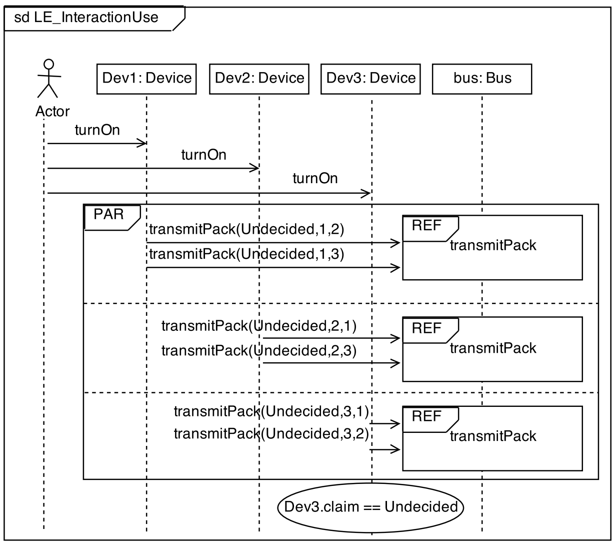
Participants communicate via messages. For example, Figure 2.8 shows the actor sending messages to turn on devices. Messages are sent in sequence along a participant lifeline. So, the first message sent by the actor goes to Dev1, the second to Dev2, and the third to Dev3. However, messages from different lifelines may happen in any order. For example, the second message sent by the actor that goes to Dev2 and the first message sent by Dev1 that goes to the Bus may happen in any order. This mechanism is called weak sequencing. Messages are linked to two message events, one for sending and another for receiving the current message.

Messages can be of three types: asynchronous (open arrow head), synchronous call (closed arrowhead), or reply from a synchronous call (dashed arrow). All messages shown in Figure 2.8 are asynchronous.

Message exchanges can be grouped inside combined fragments that describe operators like parallel composition, conditional, loops, and so on. Figure 2.8 shows a parallel combined fragment PAR, which must be composed of two or more operands, separated by horizontal dashed lines, describing scenarios that can proceed in interleaving. In Figure 2.8, there are three operands, each sending two messages in sequence. For instance, transmitPack(Undecided,1,2) is sent before transmitPack(Undecided,1,3). These messages, however, can be interleaved with those sent in the other operands.

A lifeline can include a state invariant: a constraint on the blocks. If the constraint holds, any message-exchange order just established is a valid scenario of the system, otherwise, it is

Figure 2.9: Example of a sequence diagram with *interactionUse*.


sd LE_InteractionUse
Actor
Dev1: Device
Dev2: Device
Dev3: Device
bus: Bus
turnOn
turnOn
turnOn
PAR
transmitPack(Undecided,1,2)
transmitPack(Undecided,1,3)
REF
transmitPack
transmitPack(Undecided,2,1)
transmitPack(Undecided,2,3)
REF
transmitPack
transmitPack(Undecided,3,1)
transmitPack(Undecided,3,2)
REF
transmitPack
Dev3.claim == Undecided

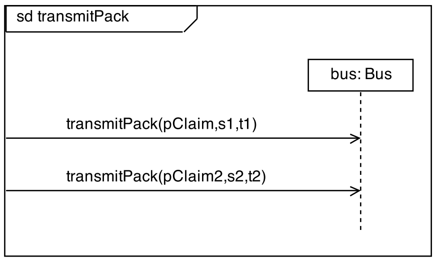
Source: Author's ownership.

forbidden. State invariants define properties of the system in terms of the attributes of a block. At the bottom of the lifeline Dev3 in Figure 2.8, there is an invariant Dev3.claim == Undecided, which verifies that the value of the attribute claim of Dev3 is Undecided after it has been turned on and all its packages have been transmitted.

A sequence diagram can include an *interactionUse* element to refer to another sequence diagram, so that part of its definition is provided by the referred diagram. In this way, a diagram can be used several times in the definition of others. For example, Figure 2.9 shows another way of describing the scenario of Figure 2.8 using *interactionUse* elements that refer to the sequence diagram transmitPack, which is shown in Figure 2.10.

Messages outside the *interactionUse* can connect to or come from an *interactionUse* via a gate. This is a connection point that relates a message from outside the *interactionUse* with a message inside the *interactionUse*. For instance, the diagram of Figure 2.10 has two gates, one for each transmitPack() message that comes from outside. These gates connect to the transmitPack() messages of the diagram in Figure 2.9 that arrive in the *interactionUse* elements.

Figure 2.10: Example of a sequence diagram with gates.



Source: Author's ownership.

2.1.6 Final Remarks on SysML models

A complete SysML model contains several diagrams. Each diagram provides a different point of view of a system that must complement and be consistent with the others. State machine and activity diagrams describe behaviours using the attributes and calling the operations of blocks. A state machine diagram may call an activity. State machine and activity diagrams may call an operation and send a signal to a block. Finally, a sequence diagram describes a scenario where exchanged messages between blocks are used to represent either operation calls or transmission of signals.

2.2 CML

The COMPASS modelling language (CML) (WOODCOCK et al., 2012a) is a formal specification language that integrates a state based notation (based on VDM++ (FITZGERALD; LARSEN, 2009)) and a process algebraic notation (based on CSP (HOARE, 1985)), as well as Dijkstra's language of guarded commands (DIJKSTRA, 1975) and the refinement calculus (BACK; WRIGHT, 1998). It supports the specification and analysis of state-rich distributed specifications.

We introduce CML by means of a specification of a simple clock. For more details, see the work by Woodcock et. al (WOODCOCK et al., 2012a,b).

A CML specification consists of a number of paragraphs that, at the top level, can declare types, classes, functions, values (i.e., bindings of expressions to identifiers), channels, channel sets, and processes. Both classes and processes declare state components, and may contain paragraphs declaring types, values, functions and operations. A process may additionally declare any number of actions, and must contain an anonymous main action, which specifies the behaviour of the process.

Initially, we specify a simple clock whose only observable behaviour is a synchronisation on a channel `tick` used to mark the passage of seconds.

```
channels
  tick
```

As previously mentioned, a class declares a state and a number of operations that manipulate the state. In particular, a class may declare an initialisation operation identified by the keyword **`initial`**, which acts as a constructor of the class.

Internally, the clock has a state variable `s` that records the number of seconds that have elapsed, and has two operations defined: `Init()` and `increment()`. The first operation simply initialises the state with 0, while the second adds one to the state component. The clock state is captured by the following class declaration.

```
class ClockSt = begin
  state
    public s: nat
  initial
    public Init()
    frame wr s
    post s = 0
  operations
    public increment()
    frame wr s
    post s = s~ + 1
end
```

The **frame** keyword in the declaration of operations specifies the state components that can be read (**rd**) and written (**wr**) by the operation. In the case of the `Init` operation, the state component `s` can be written by `Init`. The **post** keyword specifies the post condition of the operation. In the case of `Init`, the post condition states that the state component `s` (after the operation) is equal to zero. The post condition of the operation `increment` equates the state component `s`, after the operation, to the sum of its initial value (`s~`) and one.

The class `ClockSt` is used in the definition of the process that specifies the behaviour of our clock. Similarly to a class, a process encapsulates a state and may contain operations, but in addition it contains at least one action (the main action), which specifies the behaviour of a process.

Our simple clock initialises its state, waits for one time unit, which we take to mean one second, increments its counter and synchronises on `tick`. This is specified by the following process declaration.

```
process SimpleClock = begin
  state c: ClockSt
  actions
    Ticking = Wait 1; c.increment(); tick -> Skip
  @ c.Init(); mu X @  Ticking; X
end
```

The simple clock is a process that declares a state and a number of actions. The state, in this case, is formed of a single state component `c` of type `ClockSt`. The actions include `Ticking` and the action started by `@`, which is the main action of the process. In this case, `SimpleClock` simply initialises its state by calling the operation `Init()` of the state component `c` and recursively (**mu**) calls the action `Ticking`. This action waits for one time unit (**Wait** `1`), increments the internal counter and synchronises on the channel `tick`.

In our initial specification of the clock the only observable event is the synchronisation on `tick`. It might be interesting to have a clock that takes advantage of its internal counter and supplies information about how many seconds, minutes, hours and days have elapsed.

We now extend our simple clock to include this additional functionality. First, we declare four additional channels that communicate a natural number. They are used to query the seconds, minutes, hours and days that have elapsed.

```
channels
  second, minute, hour, day: nat
```

The new clock specification is similar to the simple clock; it declares the state of the process as containing just the component `c` of type `ClockSt`, but additionally defines three functions: `get_minute`, `get_hour` and `get_day`. They take the number of seconds recorded in the state, and calculate, respectively, the equivalent number of minutes, hours and days.

```
process Clock = begin
  state c: ClockSt
  functions
    get_minute(s: nat) m: nat
    post m = s/60

    get_hour(s: nat) h: nat
    post h = get_minute(s)/60

    get_day(s: nat) d: nat
    post d = get_hour(s)/24
```

The ticking action remains the same as before, but we add a new action, `Interface`, that provides the extra functionality.

```
  actions
    Ticking = Wait 1; c.increment(); tick -> Skip

    Interface = second!(c.s) -> Interface
      [] minute!(get_minute(c.s)) -> Interface
      [] hour!(get_hour(c.s)) -> Interface
      [] day!(get_day(c.s)) -> Interface
```

This action simply offers an external choice (`[]`) of communication over the channels `second`, `minute`, `hour` and `day`, and recurses. Each communication outputs (indicated by `!` after a channel name) the appropriate value calculated using the functions previously defined.

Now, the main action of the new clock is slightly different. It first initialises the state as before, but instead of offering `Ticking` alone, it composes `Ticking` in parallel with the recursive action `Interface` with the option of interrupting (`/_\`) `Interface` with a synchronisation on tick. The operator A1 [|ns1| cs |ns2|] A2 composes the actions A1 and A2 in parallel based on a set of events cs on which the two parallel actions synchronise, and two name sets ns1 and ns2 that partition the state of the process and indicate which state components can be updated by the left (ns1) and right (ns2) parallel actions. In our example, the action Ticking can update the state component c and the right parallel action does not update the state. The parallel actions synchronise on the channel tick.

```
  @ c.Init(); mu X @ (
    Ticking [|{c}|{|tick|}|{}|] (Interface /_\ tick -> Skip)
  ); X
end
```

While Ticking is waiting, the right hand side of the parallelism can offer any number of interactions over the channels in Interface. When Ticking finishes waiting, s is incremented, and the parallelism synchronises on tick. In this case, the action Interface is interrupted and both sides of the parallelism terminate. At this point, the recursive call (on X) takes place.

When the parallelism starts, both parallel actions take a copy of the state, and when the parallelism terminates, the state is updated based on the changes performed by the two parallel actions (on their copies of the state) and the partition of the state. A consequence of this is that changes to the state performed by Ticking can only be reflected on the behaviour of Interface when the parallelism terminates, the state is updated and Interface restarts (as part of the recursive call) with a copy of the updated state.

Now we have a clock that not only signals the passing of time, but can also output the time. However, we might also want to be able to restart the clock. For this, we define a channel restart and a new clock RestartableClock.

```
channels
  restart
```

The definition of the restartable clock is similar to that of the process Clock previously defined. The process RestartableClock has a new action Cycle, and its main action offers the action Cycle and the possibility of interrupting it through the channel restart. If the interruption takes place, the main action recurses and Cycle is called to reset the state.

```
process RestartableClock = begin
  state c: ClockSt
  functions
    get_minute(s: nat) m: nat
    post m = s/60

    get_hour(s: nat) h: nat
    post h = get_minute(s)/60
    get_day(s: nat) d: nat
    post d = get_hour(s)/24
  actions
    Ticking = Wait 1; c.increment(); tick -> Skip

    Interface = second!(c.s) -> Interface
      [] minute!(get_minute(c.s)) -> Interface
      [] hour!(get_hour(c.s)) -> Interface
      [] day!(get_day(c.s)) -> Interface

    Cycle = c.Init(); mu X @ (
      Ticking [|{c}|{|tick|}|{}|] (Interface /_\ tick -> Skip)
    ); X

  @ mu X @ Cycle /_\ restart -> X
end
```

We can further extend the functionality of the clock by specifying a multi-clock. A simple way of defining such a clock is to compose a number of restartable clocks (or any other variety of clock). This raises the question of how the clocks are composed. For instance, do all clocks synchronise on tick? Can they be restarted independently? We present below two processes that model a multi-clock. Both of them assume that the clocks are synchronous, but the first process allows independent restarting, while the second does not.

First, we define the channels that allow the environment to communicate with specific clocks. We assume that the clocks in the multi-clock are indexed using natural numbers, and are similar to those already defined, communicating a natural number (the identifier of the clock) and the value originally communicated. We prefix the name of the new channels with an `i`.

```
channels
  isecond, iminute, ihour, iday: nat * nat
  irestart: nat
```

Our first model of a multi-clock is specified by the process `NRestartableClocks1`. This is a parametrised process that takes the number `n` of clocks, and starts `n` copies of the process `RestartableClock` running in parallel and synchronising on `tick`. The channels in the `RestartableClock` process need to be renamed to distinguish one clock from another. We rename each channel (except `tick`) to its `i` version.

```
process NRestartableClocks1 = n: nat @
  [|{|tick|}|] i: {1,...,n} @
    RestartableClock[[second <- isecond.i,
              minute <- iminute.i,
              hour <- ihour.i,
              day <- iday.i,
              restart <- irestart.i]]
```

Our alternative process `NRestartableClocks2` is similar, except that the different clocks synchronise on `restart` as well, and this channel is not renamed. Thus, a synchronisation on `restart` restarts all the clocks simultaneously.

```
process NRestartableClocks2 = n: nat @
  [|{|tick, restart|}|] i: {1,...,n} @
    RestartableClock[[second <- isecond.i,
              minute <- iminute.i,
              hour <- ihour.i,
              day <- iday.i]]
```

One might consider that, whilst these definitions are reasonably intuitive, they are not the most efficient for implementation purposes due to the required synchronisations between the different clocks. So, one might implement a multi-clock with a single clock that provides all services on the `i` channels. To model this design, we simply associate each channel of a restartable clock with the equivalent `i` channel, but ranging over all the possible clocks.

```
process NRestartableClocksImpl = n: nat @
  RestartableClock[[second <- isecond.i,
            minute <- iminute.i,
            hour <- ihour.i,
            day <- iday.i | i in set {1,...,n}]]
```

This process simply renames each channel of `RestartableClock` (except `tick` and `restart`) to a set of communications on the associated `i` channel communicating the identifiers of the clocks.

This process raises the question of which of our multi-clock processes is being implemented by `NRestartableClocksImpl`. These questions can be formulated using refinement, which is a behaviour-preserving relation between processes. It is a core concept in CML and CSP. If a CML process `P` is refined by another process `Q`, then `Q` can only engage in interactions that are possible for `P`, and can only deadlock or livelock, when `P` can. Regarding the restartable clock processes, we can verify if the `NRestartableClocksImpl` process respects the behaviours of the `NRestartableClocks1` and `NRestartableClocks2` process with the following refinement assertions.

```
assert NRestartableClocks1 [= NRestartableClocksImpl
assert NRestartableClocks2 [= NRestartableClocksImpl
```

The first assertion states that the process `NRestartableClocksImpl` is a refinement of the process `NRestartableClocks1`, and the second asserts that the implementation is a refinement of `NRestartableClocks2`. These assertions can be checked using a model-checker. The first assertion is not valid because the `restart` events of `NRestartableClocksImpl` are not part of the executable alphabet of `NRestartableClocks1` (they are renamed to `irestart.i`), hence, the model checker returns a counterexample with a trace having such an event. On the other hand, the second assertion is valid.

More details about the CML notation will be presented as needed. Moreover, the semantics of CML are provided in technical reports of the COMPASS project (WOODCOCK et al., 2012b, 2013).

2.3 Final Remarks

The next chapters describe how a SysML semantics can be defined in terms of CML. However, not all SysML models can be translated to CML because whilst SysML models can be ambiguous, CML models cannot. Therefore, we must restrict the way SysML models are designed to allow their representation in terms of well-defined CML specifications. Hence, not all SysML models can have a sibling CML specification. The next chapter details the restrictions we impose on the SysML models.

3 Modelling Patterns

One of the main features of SysML (which is inherited from UML), namely its flexibility, hinders the task of enriching it with a formal semantics. Certain uses of the notations are not addressed in the informal semantics, and even explicitly allowed omissions (like operation definitions) can lead to incomplete SysML models that could not be mapped to meaningful CML models. To avoid these problems, we propose guidelines of usage for SysML.

These guidelines provide a means to define concrete SysML models in order to enable the generation of meaningful CML models. They also can be seen as a modelling style to guide practitioners during the design of systems. In addition, they impose roles for the modelling elements, for instance, behaviours can only be described by state machines and activities while sequence diagrams are only used for validation of the model. Therefore, we limit the purpose of each element in the model. Moreover, some guidelines require a minimum level of information to be available in the model, hence, models too abstract cannot be tackled by our semantics. That is why we position our approach in the design phase of systems when such required information should be available.

In this chapter, we describe the assumptions for defining the semantics of SysML models. They act upon the SysML models themselves, and establish what is required of a model for a CML model to be ascribed to it. We also discuss in this chapter the metalanguage we use to describe our semantic mappings from SysML to CML and we present an overview of how this semantics is defined. Section 3.1 describes the guidelines and Section 3.2 illustrates them using examples that are case studies of this thesis. Section 3.3 gives details about the conventions of the metalanguage used to describe our translation rules, while Section 3.4 presents an overview of the semantics for SysML models that are presented in chapters 4, 5, 6 and 7.

3.1 Guidelines of usage

We identify here a subset of the SysML notation and describe usage guidelines to allow us to define a semantics via a translation into CML. Our subset of SysML includes block definition diagrams, internal block diagrams, state machine diagrams, activity diagrams and sequence diagrams. Next, we cite the constructors considered for each diagram.

- Block definition and internal block diagrams: blocks, operations, attributes, signals, ports, provided and required interfaces, associations, composition and generalisation.
- State machine diagrams: simple and composite states, regions, final states, initial and junction pseudostates, fork and merge pseudostates, transitions, and completion and deferred events[1].
- Activity diagrams: control nodes (initial, activity final, flow final, decision, merge, join and fork), action nodes (accept event, send signal, call operation, call behaviour, value specification, structured feature, read self, opaque), object nodes (parameter, input and output pins, datastore), control and object flow, and interruptible regions.
- Sequence diagrams: lifelines, messages (synchronous, asynchronous and reply), message arguments, local attributes of the Interaction, combined fragments (PAR, STRICT, SEQ, ALT, OPT, BREAK, LOOP, CRITICAL), state invariant, interactionUse and gates.

Our guidelines of usage aim at supporting the definition of interesting cohesive, comprehensive, and formal representations of SysML models. The guidelines maximise the *definedness* of a SysML model at both the entity definition level and at the instance definition level. In this respect, they are similar to constraints imposed by most tools to enable automatic code generation from models.

A SysML model typically provides independent (although expected to be consistent) and disconnected views of an application (much like a set of classes in an object-oriented program without a main class that coordinates all other classes). To generate a CML model (or indeed code), we need a model with an element (typically a block) that represents the application as a whole. Nevertheless, this block can be described in terms of several other blocks. Our guidelines for maximal *definedness* at the entity level, however, still allow for models that provide independent disconnected views. The guidelines for *definedness* at the instance level, therefore, require only that enough information is available in the SysML model to link models provided by the diagrams. Furthermore, the guidelines include some restrictions that simplify the semantics and make it more amenable to automated reasoning.

3.1.1 Entity definition guidelines.

Our guidelines for entity definitions are:

1. Operation calls must be treated either by the block's state machine diagram or by an activity diagram;

[1]Although the state machine semantics is not a contribution of this thesis, the guidelines for the design of translatable state machines are.

2. In the block definition diagram there must be one block representing the system as a whole (a root block).
3. The blocks in the model must form a connected graph where the edges are either generalisation or composition relations;
4. A composite block (the head of the composition relation) must not have attributes, operations, signals, activities or state machines;
5. Associations in the block definition diagram must match the internal block diagram typed connectors;
6. Associations must be used in place of aggregation; and
7. Connectors between ports are not typed.

The second guideline guarantees that there is a root block of the model from which the semantics of the whole model can be defined.

The third guideline guarantees that all other blocks can be reached from a block in the model. We observe that this is essential to ensure that we have a connected model of the application, but it is not enough, since it is concerned only with block diagrams.

The fourth guideline requires that a composite block is simply the composition of its parts. The reason for this restriction is that it is not clear what it means for a composition of blocks to have an associated behaviour (in the form of a state machine, for instance). We envisage two possible interpretations: (1) the state machine is a specification of the desired behaviour of the composition of the parts, or (2) the state machine specifies a behaviour that is specific to the composite block and must be executed in parallel with the parts of the block. In the first interpretation, one approach would have the state machine of the composite block running in parallel with the composition of the parts, and synchronising on all events. In this case, a deadlock originating in that parallelism would indicate that the parts do not correctly implement the state machine. If the second interpretation is the intention, it is possible to write a model that captures this behaviour and follows our guidelines. We need to create a new part and move all components of the composite block to the new part. In this case, it would be possible to adequately specify the dependencies between the behaviours in the new part with respect to the original parts. Whilst operations and signals are not allowed in the composite block, ports are supported and operations and signals on the ports are allowed. This restriction forces the user to indicate, through the use of connectors and interfaces, which parts treat which requests received at the ports of the composite block.

The fifth and sixth guidelines enforce our view that whilst we know that composition is a relation where a part cannot live without its owner, the distinction between aggregation and association is not clear at all. We require that associations are used instead of aggregation, and that associations are matched by connectors at the instance level. The block definition diagram shown in Figure 2.3 on page 26 respects these guidelines because the block SoS are related to blocks Device and Bus through a composition relationship. Our view is motivated by the fact that composition entails uniqueness in the sense that the same instance cannot be part of two different blocks, and therefore the semantics of the parent block is directly defined in terms of the sub-blocks. Moreover, since the instances of the sub-blocks that compose an instance of the parent block are unique, we do not require any additional information to instantiate them as long as we can guarantee that they are unique.

Aggregation and association on the other hand allow sharing of block instances, and without the uniqueness assumption, we require additional information to determine whether instances are shared and how. Since block definition diagrams cannot refer to particular instances, any aggregation or association present in the block definition diagram must be further specified in the internal block diagram to provide this information. For associations, a connector in the internal block diagram must be typed by an association in the block definition diagram. In the case of aggregation, the same approach is not possible because connectors cannot be typed by aggregations. For this reason, our guidelines do not allow the use of aggregation.

Finally, since a connection between ports is specified by the provide and required interfaces of the connected ports, typing information on the connector is unnecessary.

3.1.2 Instance-level guidelines.

Instance-level guidelines are related to internal block diagrams.

1. Each owner block of a composition relation (for example, block SoS in Figure 2.3) must specify the connections between its parts through an internal block diagram.

2. All part blocks in a composition whose lower-bound multiplicity is larger than 0 must appear in the internal block diagram of the owner block in numbers compatible with their multiplicities.

3. The cardinalities that appear in an internal block diagram must be constants.

4. Ports may only be connected to other ports.

The internal block diagram specifies exactly which instances are present in a particular realisation of the system, and exactly how they are associated. This is enforced by the first three guidelines above.

The last guideline is necessary because a connector starting in a part (not a port) must correspond to an association of the part, which in turn must appear in the block definition diagram. A port can be connected directly to a block, but the meaning of such a connection is not defined in SysML. In this case, restriction 4 above is simply a consequence of the entity-level restriction 5.

3.1.3 Action language assumptions.

Since the action language of SysML is not defined, we require actions to be expressed in a subset of CML that can be used to define data operations, enriched with statements for sending a signal or calling an operation of another block through a port or association, and for calling an activity. CML reactive behaviour is not allowed in this subset, since the SysML paradigm does not include channel-based communication to model interactions, like CML. Instead, attributes, signals and operations define the services provided by an application. SysML actions are, therefore, data operations that explain how the state embedded in blocks and other operations can be used to specify the services. We could use any data language to specify such actions, and we choose the CML data language for convenience. The really important point is that the language is defined and it can be used in constructors that specify behaviours, like the action behaviour of a transition in a state machine or in opaque actions in activities.

In the context of state machine diagrams, we extend the subset of CML used as action language to include the following constructs:

Block operation call without return value `Op(p1,...,pn)`, where `Op` is the name of the operation

Block operation call with return value `v := Op(p1,...,pn)`, where `Op` is the name of the operation

Activity call `call Act(p1,...,p2)`, where `Act` is the name of the activity

return statement `return v`

Whilst the syntax of these constructs is similar to the syntax of CML, their semantics is not that of CML. The return statement is only allowed in transition actions triggered by a call event. This restriction is necessary because the return statement is part of the definition of an operation and, as such, must be associated with an operation definition. For state machine diagrams, this is the case only for actions triggered by an operation call event. It is important to note that we do not require the knowledge of how the translation functions work in order to specify any CML action language command.

3.1.4 Simplification assumptions.

The simplification assumption guidelines provide mandatory patterns for modelling systems whose purpose is to simplify the semantics. They are:

1. Each block must either have one associated state machine diagram or no other associated diagram specifying its behaviour;

2. Sequence diagrams specify valid and forbidden scenarios of interactions between blocks;

3. Operations are always synchronous. Asynchronous operations should be modelled as signals;
4. Operations modelled by activities and state machines must not call themselves;
5. Operations modelled by activities and state machines must not call each other;
6. Activities must not be recursive (mutually or otherwise);
7. Sequence diagrams must not be recursive (mutually or otherwise);
8. Ports do not have an associated behaviour;

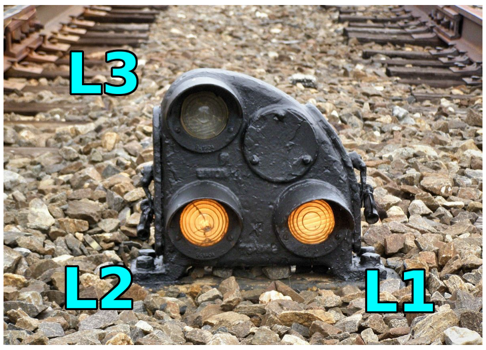
The first guideline establishes that the only valid form of restricting the behaviour of a block is by means of state machines. Essentially, this forbids the use of activity and sequence diagrams for the specification of the overall behaviour of a block. State machine diagrams are particularly well suited for this task, since they support deferred events (useful to model blocking behaviour) as well as call events (useful to model operations). Moreover, there is a convention (FRIEDENTHAL et al., 2011) for modelling operations that return values (i.e., the **`return`** keyword) in a state machine, and the semantic definition of state machine diagrams covers such convention.

The second guideline restricts the possible uses of sequence diagrams to scenario validation purposes. This use is consistent with various case studies and recommendations (FRIEDENTHAL et al., 2011). Although sequence diagrams have been used for other purposes (e.g., operation definition), it is our view that the emphasis on interaction between blocks given by sequence diagrams makes them more suitable to model such scenarios than, for instance, to model operation definition.

The third guideline forbids the use of asynchronous operations. The reason for this guideline is that asynchronous operations give rise to some unclarities. It is not clear what is the intended meaning of an asynchronous operation with return values, and the difference between asynchronous operations and signals. If an asynchronous operations is required, we suggest the use of signals.

The next four guidelines restrict the use of recursion in operations defined in state machines and activities, also in the definition of sequence diagrams. One of the main reasons is because there is no definition of the semantics for these facilities in UML or SysML. Moreover, scenarios like this would result in untreatable CML models due to the state space explosion problem (as we use parallelism to compose the model elements, recursion at this level would result in a infinity parallel composition). Therefore, we decided to restrict such possibilities in our semantics. Although recursion is a widely used technique in programming, it is not common in modelling artefacts, probably because they are used for defining abstraction with a higher level of detail than those needed in code. We do not recall facing any recursive sequence diagram

Figure 3.1: An illustration of a dwarf signal.



Source: (FOSTER; WOODCOCK, 2013).

or state machine, however, as SysML and UML also do not restrict these uses, we decided to explicitly forbid them.

Finally, the eighth guideline restricts the use of ports with behaviour. For simplicity, ports act just as relays of messages (operation calls and signals). Although we know that sometimes it is suitable to describe the protocol of communication of a port in terms of a state machine, we do not allow this possibility of modelling in our semantics. However, such a protocol can be described in the state machine of the block that owns the port. The modelling pattern for describing a port with behaviour is one possible extension to our work. There is no tool support to check the adequacy to our guidelines, however, as it is further discussed in Chapter 8, there is a tool to translate the SysML models to CML specifications, and in case the guidelines are not respected, then the corresponding specification is generated with syntactic errors. However, we plan to implement the guidelines in the future using a constraint language, like OCL (WARMER; KLEPPE, 2003).

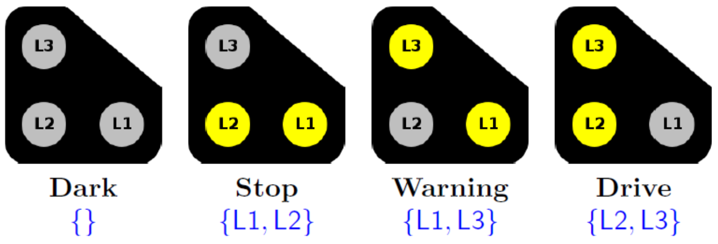
3.2 Examples

In this section we present two case studies that follow our guidelines and that are used along this document, the dwarf signal and the leadership election problem. We have modelled the case studies in SysML with the support and validation of the experts that provided them during the COMPASS project.

3.2.1 Dwarf Signal

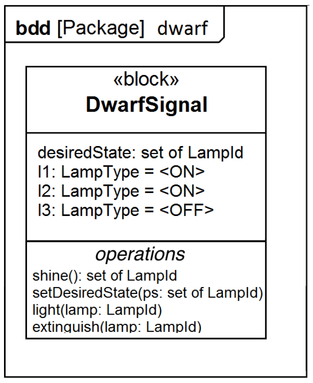
Our first example is the model of a dwarf signal, which is a railway signal used at the side of the track. It has three lamps which are displayed in different configurations to give instructions to train drivers. An illustration is shown in Figure 3.1. The signal's three lamps are named L1-L3, as illustrated, and different configurations of a signal can be written using set notation. For instance, {L1, L2} means **Stop**.

Figure 3.2: The proper states of a dwarf signal.



Source: (FOSTER; WOODCOCK, 2013).

Figure 3.3: The BDD of a dwarf signal.



Source: Author's ownership.

The signal has a total of four proper states that are the well-defined commands to a driver. These are enumerated in Figure 3.2. When all lamps are off (indicated by the empty set { }) the signal is in the **Dark** state, which is a power saving mode for when the signal is not in use. The three remaining proper states **Stop**, **Warning** and **Drive** are indicated by a combination of two lamps and correspond to the positions of an old-style semaphore signal.

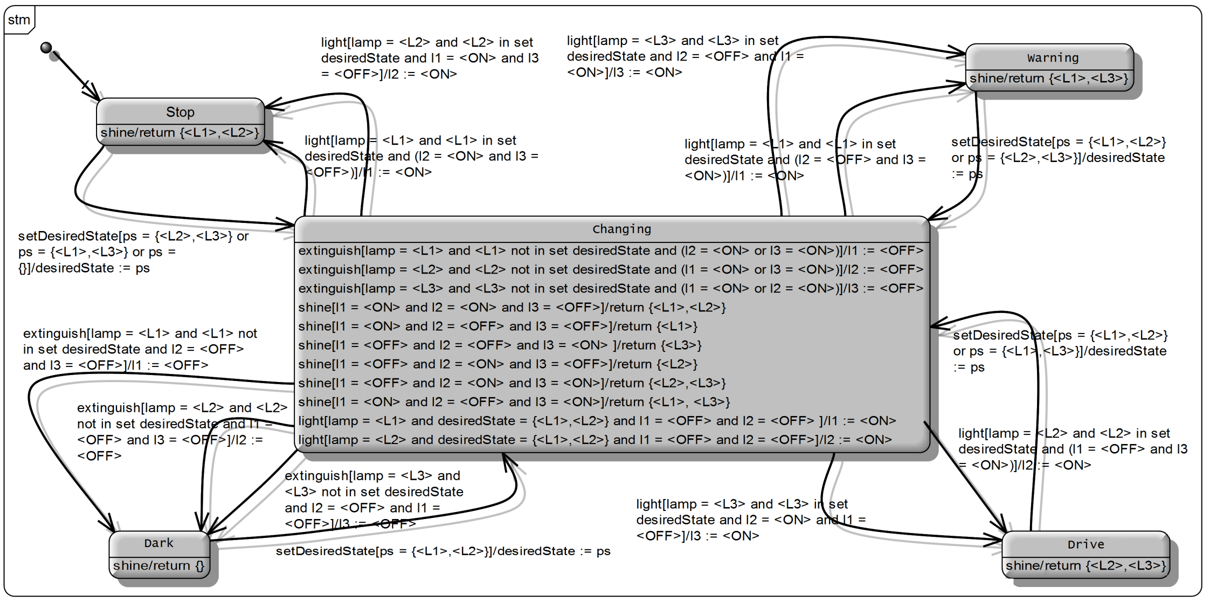
Along with the four proper states there are also three transient states which describe the unstable states when a signal is moving from one proper state to another, since the signal may only light or extinguish one lamp at a time. These states are {L1}, {L2} and {L3}. Finally there is the ambiguous state {L1, L2, L3} which a signal should never display as it is meaningless. This makes a total of $2^3 = 8$ states.

To ensure the safety of this system, four safety properties are given:

- Only one lamp may be changed at once
- All three lamps must never be on concurrently
- A change to or from Dark is allowed only from Stop or to Stop, respectively

We present a simple SysML model that represents the behaviour of a dwarf signal. Figure 3.3 shows the block **DwarfSignal**, which is composed of four attributes, **desiredState** which corresponds to the state that should be reached by the dwarf signal, and three lamps **l1**,

Figure 3.4: The state machine of the **DwarfSignal** block.


stm
light[lamp = <L2> and <L2> in set desiredState and l1 = <ON> and l3 = <OFF>]/l2 := <ON>
light[lamp = <L3> and <L3> in set desiredState and l2 = <OFF> and l1 = <ON>]/l3 := <ON>
Warning
shine/return {<L1>,<L3>}
Stop
shine/return {<L1>,<L2>}
light[lamp = <L1> and <L1> in set desiredState and (l2 = <ON> and l3 = <OFF>)]/l1 := <ON>
light[lamp = <L1> and <L1> in set desiredState and (l2 = <OFF> and l3 = <ON>)]/l1 := <ON>
setDesiredState[ps = {<L1>,<L2>} or ps = {<L2>,<L3>}]/desiredState := ps
setDesiredState[ps = {<L2>,<L3>} or ps = {<L1>,<L3>} or ps = {}]/desiredState := ps
Changing
extinguish[lamp = <L1> and <L1> not in set desiredState and (l2 = <ON> or l3 = <ON>)]/l1 := <OFF>
extinguish[lamp = <L2> and <L2> not in set desiredState and (l1 = <ON> or l3 = <ON>)]/l2 := <OFF>
extinguish[lamp = <L3> and <L3> not in set desiredState and (l1 = <ON> or l2 = <ON>)]/l3 := <OFF>
shine[l1 = <ON> and l2 = <ON> and l3 = <OFF>]/return {<L1>,<L2>}
shine[l1 = <ON> and l2 = <OFF> and l3 = <OFF>]/return {<L1>}
shine[l1 = <OFF> and l2 = <OFF> and l3 = <ON>]/return {<L3>}
shine[l1 = <OFF> and l2 = <ON> and l3 = <OFF>]/return {<L2>}
shine[l1 = <OFF> and l2 = <ON> and l3 = <ON>]/return {<L2>,<L3>}
shine[l1 = <ON> and l2 = <OFF> and l3 = <ON>]/return {<L1>, <L3>}
light[lamp = <L1> and desiredState = {<L1>,<L2>} and l1 = <OFF> and l2 = <OFF>]/l1 := <ON>
light[lamp = <L2> and desiredState = {<L1>,<L2>} and l1 = <OFF> and l2 = <OFF>]/l2 := <ON>
extinguish[lamp = <L1> and <L1> not in set desiredState and l2 = <OFF> and l3 = <OFF>]/l1 := <OFF>
setDesiredState[ps = {<L1>,<L2>} or ps = {<L1>,<L3>}]/desiredState := ps
extinguish[lamp = <L2> and <L2> not in set desiredState and l1 = <OFF> and l3 = <OFF>]/l2 := <OFF>
light[lamp = <L2> and <L2> in set desiredState and (l1 = <OFF> and l3 = <ON>)]/l2 := <ON>
extinguish[lamp = <L3> and <L3> not in set desiredState and l2 = <OFF> and l1 = <OFF>]/l3 := <OFF>
light[lamp = <L3> and <L3> in set desiredState and l2 = <ON> and l1 = <OFF>]/l3 := <ON>
Dark
shine/return {}
setDesiredState[ps = {<L1>,<L2>}]/desiredState := ps
Drive
shine/return {<L2>,<L3>}

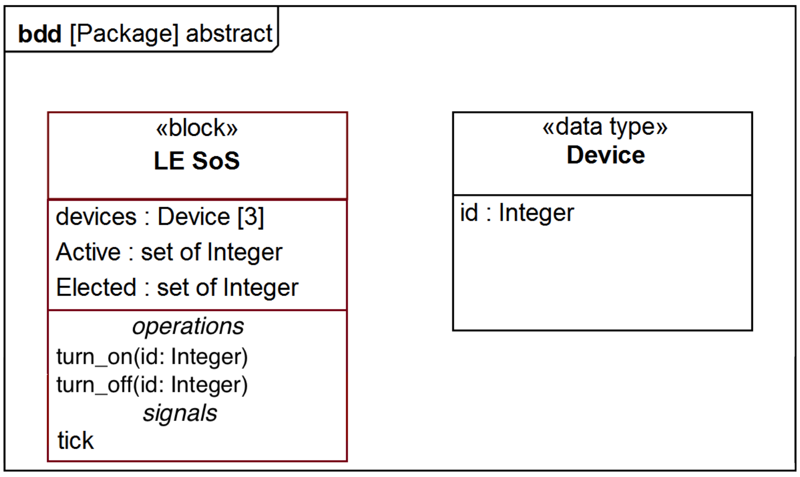
Source: Author's ownership.

l2 and **l3** that are initialised in the **Stop** configuration. The types **LampId** and **LampType** are enumerations where the former has three values: <**L1**>, <**L2**> and <**L3**>; and the latter has two values: <**ON**> and <**OFF**>. Finally the block has four operations. The **shine()** operation returns the set of lamps that are on, the **setDesiredState()** operation updates the value of attribute **desiredState**, the **light()** operation turns a lamp on and the **extinguish()** operation turns a lamp off. This block definition diagram follows our guidelines because it has only one block, which is the root block of the system.

The behavioural description of this block is modelled in terms of a state machine, which is depicted in Figure 3.4. It has the four proper states, the intermediate state where only one lamp may be on and the possible transitions between the states. Each transition deals with a call event to one of the operations. For example, if the system is in the **Stop** state, it can only shine the lamps **L1** and **L2**. In other situations the transitions are triggered according to guards, for example, from the **Dark** state the dwarf signal can only change to **Stop**, then, the set parameter of the **setDesiredState** operation can only have the value **{**<**L1**>**,**<**L2**>**}**, which means going to the **Stop** state.

This state machine also respects our guidelines because it describes the behaviour of the **DwarfSignal** block and the operations of the block are treated by the state machine model. The sequence diagrams describing scenarios and an activity diagram of the Dwarf Signal are described in Chapter 8 when we discuss how to analyse this model.

Figure 3.5: Abstract model: block definition diagram



Source: Author's ownership.

3.2.2 The Leadership Election Problem

This example is a system (of systems) that includes a number of devices running in parallel and cooperating among themselves. Typically, such a device could be a cell phone, a MP3 player, a home theatre, a TV, a set of speakers, etc., all running in parallel and sharing information. The main requirement is that among the active devices there is exactly one leader. We provide two possible models: an abstract model that has a simpler specification of the desired behaviour, and a concrete model that has more details related to the implementation level. The abstract model is specified in Section 3.2.2.1. The concrete model uses a distributed architecture with no centralised control and is defined in Section 3.2.2.2 (and it is already partly described in Section 2.1).

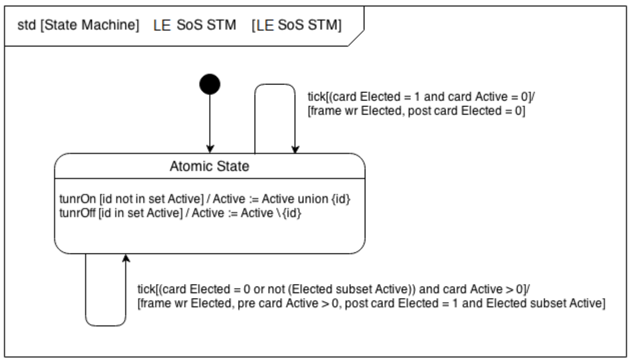
3.2.2.1 The Abstract SysML model

The abstract model specifies a centralised system that controls any number of devices. To follow our guidelines, we assume a fixed number of devices: three. The goal of the system is to maintain itself in a state where, if there are active devices, exactly one of them is identified as a leader and all others are marked as followers.

The system is modelled by the block LE SoS in Figure 3.5, which has three attributes, devices, Active and Elected, the operations turn_on and turn_off, and the signal tick. The attributes record the devices, those that are Active, and the current Elected leader. The type of Elected is a set because the system can be in an undesired state where no devices claim to be the leader; in this case, an election takes place. The type Device has a single component, id, that records the identifier of a device. The operations of LE SoS take as an argument the device identifier and model the activation and deactivation of the corresponding device. The signal tick models the discrete passage of time.

As required by our guidelines, all the operations are defined using the state-machine

Figure 3.6: Abstract model: state machine diagram


std [State Machine] LE SoS STM [LE SoS STM]
tick[(card Elected = 1 and card Active = 0]/
[frame wr Elected, post card Elected = 0]
Atomic State
tunrOn [id not in set Active] / Active := Active union {id}
tunrOff [id in set Active] / Active := Active \{id}
tick[(card Elected = 0 or not (Elected subset Active)) and card Active > 0]/
[frame wr Elected, pre card Active > 0, post card Elected = 1 and Elected subset Active]

Source: Author's ownership.

diagram in Figure 3.6. It contains a single state with five transitions: the transition from the initial state, two represented implicitly inside the state, triggered by turn_on and turn_off, and two shown explicitly, both triggered by tick. The turn_on and turn_off transitions model the activation and deactivation of devices by adding and removing their identifiers from Active. These actions are specified using CML assignments.

There are two transitions triggered by the signal tick. The first is executed when there are active devices, but either there is no leader or the current leader is no longer active. It specifies by means of a specification statement that Elected must be updated to contain exactly one of the identifiers of the active devices. The second transition is executed if there is a leader but no active devices, and specifies that Elected must be emptied. The behavioural part of the explicit transitions are detailed in terms of CML pre and post-condition behaviours. The terms pre and post identify pre and post-conditions, respectively, that must be valid for a state declared in the frame statement. The card function returns the cardinality of a set. The subset function returns a boolean that yields true if the set on the left-hand side is a subset of the set on the right-hand side and false otherwise.

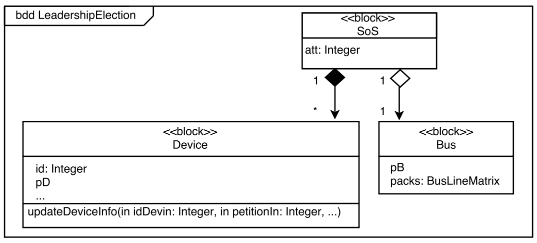
3.2.2.2 The Concrete SysML model

The concrete model depicts an implementation view of the system. The block diagram is that in Figure 2.3 (on page 26), where we have blocks for the devices and a bus for data transportation. Since there is no centralised control, each device maintains information about all others to decide which one is the leader. A device communicates with the others in cycles; in each cycle it receives information from the others and broadcasts its data. At the end of each cycle a device knows the state of all devices.

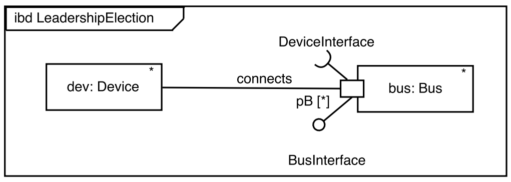
In spite of being much more complex than the abstract model, this model can still be described following our guidelines. The block definition and the internal block diagrams are depicted in Figures 2.3 and 2.4 (on page 26). Both satisfy the entity-definition and instance-level guidelines.

Figure 3.7 shows examples of diagrams that violate some entity-definition and instance-

Figure 3.7: Violation of guidelines of usage.



(a) Example of BDD that violates the guidelines.



(b) Example of IBD that violates the guidelines.

Source: Author's ownership.

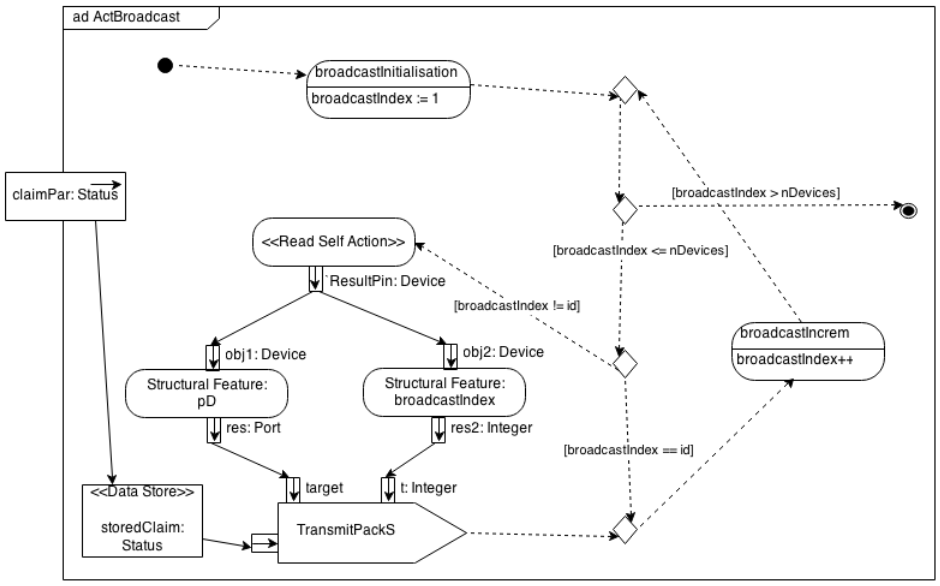
level guidelines. The block SoS in Figure 3.8a is the root block of a composition relation, therefore, it cannot have any operations and attributes (entity-definition guideline 4). In addition, there is an aggregation relationship between blocks SoS and Bus. Associations must be used instead of aggregations (entity-definition guideline 6). Figure 3.8b violates all instance-level guidelines. The block Bus is not part of a composition relation, therefore, it should not appear in the IBD of block SoS. The multiplicities are not compatible with the BDD, the cardinalities are not constants, they are described using *, and a port is connected to a block while it should be connected only to other ports.

Figure 2.5 (on page 27) depicts the state machine for a Device. It shows how a device decides if it is a leader. Each device has a variable (currentState) to define if it must broadcast data or receive a package from a specific device. For instance, if the variable currentState of device 2 has value 1, this device waits for data from the Bus regarding the device 1 or for a timeout, advances currentState to 2, and broadcasts its own data. When currentState is 3, it waits for a package from device 3, and repeats the whole process. At the end of each cycle (characterised by currentState == id), each device knows if it should be a leader. The undecided state is used when a device enters the network or when an election must happen, that is, there is no leader (for example, when the previous leader is turned off) or there are more than one leader (for example, when a follower loses communication with other devices and defines itself as a leader). The entry actions of the blocks Undefined, Leader and Follower are specified as CML assignments.

Transitions can fire activities. For example, both transitions from Undefined, to Follower and to Leader, fire the ActBroadcast activity. Figure 3.9 shows the diagram for this activity. It defines a loop that transmits data for each of the devices. The actions broadcastInitialisation and broadcastIncrem are opaque: their behaviour is not defined using diagrams. Instead, we use the CML-based data language following our guidelines.

Finally, we have sequence diagrams that specify scenarios. Figure 2.8 (on page 31) shows a possible initialisation of the system where three devices are turned on and they send their data in parallel to the network.

Figure 3.9: Example of an activity diagram.



Source: Author's ownership.

For didactic reasons, most of the diagrams presented here are simplified. The complete model is available in (LIMA, 2014).

3.3 Semantic rules metalanguage

In this section, we outline the metalanguage used in the presentation of the translation rules from SysML to CML. We decided to define this metalanguage instead of using one existent transformation language to avoid dependence with any implementation technology. Thus, our rules could be understood and implemented by anyone interested. For instance, Rule 3.1 takes a SysML type and outputs a CML type. The font differences emphasise the distinction between CML and the metalanguage. For instance, the if-then-else statements are part of the metalanguage and they do not appear in the generated CML models but the bold face **nat** between quotes appears in the CML model as the translation of the SysML type Nat. SysML elements are arguments of the translation functions, for instance, `t` is a SysML type.

Rule 3.1: t_types

```
t_types(t: Type): CML Type =
   if t = Nat then "nat"
   elseif t = Nat1 then "nat1"
   elseif t = Rat then "rat"
   elseif t = Int then "int"
   elseif t = Bool then "bool"
   elseif t = Real then "real"
   elseif t = Char then "char"
```

```
    elseif t = String then "seq of char"
    elseif t is a block name then "ID"
    elseif t = X[n] then "seq of " t_types(X)
    elseif t = set of X then "set of " t_types(X)
    elseif t is a datatype name then t
    elseif t is a valuetype name then t
    else "token"
    end if
```

The definition of the translation rules adopts the following conventions:

- Terms of the CML syntax are presented in the CML style, with a blue teletype font, bold face keywords and text between double quotes;
- The translation notation (metalanguage) is presented in black teletype font and the keywords are boldfaced (e.g., **if**, **then**, **for**, **begin**, **end**);
- Each translation rule takes as arguments SysML constructs in a textual notation and produces a CML construct;
- The type of SysML constructs (the domain of the semantic function) and the type of the CML constructs generated by the rule (the image of the semantic function) are specified in the function signature (e.g., `t_types(t:Type): CML Type` takes a SysML type as parameter and defines a CML type);
- Components of SysML constructs are accessed using a "." (e.g., `b.name` refers to the name of the SysML block denoted by `b`). Such components are defined in the SysML abstract syntax;
- **if-then-else** and **for** statements are used with their usual meaning;
- The function `name` takes a SysML object (e.g., state, transition, etc) and produces a unique name that identifies that object. When the uniqueness of a name is not relevant we access the name of the object using the `.name` notation;
- The function `id` is defined by Rule 3.2: it produces a sequence containing a single token composed by the unique name (as a string) of a SysML object. The unique name is obtained through the application of the function `name`. The *"mk_token"* is a constructor function of a CML token. The CML token type is similar to the string type.
- We assume that each diagram constructor can be uniquely identified by the field `index`, which is accessed using "." (e.g., `transition.index` for a transition in a state machine). Such field is important to differentiate two elements inside the same

diagram, because we may have the case where elements exist with same name or signature. For the purpose of translations we assume that this field is a natural number, however, implementers may decide to use any type that fits the requirement to uniquely identify an element of the same diagram. In this latter case, it is important to change the channel types where this field is used.

- The function `set2seq(s)` takes a set `s` and produces a sequence whose elements are those of `s`, placed into the sequence in an arbitrary order;
- Lists whose separators are symbols of the CML syntax are defined by the list constructor **`sep`** followed by the separator identifier. This constructor can be used in **`for`** commands where the generated elements at each iteration are intercalated with the separator (see Rule 3.3). Also, this constructor can be used before set definitions intercalating each element of the set. For example, **`sep`** `"." {"1", "2", "3"}` is the list `"1.2.3"`. The list built can be in any order as sets are not ordered.

Rule 3.2: id

```
id(x: Element): Identifier = "([mk_token(\""name(x)"\")])"
```

The formalisation of the list constructors is simple and omitted here. We further explain and illustrate its use as needed. All these operators of the metalanguage are identified by boxed symbols.

We call each of the equations that define our translation function a translation rule. This reflects the fact that these definitions can be used as rewrite rules to generate CML models (automatically).

3.4 Overview of the semantics of SysML models

We propose a denotational semantics for SysML: functions from the constructs of the SysML metamodel to constructs of the CML abstract syntax. These functions are described by translation rules that take well-defined elements of SysML and output well-formed CML components. Table 3.1 lists the main semantic functions and the elements to which they apply. Since a state machine or activity is defined in the context of a block, the semantic functions for state machines and activities take a block as a second argument.

The domains of the translation functions are sets of SysML constructs and, as far as possible, we have adhered to the metamodel of SysML presented in the standards for both UML 2.0 and SysML. We do not adhere to the metamodel in situations where the necessary component is not directly available. For instance, we assume a block has a component `stm` which may contain a reference to the state machine associated with the block.

Table 3.1: Main semantic functions for SysML models.

Semantic function	**SysML element**
`t_model`	Model
`t_simple_block`	Simple block
`t_composite_block_process`	Composite block
`t_port`	Port
`t_type_operation`	Operation
`t_type_signal`	Signal
`t_statemachine`	State machine and Block
`t_activity_diagrams`	Activity and Block
`t_sequencediagram`	Sequence diagram

Source: Author's ownership.

As explained in Section 3.1, we consider SysML models that contain a number of connected blocks and associated behavioural diagrams defined in accordance with our guidelines of usage. We also assume that the model has one root block. Under these assumptions, the semantics of a model is defined by the semantics of the root block. The semantics of a model is given by the `t_model` translation function in Rule 3.3. Next, we describe this rule and provide an example to illustrate its application.

Rule 3.3: t_model

```
t_model(m: SysML model): program =
   "types
      ID = seq of token
      DL = bool | <defer>"

   for each dt in m.AllDataTypes do
      if dt is enumeration then
         dt.name" = " for each a in dt.ownedLiteral sep "|" do
            "<"a.name">" end for
      else
         dt.name" :: "
         for each a in dt.Attributes do
            a.name": "t_types(a.type)
         end for
      end if
   end for

   for each op in m.AllOperations do
      t_type_operation(op)
   end for
```

```
for each s in m.AllSignals do
   t_type_signal(s)
end for

if m.AllOperations.size() > 0 then
   "OPS = " for each op in m.AllOperations sep "|" do
      op.name"_I | "op.name"_O"
   end for
   else "OPS = compose NullOperation of $id: token end"
end if

if m.AllSignals.size() > 0 then
   "S = " for each s in m.AllSignals sep "|" do
      s.name
   end for
else "S = compose NullSignal of $id: token end"
end if

"MSG = OPS | S"
"E = nat*ID*ID*MSG"

for each i in m.AllInterfaces do
   t_interface_types(i)
end for

declare_bag()
declare_aux_functions()
define_stm_channels(m)

for each b in m.AllBlocks do
   if b.isSimple
   then t_simple_block(b)
   else t_composite_block_process(b)
   end if
end for

t_sequencediagram(m.AllInteractions)
```

Rule 3.3 first declares the types `ID` and `DL` as a sequence of elements of type **`token`** and a union type composed by the set of booleans and the quote type `<defer>`. The type `DL` is necessary for encoding a state machine, and is the same for all state machines and, therefore, is defined globally. It encodes the possible outcomes of processing an event, which can be deemed consumed (**`true`**), not consumed (**`false`**) or deferred (`<defer>`). The event is consumed if it can be treated by a state machine or activity, when it cannot be treated at the point it is declared not consumed, and, sometimes, a state machine can defer events to be treated as soon as possible, which is the latter case. This type and its associated function are used in Chapter 5.

Following these initial types, we declare any datatype defined in the model. In case it is an enumeration, we simply declare the type with its possible values, otherwise, we declare the datatype as record type with its attributes and respective types.

Next, for each operation in the model, it applies the rule `t_type_operation`, which produces a record type declaration that encodes the operation. Similarly to operations, for each signal in the model, a record type is defined by the application of a translation rule. Next all the operation types are gathered in the union type `OPS`, all the signal types in the type `S`, and both types are joined to form the type of all messages `MSG`.

For all interfaces defined in the model, values for each interface are generated by the rule `t_inteface_types` where the operations and signals identifiers of each interface are defined in sets (see Rule 4.3). Afterwards, function `declare_bag` (Rule A.1 on Appendix A) is invoked to define the type bag, which is used for storing operation call requests of a block. Some auxiliary CML functions like checking if a identifier is a prefix of another are defined in `declare_aux_functions()` and channels used internally in the model are defined in function `define_stm_channels()`. Both functions are described in Appendix A.

Finally, for each block and sequence diagram in the model, the process that defines them is declared. The result of applying this rule to the concrete leadership election example discussed in Section 3.2.2.2, whose diagrams are illustrated in Section 2.1, is shown below.

```
types
   ID = seq of token
   DL = bool | <defer>
   Status = <Leader> | <Follower> | <Undecided>
   public updateDeviceInfo_I ::
      $id: token
      idDevin: int
      petitionIn: int
   public updateDeviceInfo_O ::
      $id: token
   public turnOff ::
      $id: token
      id: int
   public turnOn ::
      $id: token
      id: int
   public tick ::
      $id: token
   ...
   OPS = updateDeviceInfo_I | updateDeviceInfo_O | turnOn_I | ...
   S = compose NullSignal of $id: token end"
   MSG = OPS | S
   ...
functions
   DL_or(a: DL, b: DL) c: DL
   post (is_bool(a) and is_bool(b) => c = a or b)
      and (not is_bool(a) or not is_bool(b) => (
      ((a = true) or (b = true) => (c = true))
      and ((not (a = true) and not (b = true)) => c = <defer>)))
...
channels
      Device_op: nat*ID*ID*OPS
      Device_sig: nat*ID*ID*S
...
process Device = ...
...
process Bus = ...
...
process SoS = ...
...
process sd_LeadershipElection = ...
...
```

The semantics of the whole model is given by the process `SoS`, which corresponds to the root block **SoS**. Larger definitions and auxiliary declarations covered in the following chapters are omitted.

The rule `t_types` shown in the previous section takes simple SysML types (e.g., `Int`) and outputs equivalent CML types. A block name used as a type corresponds to the set of instances of that block; it is translated to the type `ID`, which identifies instances of blocks. A type with multiplicity (i.e., `X[n]`) is translated into a sequence of the CML types associated with the component type (i.e., `X`). Finally, any other types are associated with the generic CML type **`token`**, which supports only equality comparison. The remaining rules used in `t_model` are defined and explained in the next chapter and in Appendix A.

Table 3.2 summarises the correspondence between elements of a SysML model and elements of CML. A SysML element that exhibits some form of behaviour, namely, blocks, ports, activities and state machines, are modelled by CML processes; sequence diagrams are also represented by a CML process; connectors, which specify communication links, are modelled by channels; static elements (that is, without intrinsic behaviour), namely operation and signal signatures, are modelled by record types, and interfaces, which are collections of static elements, are modelled by sets of tokens. Operation signatures are considered static because they specify the message that is sent to blocks, not the behaviour of the operation itself, which is specified by a state machine or activity diagram.

Table 3.2: SysML-CML correspondence

SysML element	CML element
Simple block	Process
Composite block	Process
Activity	Process
State Machine	Process
Sequence Diagram	Process
Port	Process
Connector	Channel
Interface	Set of tokens
Operation call	Record Type
Signal	Record Type
Event	Communication

Source: Author's ownership.

3.5 Final remarks

In this chapter, we have presented our SysML guidelines of usage that allow us to formalise the semantics of SysML models, the conventions adopted in the presentation of the translation rules, and the rule that is the root of our translation: `t_model`. Additionally, we have presented the rule for the translation of types, which is used throughout the next chapters.

The semantics specified by these translation rules and the ones that follow provide a view of the system where the number of instances of blocks as well as the communication structure is fixed. In order to support dynamic creation and destruction of blocks, and reconfiguration of the dynamic structure, we require SysML patterns that support the specification of these behaviours. To the best of our knowledge, such patterns do not yet exist.

Although the guidelines may restrict the way of modelling systems, we believe they provide enough mechanism to reduce the impact on expressiveness. In addition, other styles of modelling can be adjusted to the requirements of our guidelines. The required effort will depend on how different they are. Also, our guidelines can act as templates for novice designers that are learning on how to model systems. Moreover, changes can be incorporated in the future according to specific needs, however, such changes can have impact on the rules, which may be updated accordingly. We plan to implement them in the future using a constraint language, like OCL (WARMER; KLEPPE, 2003).

4 Structural Diagrams

In this chapter, we present the semantics of blocks specified by means of block definition diagrams and internal block diagrams. The work reported in this chapter was developed in collaboration with Alvaro Miyazawa, as originally reported in (MIYAZAWA et al., 2013). Our translation is based on the underlying SysML model as described by the meta-model of SysML (OMG, 2012). In SysML the diagrams are seen as potentially disjoint views of a cohesive model. For instance, the same block may appear in two different diagrams, showing one operation in one diagram, and a different operation on another diagram. The diagrams are two different views of the same block, each emphasising a different operation. The block in the underlying model contains both operations.

One of the challenges is to provide a compositional semantic, such that the different elements can be combined without any dependence on their definitions. To achieve that, we partition the SysML concepts as much as possible in order to create a compositional way of defining the model elements through the combination of their different parts. The main CML constructors used to combine the different model elements are the parallelism operators. In this strategy, SysML model elements are defined in terms of CML processes or CML actions that are composed in parallel when needed to be assembled.

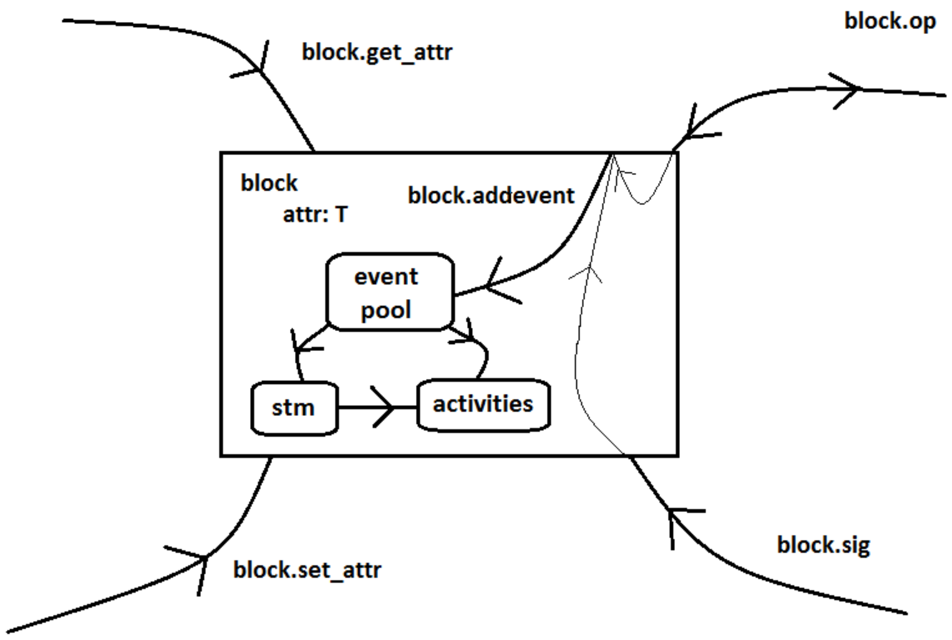
We exemplify our semantics for blocks through the example in Section 3.2. In particular, in this chapter we use the diagrams shown in figures 2.3, 2.4 and 3.5. They clearly respect the guidelines described in Section 3.1.

Section 4.1 presents an overview of the semantics of blocks, Sections 4.2 and 4.3 present the translation rules for operations, signals and interfaces. Section 4.4 presents the rules that define the semantics of ports. Finally, Sections 4.5 and 4.6 define the semantics of simple blocks and composite blocks.

4.1 Overview

The translation of SysML blocks takes into account a number of other elements: attributes, signals, operations, interfaces, ports, association, composition, connectors, parts, state machines and activities. The semantics of a block is defined in terms of the elements that form the block. Figure 4.1 illustrates the possible paths of communication with a block. Operations and signals can be requested, attributes can be read and updated, events related to operation and signals are stored in the event pool and can be sent for treatment (by a state machine or activity, for instance) and state machines can invoke activities in order to treat some of these events. A more detailed version of this figure is explained in Chapter 7 to illustrate the interactions between different diagrams.

Figure 4.1: Communication channels for blocks.



Source: Author's ownership.

Simple blocks may contain attributes and operations as well as ports. A port may respect a number of interfaces, which are provided or required by the block. An interface on the other hand may contain a number of signals and operations. For example, Figure 2.4 on page 26 shows the block Device that has a port pD that offers services described in the interface DeviceInterface and requires other services described in the interface BusInterface. Finally, simple blocks may have a state machine diagram and a number of activity diagrams, some of which may be associated with operations.

The semantics of a simple block is given by a process that is constructed using the processes that model its components. These are put together through parallel composition, and the alphabets (channel sets) of the processes determine how they communicate. The translation rule for a simple block declares all the necessary components: types, channels and processes.

The translation of a simple block produces three processes. The first is a simple model of the signature of the block (i.e., its operations and attributes) without taking into consideration generalisation. The second is the so called bare model of the block that combines the encoding of the signature of the process and the bare model of any parent block. Generalisation is modelled by interleaving. The third process combines the bare model of the block with its ports and state machines, if any exist.

Composite blocks may contain other blocks (by composition) and ports. As for simple blocks, the ports may contain interfaces, which may have operations and signals. Additionally, the different blocks that form the composite block may be linked by connectors directly or through ports.

Similarly to the case of simple blocks, the semantics of a composite block is a process that composes in parallel its constituent elements, that is, its parts and ports. The connectors are modelled by channels (see Rule 3.3 on page 55), which are used to rename the channels of the parts and link them. In this case, the translation rule also declares the necessary components (i.e., types, channels and processes), except for the processes that model the parts, as these have already been declared by the Rule 3.3.

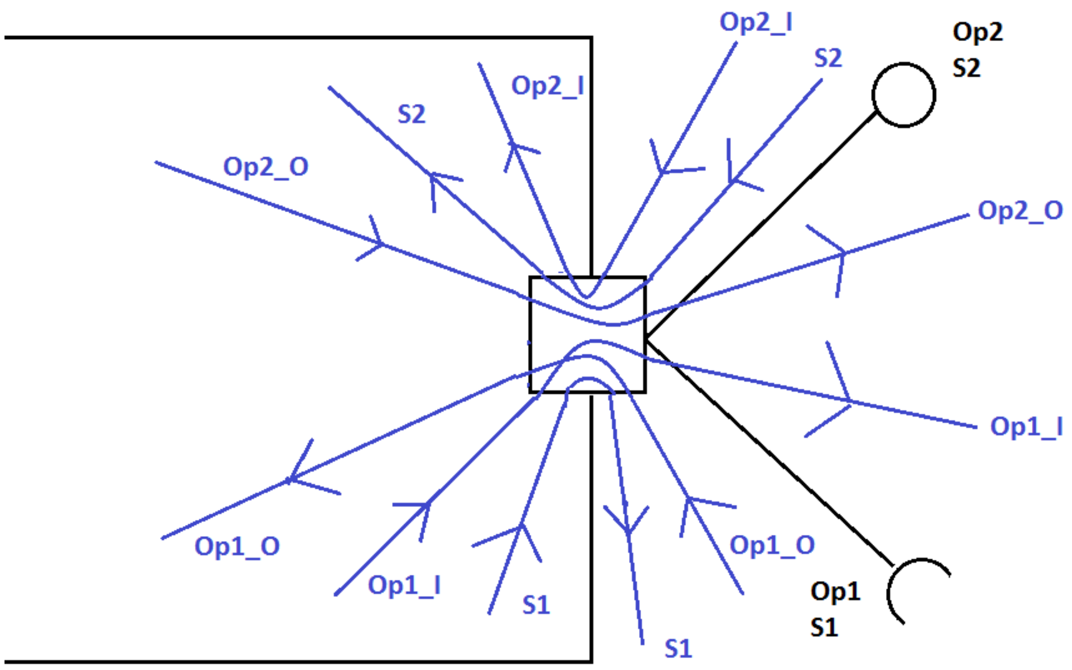
The process that models the core of a block (its operations and signals) receives signals and operation calls, and sends operation responses. Every time an operation call occurs, the simple block process registers all the possible responses because it delegates the decision of which is the appropriate response to its internal behaviour (or its parts) process, which synchronises such a response with the simple block process restricting the one that will be made available to the caller of the operation. The possible responses are recorded in a bag (i.e., a set with repetition), whose type is declared for each block as a bag of the block operations.

Ports are modelled by processes similar to those that model simple blocks, but since ports only define communications restricted by required and provided interfaces, they are not composed of any other processes. Interfaces are simply a form of collecting operations and signals, and, as such, are modelled as records defined as CML values that provide the appropriate scope. As explained in Chapter 2, a CML value binds an expression to an identifier. A port simply relays requests and responses to the appropriate element of the model (i.e, the block state machine, a part of a composite block or another port). The nature of the interfaces (provided or required) determine which communications can occur between the port, the block and its parts, and the external environment.

Figure 4.2 shows the patterns of communication between the external and internal environments through a port. The port has a provided interface containing the operation **Op2** and the signal **S2**, and a required interface containing the operation **Op1** and the signal **S1**. The input of **Op2** can only go inwards, and the output can only go outwards. The input of **Op1** can only go outwards, and the output can only go inwards. The signals **S1** and **S2** can only go, respectively, outwards and inwards.

Similarly, state machines (specified by state machine diagrams) and activities (specified

Figure 4.2: Allowed communication patterns for a port.



Source: Author's ownership.

by activity diagrams) yield processes and are the subject of sections 5.1 and 5.2 in Chapter 5.

4.2 Operations and signals

Each operation in the model produces two record types that encode the input and the output values of the operation, and each signal produces a single record type. These types are declared by the translation rules 4.1 and 4.2.

Rule 4.1: t_type_signal

```
t_type_signal(s: Signal): record type declaration =
"public "s.name" ::"
   "$id: token"
   for p: s.params do
      p.name ": " t_types(p.types)
   end for
```

The record type associated with a signal is named after the signal. Every signal and operation record type has an implicit parameter `$id` of type **token**, whose value is the exact name of the signal or the operation. This parameter is used to match the name of an operation or a signal of a block (see Rule 4.21). For each parameter of the signal, a record component of the appropriate type is declared. The result of applying this rule to the abstract leadership election example on Figure 3.5 (page 49) is as follows.

```
        public tick ::
                $id: token
```

Note that tick in Figure 3.5 has no parameters. Therefore the only parameter generated for this record type is `$id`, which is produced by default for all signals.

Rule 4.2: t_type_operation

```
t_type_operation(op: Operation): seq of record type declaration =
   "public "op.name"_I ::"
      "$id: token"
      for p: (op.input_params union op.input_output_params) do
         p.name " : " t_types(p.type)
      end for

   "public "op.name"_O ::"
      "$id: token"
      for p: (op.output_params union op.input_output_params) do
         p.name " : " t_types(p.type)
      end for
      if op.return != nil then
         "$ret: " t_types(op.return.type)
      end if
```

An operation in SysML can have three types of parameters: **in**, **out** and **inout**. The first and second are the input and output parameters. The third are parameters that are used for input and output. An operation yields two record types, both named after the operation (using `name`), but the first suffixed by `_I` indicating that this type encodes the input parameters of the operation, and the second suffixed by `_O` indicating that this type encodes the output parameters. The definition of the records is similar to that of the records that encode signals, but in the first record only the input parameters are generated; these include parameters classified as **in** and **inout**, that is, input and input-output parameters. The second record includes all output parameters (those classified as **inout** and **out**) as well as the return value, if the operation has one. The return parameter is always named `$ret`. The result of applying this rule to the operation **turnOn** in the model of Figure 3.5 is as follows.

```
        public turnOn_I :: $id: token
                           id: int
        public turnOn_O :: $id: token
```

The operation **turnOn** does not have a return parameter. The parameter `$id` corresponds to the internal parameter used by our semantics, while the parameter `id` is the only input parameter of the operation.

4.3 Interfaces

An interface generates CML values, which can be used along the CML specification. In rule 4.3, these values declare sets of tokens `name(i)"_I"`, `name(i)"_O"`, `name(i)"_OPS"` and `name(i)"_S"` of all (input and output) operations and signals contained in the interface. The translation rule for interfaces is shown below.

Rule 4.3: t_interface_types

```
t_interface_types(i: Interface): value declaration =
    "values"
    t_interface_signal_type(i)
    t_interface_input_type(i)
    t_interface_output_type(i)
    name(i)"_OPS = "name(i)"_I union "name(i)"_O"
```

It declares CML values named after the interface (`name(i)`). There are four value declarations: three for operations and one for signals. As operations are described by names that represent their input values and their output values, similarly, the operations of the interface are defined using two names, one for the operations input `name(i)"_I"`, and another for the operations output `name(i)"_O"`. The signals are gathered in a value `name(i)"_S"`. These values contain only the names used by the operations (input and output) and signals declared in the interface, just like a signature. These values are used by the processes of ports to restrict the operations and signals that can be transmited through the ports. If there are no operations or signals, the appropriate types (`I` and `O`, or `S`) are declared as empty types. Finally, the input and output operation types are grouped in the `name(i)"_OPS"` type.

The input and output and the signal types are declared by Rules 4.4, 4.5 and 4.6. The following rule defines the function that declares the type `S` of all the signals in the interface.

Rule 4.4: t_interface_signal_type

```
t_interface_signal_type(i: Interface): seq of values paragraph =
    name(i)"_S = {"
    sep "," for each s: i.Signals do
        "mk_token(\""s.name"\")"
    end for
    "}"
```

The next rule declares the input type of all the operations in the interface.

Rule 4.5: t_interface_input_type

```
t_interface_input_type(i: Interface): seq of values paragraph =
   name(i)"_I = {"
   sep "," for each op: i.Operations do
      "mk_token(\""op.name"_I\")"
   end for
   "}"
```

The next rule similarly declares an output type for all the operations.

Rule 4.6: t_interface_output_type

```
t_interface_output_type(i: Interface): seq of values paragraph =
   name(i)"_O = {"
   sep "," for each op: i.Operations do
      "mk_token(\""op.name"_O\")"
   end for
   "}"
```

Despite not being present in Figure 2.4 on page 26, the interface BusInterface declares a signal transmitPack. The value associated with it is shown below.

```
values
BusInterface_S = {mk_token("transmitPack")}
BusInterface_I = {}
BusInterface_O = {}
BusInterface_OPS = BusInterface_I union BusInterface_O
end
```

4.4 Standard ports

A standard port is a mechanism for communication between blocks without a specific reference to a block instance. We model a standard port similarly to a simple block, except that it does not treat the received request: it simply relays it to the appropriate block.

For the purpose of our semantics, we distinguish two sides of a port: the external and the internal. If the port belongs to a simple block, then the internal side is connected implicitly to the block itself. If the port belongs to a composite block, it must be explicitly connected to a port of one of the parts of the block.

The set of provided and required interfaces connected to a port determine the possible patterns of communication of a port. In the most general case, a port has a set of provided interfaces (PI) and a set of required interfaces (RI). For instance, for the port pD in Figure 2.4 on page 26, we have the following sets:

$$RI = \{BusInterface\}$$
$$PI = \{DeviceInterface\}$$

These sets of interfaces are determined with respect to the external side of the port and are implicitly mirrored on the internal side. A provided interface is mirrored in a required internal interface on the same port, and a required interface is mirrored in a provided internal interface on the same port.

In our CML model, a port gives rise to four channels of communication (two for operations and two for signals) associated with its sides: internal and external. Signals are asynchronous communications, whereas operations are considered synchronous in our semantics. The patterns of communication on these channels depend on the nature of the requests. Operations in a provided interface of a port can only be requested on the external operation channel because the entity that requests is external to the block. On the other hand, operations in a required interface can only be requested on the internal channel because the request comes from inside the block.

The types associated with a port are those of its interfaces. They are grouped according to the type of the interface (provided or required), and whether the type refers to a signal or an operation. Rule 4.7 declares the CML values for types associated with a port and it is used in the definition of a port.

Rule 4.7: t_port_types

```
t_port_types(p: Port): class declaration =
   "values"
      name(p)"_P_I = " if p.PI.size() > 0 then sep "union"{name(i)"_I" |
          i in set p.PI} else "{}" end if
      name(p)"_P_O = " if p.PI.size() > 0 then sep "union"{name(i)"_O" |
          i in set p.PI} else "{}" end if
      name(p)"_P_S = " if p.PI.size() > 0 then sep "union"{name(i)"_S" |
          i in set p.PI} else "{}" end if
      name(p)"_R_I = " if p.RI.size() > 0 then sep "union"{name(i)"_I" |
          i in set p.RI} else "{}" end if
      name(p)"_R_O = " if p.RI.size() > 0 then sep "union"{name(i)"_O" |
          i in set p.RI} else "{}" end if
      name(p)"_R_S = " if p.RI.size() > 0 then sep "union"{name(i)"_S" |
          i in set p.RI} else "{}" end if
```

The provided operations types (`name(p)"_P_I"` and `name(p)"_P_O"`) and the provided signals type (`name(p)"_P_S"`) are built from the operations and the signals of the provided interface (e.g, `BusInterface_I` or `BusInterface_O`). Similarly, the required operation and signal types (`name(p) "_R_I"`, `name(p) "_R_O"` and `name(p) "_R_S"`) are built from the required interfaces. The set of all operations (both provided and required) is named `OPS` and the set of all signals is named `S`.

The application of `t_port_types` to port pB is as follows (Figure 2.4 on page 26).

```
values
        pB_P_I = BusInterface_I
        pB_P_O = BusInterface_O
        pB_P_S = BusInterface_S
        pB_R_I = DeviceInterface_I
        pB_R_O = DeviceInterface_O
        pB_R_S = DeviceInterface_S
```

The four channels used by a port are declared by the following rule.

Rule 4.8: t_port_channels

```
t_port_channels(p: Port): channel declaration =
   "channels"
     name(p) "_int_sig: nat*ID*ID*S"
     name(p) "_ext_sig: nat*ID*ID*S"
     name(p) "_int_op: nat*ID*ID*OPS"
     name(p) "_ext_op: nat*ID*ID*OPS"
```

The first two channels, `name(p) "_int_sig"` and `name(p) "_ext_sig"`, communicate signals in and out of the port; they communicate a natural number identifying a particular request (necessary to distinguish requests made by the same block), the identifiers of the source and the target of the send signal action, and the encoding of the signal as an instance of the appropriate record type. Similarly for operations, the channels `name(p) "_int_op"` and `name(p) "_ext_op"` communicate a natural number identifying the instance of the operation call, the identifiers of the source and the target of the call, and the operation call itself, in the form of an input or output type. Each operation call involves two communications on the channel, the first with an input value and the second with the output value.

Using as example the same port pB (Figure 2.4 on page 26) the resulting translation of the application of Rule 4.8 is as follows.

```
channels
        pB_int_sig: nat*ID*ID*S
        pB_ext_sig: nat*ID*ID*S
        pB_int_op: nat*ID*ID*OPS
        pB_ext_op: nat*ID*ID*OPS
```

The process that models a particular port is declared by the Rule 4.9; it is used in the definition of both simple and composite blocks.

Rule 4.9: t_port

```
t_port(p: Port): seq of paragraph=
    t_port_types(p)
    t_port_channels(p)
    "process port_"name(p)
    " = $id: ID @ begin"
    "@ mu X @ (("
        name(p)"_ext_sig?i?o!$id?x:(x.$id in set "name(p)"_P_S) ->"
            name(p)"_int_sig.i.$id?y.x -> Skip"
        "[]"
        name(p)"_int_sig?i?o!$id?x:(x.$id in set "name(p)"_R_S) ->"
            name(p)"_ext_sig.i.$id?y.x -> Skip"
        "[]"
        name(p)"_ext_op?i?o!$id?x:(x.$id in set "name(p)"_P_I) ->"
            name(p)"_int_op.i.$id?y.x -> Skip"
        "[]"
        name(p)"_int_op?i?o!$id?x:(x.$id in set "name(p)"_P_O) ->"
            name(p)"_ext_op.i.$id?y.x -> Skip"
        "[]"
        name(p)"_int_op?i?o!$id?x:(x.$id in set "name(p)"_R_I) ->"
            name(p)"_ext_op.i.$id?y.x -> Skip"
        "[]"
        name(p)"_ext_op?i?o!$id?x:(x.$id in set "name(p)"_R_O) ->"
            name(p)"_int_op.i.$id?y.x -> Skip"
    "); X)"
"end"
```

The main action of the process, which is named after the port, defines that it receives signal events of the provided interfaces on the external channel, and forwards them through the internal channel. Conversely, signal events of the required interfaces are received only on the internal channel and forwarded on the external channel.

The relaying of an operation call is slightly more complicated because the relaying of inputs and outputs is carried out separately. Input records of operation calls of the provided interface are only received through the external channel and relayed through the internal channel. Output records of operation calls of the provided interface are received through the internal channel and forwarded through the external channel. Input and output records of operation calls of the required interface are received and sent in a complementary way. Inputs are received in the internal channel (and forwarded on the external channel), and outputs are received in the external channel (and forwarded on the internal channel).

The application of `t_port` to port pB is as follows (Figure 2.4 on page 26). We use ellipsis in the begining to hide the content already displayed in the previous two translation function examples.

```
...
process port_pB = $id: ID @ begin
@ mu X @ ((
        pB_ext_sig?i?o!$id?x:(x.$id in set pB_P_S) ->
        pB_int_sig.i.$id?y.x -> Skip
        []
        pB_int_sig?i?o!$id?x:(x.$id in set pB_R_S) ->
        pB_ext_sig.i.$id?y.x -> Skip
        []
        pB_ext_op?i?o!$id?x:(x.$id in set pB_P_I) ->
        pB_int_op.i.$id?y.x -> Skip
        []
        pB_int_op?i?o!$id?x:(x.$id in set pB_P_O) ->
        pB_ext_op.i.$id?y.x -> Skip
        []
        pB_int_op?i?o!$id?x:(x.$id in set pB_R_I) ->
        pB_ext_op.i.$id?y.x -> Skip
        []
        pB_ext_op?i?o!$id?x:(x.$id in set pB_R_O) ->
        pB_int_op.i.$id?y.x -> Skip
        ); X)
end
```

4.5 Simple blocks

The next set of rules applies to simple blocks, that is, blocks that are not composed of other blocks. Whilst simple blocks cannot be the source of a composition relation, they can be part of another block (target of a composition relation) and can be associated with other blocks through generalisation and association. These aspects are dealt with by the translation rule for composite blocks.

The root translation rule is `t_simple_block`. It takes a simple SysML block and produces a number of definitions: types, channels and processes. It produces these definitions by calling other translation rules, namely, `t_block_types`, `t_block_channels`, `t_port` and `t_port_process`.

Rule 4.10: t_simple_block

```
t_simple_block(b: Block): seq of program paragraph =
   t_block_types(b)
   t_block_channels(b)
   for each p in set b.Ports do
      t_port(p)
   end for
   t_block_process(b)
```

Rule 4.10 simply invokes rule `t_block_types` to define the types needed for manipulating the block operation and signals, then it calls rule `t_block_channels` to define the channels of the block. Next, for each port of the block, rule `t_port` is called to define the process of the port and, finally, rule `t_block_process` defines the CML process of the block.

In the following subsections, we use blocks Device of Figure 2.3 on page 26 and block LE SoS of Figure 3.5 on page 49 to present the application of the rules `t_block_*` of Rule 4.10.

4.5.1 Simple blocks: types

Similarly to the rule for interface, `t_block_types` declares values for the block's signals and operations. The value `name(b) "_I"` is the union type of the input values of the operations in the block, the type `name(b) "_O"` is the union of the output values of the operations in the block and the type `name(b) "_S"` is the union of the types associated with the signals in the block. Finally, Rule 4.11 declares a value `name(b) "_OPS"` as the union of all the values associated with the operations in the block, and specifies a bag type and its operation types.

Rule 4.11: t_block_types

```
t_block_types(b:Block): value declaration =
   "values"
      t_block_values(b)
      t_block_signal_type(b)
      t_block_input_type(b)
      t_block_output_type(b)
      name(b)"_OPS = "name(b)"_I union "name(b)"_O"
```

The rule `t_block_values` declares constant values defined by the block. The next rules `t_block_signal_type`, `t_block_input_type` and `t_block_output_type` are similar to the rules for interfaces, except that they take into account the generalisation relation. The full specification of these rules can be found in Appendix A.

In case the block does not contain operations, the types `_I` and `_O` are declared as empty sets. This assures that it is always possible to recover the values `_S`, `_I` and `_O` from a parent, even when they are empty at the children.

The application of `t_block_types` to the block **LE SoS** (Figure 3.5 on page 49) is as follows.

```
...
values
  LE_SoS_S = {mk_token("tick")}
  LE_SoS_I = {mk_token("turnOn_I") , mk_token("turnOff_I")}
  LE_SoS_O = {mk_token("turnOn_O") , mk_token("turnOff_O")}
  LE_SoS_OPS = LE_SoS_I union LE_SoS_O
```

This translation shows the definitions of the block types of block LE SoS regarding its operations and signals. These values are sets of tokens that identify the signal records and the operation records of the block.

4.5.2 Simple blocks: channels

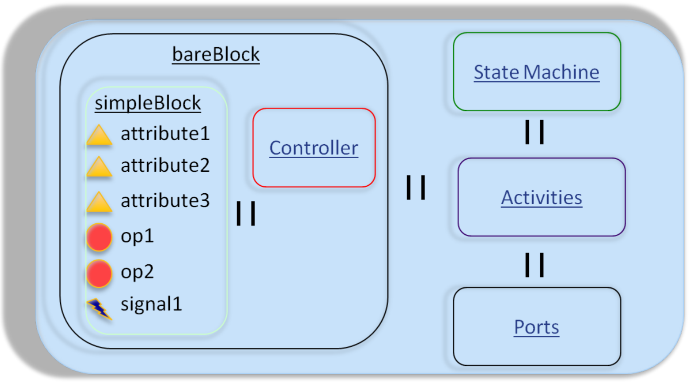
All channels are prefixed with the name of the block followed by the underscore character, for instance, `Device_op` and `Device_sig`, for the block Device. However, for simplification, when we refer to these channels in a general manner we omit this prefix, therefore, instead of `Device_op` we say `op`. The channels associated with a block are the `op` channel, the `sig` channel, the `addevent` channel, and getter and setter channels for each attribute of the block. The `op` and `sig` channels are used for communication between the environment and the block (passing the identifier of the particular request, the source and target of the request as well as operation or signal requests), while the `addevent` channel is used to communicate received requests to the event pool of the block. The names of the getter and the setter channels reflect the names of the block and its attribute and are used to provide concurrent access to the attributes by the processes that model the block.

Rule 4.12: t_block_channels

```
t_block_channels(b: Block): channel declaration =
   "channels"
      name(b)"_op: nat*ID*ID*OPS"
      name(b)"_sig: nat*ID*ID*S"
      name(b)"_addevent: nat*ID*ID*MSG"
   for a: b.Attributes do
      name(b)"_get_"a.name":ID*ID*"t_types(a.type)
      name(b)"_set_"a.name":ID*ID*"t_types(a.type)
   end for
```

The application of `t_block_channels` to the block Device gives the following declaration.

Figure 4.3: Processes of a simple block.



Source: Author's ownership.

```
        channels
          Device_op: nat*ID*ID*OPS
          Device_sig: nat*ID*ID*S
          Device_addevent: nat*ID*ID*MSG
          Device_get_id: ID*ID*int
          Device_set_id: ID*ID*int
          ...
```

4.5.3 Simple blocks: processes

Figure 4.3 shows the processes involved in the definition of a simple block. The bareBlock process composes in parallel the simpleBlock and the Controller processes. The former describes the access to attributes and the reception and replies of operation calls and signals, while the latter stores the events of operation calls and signals to be treated. The bareBlock process is then composed in parallel with the StateMachine process, which defines the behaviour of the state machine of the block, the Activities process, which composes in parallel all the activities of the block, and, finally, the Ports process, which provides the events related to all ports of the block.

Rule 4.13 for blocks uses five other translation rules: `t_block_simple_process` to produce the simple model of the block, `t_block_bare_process` to produce the bare model of the block, `t_activity_diagrams` and `t_activities_chanset` to generate the model of activity diagrams and the set of external events associated with the block's activities, and `t_statemachine` to generate the model of a state machine.

If a state machine is present, its process (*"stm_"*`name(b.stm)`) is combined in parallel with the bare model of the block (*"bare_"*`name(b)`) to form the process named after the block. If there are any activity diagrams, the process that models them (`name(b)` *"_ads"*) is composed in parallel with the block (and the state machine) model. The synchronisation set is determined by the allowed external communications of the block's activities. If there are any ports in the block, then they are composed in interleaving (processes `"port_"name(p)`), which are, in turn, composed in parallel with the block (as well as activities and state machine). The interaction patterns between the different processes are determined by the synchronisation sets of their parallel compositions.

The parallelism between a bare model and the model of a state machine involves the channels `inevent` and `consumed`, the events associated with the channel `op` where the second parameter is the identifier of the block, and the fourth parameter is a value of an output operation type, and the channels for manipulating attributes of the block. The `inevent` and `consumed` channels are used to treat an event in the state machine and communicate that it has been consumed, respectively. The output `op` channel is used to allow the state machine to respond to an operation call. When the request for an operation arrives at the block through an input operation type, the reply to that operation is given through an output operation type, which in this case is given by the state machine that is treating the event. The remaining channels of this alphabet of synchronisation are used by the state machine for accessing any attributes of the block when needed.

The parallel composition of block and state machine is then composed in parallel with the processes of activities of the block in function `t_block_activity_parallelism` (Rule 4.14). This resultant process is again composed in parallel with the processes related to the ports of the block by function `t_block_ports_parallelism` (Rule 4.15). Finally, we hide the access to attributes and internal calls according to function `t_block_hiding_access` (Rule 4.16).

Rule 4.13: t_block_process

```
t_block_process(b: Block): seq of process declaration =
   t_block_simple_process(b)
   t_block_bare_process(b)
   if b.activities.size() > 0
   then t_activity_diagrams(set2seq(b.activities),b)
   end if

   if b.stm <> NULL then t_statemachine(b.stm) end if

   "process "name(b)" = $id: ID @ (
      ("
      if b.activities.size() > 0 then
         "(("
      else
         "("
      end if
            if b.stm == NULL then
               "("
                  "bare_"name(b)"($id)"
            else
               "(("
                        "bare_"name(b)"($id)"
                           "[|{|"name(b)"_inevent, "name(b)"_consumed|}"
                              "union {|"name(b)"_op.n.x.$id.y | n: nat, x: ID,
                              y: OPS @ y.$id in set "name(b)"_O |}"
                              for a in set b.AllAttributes do
                                 "union {|"name(b)"_get_"a.name".x.$id |
                                 x: ID @ prefix($id^[mk_token(\"stms\"),
                                 mk_token(\"stm\")],x)|}"
                                 "union {|"name(b)"_set_"a.name".x.$id |
                                 x: ID @ prefix($id^[mk_token(\"stms\"),
                                 mk_token(\"stm\")],x)|}"
                              end for
                           "|]"
                           "stm_"name(b.stm)"($id^[mk_token(\"stms\"),
                           mk_token(\"stm\")])"
                     ")\\({|"name(b)"_inevent, "name(b)"_consumed|}"
                        for a in set b.AllAttributes do
                           "union {|"name(b)"_get_"a.name".x.$id |
                           x: ID @ prefix($id^[mk_token(\"stms\"),
                           mk_token(\"stm\")],x)|}"
                           "union {|"name(b)"_set_"a.name".x.$id |
                           x: ID @ prefix($id^[mk_token(\"stms\"),
                           mk_token(\"stm\")],x)|}"
                        end for
                     ")"
            end if
            ")"
         t_block_activity_parallelism(b)
         t_block_ports_parallelism(b)
   ")" t_block_hiding_access(b)
```

Rule 4.14 defines the parallelism between the block and state machine process with the activity process of the block. The alphabet of synchronisation of the parallelism between the bare model plus state machine and the activities process is defined by the function `t_activities_chanset`, which is shown in Rule A.4 of Appendix A. It has the channels `hasevent` and `getevent`, which checks if an event is available in the block's event pool and removes such an event from the pool, respectively. The channels `startActivity_`, `endActivity_` and `interruptActivity_`, which are used for firing a certain activity, communicating its termination and interrupting it, respectively, are also part of the synchronisation alphabet added by channels that invoke operations and signals of blocks, and, finally, channels to manipulate block attributes.

Rule 4.14: t_block_activity_parallelism

```
t_block_activity_parallelism(b: Block): seq of process declaration =
   if b.activities.size() > 0 then
      "[|"t_activities_chanset(b.activities,b)"|]"
      "("name(b)"_ads($id^[mk_token(\"acts\")]))"
      ")\\({|"name(b)"_hasevent,"name(b)"_getevent|})"
   end if
   ")"
```

Rule 4.15 defines the parallelism between the block model (bare model, state machine and activities) and the ports. The synchronisation set of this parallelism contains all the events associated with the channel `op` where the second and third parameters are the identifier of the block and the identifier of one of the ports (in any order). Since the model of a port does not use the same channel `op` as the block, its internal channel `int_op` is renamed to match the blocks `op` channel. As the communications between the block and the process are internal, we hide the events on the `op` and `sig` channels between the block and the ports from the external environment of the block.

Rule 4.15: t_block_ports_parallelism

```
t_block_ports_parallelism(b: Block): seq of process declaration =
      if b.Ports.size() > 0 and (
         exists p in set b.Ports @
            p.allProvidedAndRequiredOperations().size() > 0
            or p.allProvidedAndRequiredSignals().size() > 0)
      then
         "[|"
         if b.allProvidedAndRequiredOperations().size() > 0
         then "{|" name(b)"_op.n.$id.p |"
                "n: nat, p in set {"sep ","{"$id^"id(x) | x in set
                   b.Ports}"}|}"
              "union"
              "{|"name(b)"_op.n.p.$id |"
                "n: nat, p in set {"sep ","{"$id^"id(x) | x in set
                   b.Ports}"}|}"
         end if
         if b.allProvidedAndRequiredOperations().size() > 0 and
            b.allProvidedAndRequiredSignals().size() > 0
         then "union"
         end if
         if b.allProvidedAndRequiredSignals().size() > 0
         then "{|" name(b)"_sig.n.$id.p |"
                "n: nat, p in set {"sep ","{"$id^"id(x) | x in set
                   b.Ports}"}|}"
              "union"
              "{|"name(b)"_sig.n.p.$id |"
                "n: nat, p in set {"sep ","{"$id^"id(x) | x in set
                   b.Ports}"}|}"
         end if
         "|]"
         "(" sep "|||" {
            "(port_"name(p)"($id^"id(p)"))[["name(p)"_int_op <-
                "name(b)"_op]]"
            | p in set b.Ports}
         ")"
      ")"
         if b.allOperations.size() > 0 or b.allSignals.size() > 0 then
              "\\ ("
            if b.allOperations.size() > 0
            then "{|" name(b)"_op.n.$id.p |"
                   "n: nat, p in set {" sep "," {"$id^"id(x) | x in set
                      b.Ports}"}|}"
                "union"
                "{|" name(b)"_op.n.p.$id |"
                   "n: nat, p in set {" sep "," {"$id^"id(x) | x in set
                      b.Ports}"}|}"
            end if

            if b.allOperations.size() > 0 and b.allSignals.size() > 0
            then "union"
            end if

            if b.allSignals.size() > 0
            then "{|" name(b)"_sig.n.$id.p |  "
                   "n: nat, p in set {" sep "," {"$id^"id(x) | x in set
                      b.Ports}"}|}"
                "union"
                "{|" name(b)"_sig.n.p.$id |  "
                   "n: nat, p in set {" sep "," {"$id^"id(x) | x in set
                      b.Ports}"}|}"
            end if
            ")"
         end if
      else ")"
      end if
```

Finally, Rule 4.16 hides the access to attributes, operations and signals hiding the `_get_`, `_set_`, `_op` and `_sig` channels, respectively. If the attribute, operation or signal has public visibility (`visibility = #public`), then we only hide internal access to the related events, for example, a state machine updating the value of one attribute. The function `prefix($id,x)` verifies if the block identifier (`$id`) is a prefix of the id `x`, which means that x is internal to the block (we recall that the type ID represents a sequence of tokens that stores the identification of an element considering its hierarchy). Otherwise, when the attribute is private we hide all events of the attribute channels.

Rule 4.16: t_block_hiding_access

```
t_block_hiding_access(b: Block): seq of process declaration =
    " \\ ("
      for a in set b.AllAttributes sep "union" do
         if a.visibility = #public then
            "{|"name(b)"_get_"a.name".x.y | x: ID, y: ID @ prefix($id,x)
               and prefix($id,y)|}"
            "union {|"name(b)"_set_"a.name".x.y | x: ID, y: ID @
               prefix($id,x) and prefix($id,y)|}"
         else
            "{|"name(b)"_get_"a.name"|}"
            "union {|"name(b)"_set_"a.name"|}"
         end if
      end for
      for op in set b.operations sep "union" do
         if a.visibility = #public then
            "{|"name(b)"_op.x.y.z | x: ID, y: ID, z in set "b.name"_I  @ "
            "prefix($id,x) and prefix($id,y) and z.$id = "op.name"|}"
            "union {|"name(b)"_op.x.y.z | x: ID, y: ID, z in set
               "b.name"_O  @ "
            "(prefix($id,x) or prefix($id,y)) and z.$id = "op.name"|}"
         else
            "{|"name(b)"_op.n.x.y.z | n: nat, x: ID, y: ID, z in set
               "b.name"_I  @ z.$id = "op.name"|}"
            "union {|"name(b)"_op.n.x.y.z | n: nat, x: ID, y: ID, z in
               set "b.name"_O  @ z.$id = "op.name"|}"
         end if
      end for
      for sig in set b.signals sep "union" do
         if a.visibility = #public then
            "{|"name(b)"_sig.n.x.y.z | n: nat, x: ID, y: ID, z in set
               "b.name"_S  @ "
            "(prefix($id,x) or prefix($id,y)) and z.$id = "sig.name"|}"
         else
            "{|"name(b)"_sig.n.x.y.z | n: nat, x: ID, y: ID, z in set
               "b.name"_S  @ z.$id = "sig.name"|}"
         end if
      end for
   ")"
```

The application of t_block_process to the block Device results in the following specification.

```
process simple_Device = ...
process bare_Device = ...
...
process Device = $id: ID @ (
(((
bare_Device($id)
[|{|Device_inevent, Device_consumed|} union
  {|Device_op.n.x.$id.y | n: nat, x: ID,
        y: OPS @ y.$id in set Device_O|} union
  {|Device_get_id.x.$id | x: ID @
        prefix($id^[mk_token("stms"), mk_token("stm")],x|} union
  {|Device_set_id.x.$id | x: ID @ prefix($id^[mk_token("stms"),
        mk_token("stm")],x|}
|]
stm_Device(id^[mk_token("stms"),mk_token("stm")]))
)\\ ({|Device_inevent, Device_consumed|} ...)
)[|...|] (Device_ads($id^[mk_token("acts")]))
\\ ({|Device_hasevent, Device_getevent|})
) [|...|] (port_pD($id^[mk_token("pD")])
        [[pD_int_op <- Device_op]]))
\\ ({|Device_op.n.$id.p | n: nat, p in set
        {$id^[mk_token("pD")]} |} union
   {|Device_op.n.p.$id | n: nat, p in set
        {$id^[mk_token("pD")]} |})...
)))
\\ ({|Device_get_id|} union {|Device_set_id|})
)
```

Rule 4.17 for simple processes uses three other translation rules: `t_block_state` produces the state of the simple process, `t_block_state_action` produces the action that accesses the state variables, and `t_block_requests_action` produces the action that receives requests to the operations and signals of the block.

Rule 4.17: t_block_simple_process

```
t_block_simple_process(b: Block): process declaration =
   "process simple_"name(b) " = $id: ID @ begin"
      t_block_state(b)
      "actions"
      t_block_state_action(b)
      t_block_requests_action(b)
      "@"
      name(b)"_state"
      "[|||{"sep ","{a.name| a in set
         b.Attributes}"}|{"name(b)"_enabled}|||]"
      name(b)"_requests"
   "end"
```

The state access and the operation control actions are combined in interleaving with a partition of the state that gives control of the components associated with the block's attributes to the access action, and the component `enabled` to the requests action. An example of the application of Rule 4.17 to block Device is shown as follows.

```
process simple_Device = $id: ID @ begin
...
actions
...
@ Device_state [{id,pD}||{enabled}] Device_requests
end
```

Rule 4.18 for bare processes checks whether the block has a parent or not, and, if it does, composes it in interleaving with the bare process of its parent. The process of the bare block is the parallel composition of its simple process with the controller process, which corresponds to the event pool manager of the block. Every request for operation or signal reception is stored in the controller process to be treated. The simple block communicates events to the controller through the channel `addevent`. Details on the controller process are provided in Section 4.5.4.

Rule 4.18: t_block_bare_process

```
t_block_bare_process(b: Block): process declaration =
   t_controller(b)
   if b.parent == NULL
   then "process bare_"name(b)" = $id: ID @
      (simple_"name(b)"($id)[|{|"name(b)"_addevent|}|]
         controller_"name(b)"($id))\\{|"name(b)"_addevent|}"
   else
      "process bare_"name(b)" = $id: ID @"
         "((simple_"name(b)"($id)[|{|"name(b)"_addevent|}|]
            controller_"name(b)"($id))\\{|"name(b)"_addevent|})"
         "|||"
         "(bare_"name(b.parent)"($id))"
            "[["name(b.parent)"_inevent <- "name(b)"_inevent ,"
              name(b.parent)"_hasevent <- "name(b)"_hasevent ,"
              name(b.parent)"_getevent <- "name(b)"_getevent ,"
              name(b.parent)"_op <- "name(b)"_op ,"
              name(b.parent)"_sig <- "name(b)"_sig"
              for a in set b.parent.attributes do
               ","name(b.parent)"_get_"a.name" <- "name(b)"_get_"a.name","
               name(b.parent)"_set_"a.name" <- "name(b)"_set_"a.name
              end for
```

```
                "]]"
    end if
```

If the block does not have a parent, its bare process is solely the simple process composed in parallel with the controller process, which is the case for the block Device as can be seen in the application of Rule 4.18 in the following specification.

```
process bare_Device = $id: ID @ (simple_Device($id)
[|{|Device_addevent|}|] controller_Device($id))
\\{|Device_addevent|}
```

The state of the simple process is declared by the rule `t_block_state`. For each attribute of the block, a state component is declared with the appropriate name and type. Furthermore, a bag of output types (associated with the operations of the block) is declared to record which outputs are enabled. This component is initialised with the empty bag. We use a bag in this context because we need to store more than one occurrence of an event, but we do not need them to be stored in any particular order because the choice on the event to be treated is non-deterministic; this is why we use a bag instead of a sequence or set. The proviso in this rule assures that the default value of an attribute is defined in terms of a CML expression, otherwise we would generate an error in the specification because the expression could not be evaluated.

Rule 4.19: t_block_state

```
t_block_state(b: Block): state declaration =
    "state"
    for a: b.Attributes do
        a.name": "t_types(a.type) if a.default.size() > 0 then " :=
            "a.default.get(0) end if
    end for
    name(b)"_enabled: Bag := empty_bag"


provided
    1. if a.default.get(0) exists, it is a CML expression
```

This rule is applied to the block Device to declare the state components of the process `simple_Device` as follows.

```
        process simple_Device = $id: ID @ begin
        state
                id: int;
                ... // other variables
                Device_enabled: Bag := empty_bag;

        actions
        ...
        @ Device_state [{id,...}||{Device_enabled}] Device_requests
        end
```

The rule `t_block_state_action` declares a recursive (mu X) action that offers to read and write the block attributes using the channels `get_` and `set_`. If the visibility of the attribute is private, then the identifier of the source must be a prefix of the identifier of the block, that is, only the internal elements of the block can access the attributes.

Rule 4.20: t_block_state_action

```
t_block_state_action(b: Block) =
   name(b)"_state = mu X @ ("
   sep "[]" for a: b.Attributes do
      if a.visibility = #public then
         name(b)"_get_"a.name"?o!$id!"a.name" -> X"
         "[]"
         name(b)"_set_"a.name"?o!$id?x -> "a.name" := x; X"
      else
         name(b)"_get_"a.name"?o:(prefix($id,o))!$id!"a.name" -> X"
         "[]"
         name(b)"_set_"a.name"?o:(prefix($id,o))!$id?x -> "a.name" := x;
             X"
      end if
   end for
   ")"
```

This rule contributes to the definitions of the actions of the process `simple_Device`.

```
        process simple_Device = $id: ID @ begin
        state
                id: int;
                ... // other attributes
                Device_enabled: Bag := empty_bag;

        actions
```

```
         Device_state = mu X @ (
                 Device_get_id?o!$id!id -> X
                 []
                 Device_set_id?o!$id?x -> id := x; X
                 ...
         )

         ...
         @ Device_state [{id,...}||{Device_enabled}] Device_requests
         end
```

The values `?o!$id!id` in the channel `Device_get_id` refers to an identifier of the source that is requesting access to the `id` attribute (`?o`), the target identifier that is the block identifier itself (the parameter `!$id` of the process) and the communication of the state attribute id (`!id`), respectively. The channel `Device_set_id` works in a similar way. It only differs the last data, which is an input (`?x`) for the new value of the attribute id.

Similarly to Rule 4.20, `t_block_requests_action` declares a recursive action that waits for a request to start and end an operation on the op channel. If it receives a request to start an operation, it forwards the request through the `addevent` channel, adds the returned token id to the bag `enabled` and recurses. If the request is to end an operation that is in the bag `enabled`, it will remove the item that was added to the bag when the operation was first requested. Finally, if the request is for a signal, it simply sends the request through the `addevent` channel.

Rule 4.21: t_block_requests_action

```
t_block_requests_action(b: Block) =
   name(b)"_requests = mu X @ ("
      name(b)"_op?n?o!$id?x:(x.$id in set "name(b)"_I) -> ("
         if b.Operations.size() > 0 then
            sep "[]" for op: b.Operations do
               "[x.$id = mk_token(\""op.name"_I\")] & "
                  name(b)"_addevent!n!o!$id!x -> Skip;"
                  name(b)"_enabled := bunion("name(b)"_enabled,"
                  "{mk_token(\""op.name"_O\")\|\->1}); X"
            end for
         else "Stop"
         end if
      ")"
      "[]"
      name(b)"_op?n?o!$id?x:(in_bag(x.$id,"name(b)"_enabled)) -> ("
         if b.Operations.size() > 0 then
            sep "[]" for op: b.Operations do
               "[x.$id = mk_token(\""op.name"_O\")]&"
                  name(b)"_enabled :=
                       bdiff("name(b)"_enabled,{x.$id\|\->1}); X"
            end for
         else "Stop"
         end if
      ")"
      "[]"
      name(b)"_sig?n?o!$id?x:(x.$id in set "name(b)"_S) -> "
         name(b)"_addevent!n!o!$id!x -> X"
   ")"
```

In our example, the only process that communicates on the channel `addevent` is the controller, which is described in Section 4.5.4. The result of applying this rule to our Device example completes the definition of the process `simple_Device` as follows.

```
process simple_Device = $id: ID @ begin
state
id: int;
... // other attributes
Device_enabled: Bag := empty_bag;

actions

Device_state = mu X @ (
Device_get_id?o!$id!id -> X
  []
Device_set_id?o!$id?x -> id := x; X
...
)

Device_requests = mu X @ (
  Device_op?n?o!$id?x:(x.$id in set Device_I) -> (
  [x.$id = mk_token("turnOn_I")] &
  Device_addevent!n!o!$id!x -> Skip;
  Device_enabled := bunion(Device_enabled,{mk_token("turnOn_O")
                                           |->1}); X
    []
  [x.$id = mk_token("turnOff_I")] &
  Device_addevent!n!o!$id!x -> Skip;
  Device_enabled := bunion(Device_enabled,{mk_token("turnOff_O")
                                           |->1}); X
  )
    []
  Device_op?n?o!$id?x:(in_bag(x,Device_enabled)) -> (
  [x.$id = mk_token("turnOn_O")] &
  Device_enabled := bdiff(Device_enabled,{x.$id|->1}); X
    []
```

```
          [x.$id = mk_token("turnOff_O")] &
          Device_enabled := bdiff(Device_enabled,{x.$id|->1}); X
          )
            []
          Device_sig?n?o!$id?x:(x.$id in set Device_S) ->
          Device_addevent!n!o!$id!x -> X
          )
        @ Device_state [{id,...}||{Device_enabled}] Device_requests
        end
```

This process declares a state formed of the attribute **id** of the block **Device** (other attributes are omitted for simplification) and the set of enabled output `Device_enabled`. The main action of the process is the interleaving of two actions: `Device_state` and `Device_requests`. The former controls the state and allows communications that read and write to the state variables (e.g.,`id`), and the latter controls the receipt of operation and signal requests. In the interleaving, the state is partitioned between the two actions: the component corresponding to the block's attribute is assigned to the first action, and the component `Device_enabled` to the second action.

The action that controls the operation and signal requests, receives a value of an operation input type through the channel `Device_op` or a value of a signal type through the channel `Device_sig`. In the first case, it identifies which operation has been called, sends the call through the channel `Device_addevent`, and adds the answer to the bag `Device_enabled`, and recurses. In the case of a signal value, the action sends the signal through the channel `Device_addevent`, and recurses.

4.5.4 Simple blocks: controller process

As previously mentioned, the controller is responsible for managing the event pool and the queue of deferred events. The controller is modelled by a CML process produced by the function defined in Rule 4.22. First, it defines the channels used by the controller process to communicate with other entities. Channels `inevent` and `consumed` are used to communicate with the state machine, and channels `hasevent` and `getevent` are used to communicate with activities.

The process produced by this function uses a type *"controller_"*`name(b)`*"_E"*, which is defined right above the process, as the type that contains the index, source, target and the message (operation call or signal) of events that arrive in the block. Also, the process has two state components: `events` and `deferred`. The first is a set representing the event pool. According to the UML/SysML semantics, the order in which the events are treated is not defined, therefore, we use a set to store events so that the choice on the event is non-deterministic. The second state component is used to store deferred events from the state machine, which should be treated according to the order they are deferred, that is why we use a sequence instead. A state machine can defer events when given an active state and an event to be treated: such an event is in the list of deferred events of this active state. In this case, the event is stored in the list of deferred events to be treated as soon as the state machine is in a state that can consume such an event.

Rule 4.22: t_controller

```
t_controller(b: Block): process declaration =
   "channels"
      name(b)"_inevent:ID*E"
      name(b)"_consumed: ID*DL"
      name(b)"_hasevent: ID*token"
      name(b)"_getevent: ID*(E|<NOEVENT>)"
   "types"
      "controller_"name(b)"_E = nat*ID*ID*MSG inv mk_(-,-,-,m) == m.$id
         in set "name(b)"_I union "name(b)"_S"

   "process controller_"name(b)" = $id: ID @ begin"
   "state"
      "events: set of controller_"name(b)"_E := {}"
      "deferred: seq of controller_"name(b)"_E := []"
   "functions"
      "remove: (seq of controller_"name(b)"_E) * nat -> seq of
         controller_"name(b)"_E"
      "remove(s,i) == s(1,...,i-1)^s(i+1,...,len s)"
      "pre i <= (len s)"
   "actions"
   "TreatDeferredEvents = (dcl i: nat := 1 @ "
      "while (i <= len deferred) do"
         "let ev = deferred(i) in ("
            name(b)"_inevent?o!ev -> ("
               name(b)"_consumed!o?b:(is_bool(b)) -> "
                  "deferred := remove(deferred,i)"
               "[]"
               name(b)"_consumed.o.<defer> -> i := i+1"
            ")"
            "[] (" name(b)"_addevent?n?o!$id?e ->"
               "events := events union {mk_(n,o,$id,e)})"
         ")"
   ")"
   "@ mu X @ ("
      name(b)"_addevent?n?o!$id?e -> (events := events union
         {mk_(n,o,$id,e)});X"
      "[]"
      "[(card events) > 0] & ("
         "|~| ev in set events @ ("
            name(b)"_inevent?o!ev-> (events := events\{ev}; ("
               name(b)"_consumed.o.true -> TreatDeferredEvents; X"
```

```
                "[]"
                name(b)"_consumed.o.false -> X"
                "[]"
                name(b)"_consumed.o.<defer> -> deferred := deferred^[ev];X"
             "))"
          ")"
       ")"
       "[]"
       name(b)"_hasevent?o?t -> ("
          "if (exists e in set events @ e.#4.$id = t)"
             "then (|~| e in set {x | x in set events @ x.#4.$id = t} @"
                "events := events\{e}; "name(b)"_getevent!o!e -> X)"
          "elseif (exists i in set inds deferred @ (deferred(i)).#4.$id =
             t) "
             "then (|~| i in set {x | x in set inds deferred @
                (deferred(x)).#4.$id = t} @"
                name(b)"_getevent!o!(deferred(i)) -> deferred :=
                   remove(deferred,i); X)"
          "else "name(b)"_getevent!o!<NOEVENT> -> X"
       ")"
    ")"
    "end"
```

The process declares the function `remove` that takes a sequence and an index `i` (of the sequence) and returns a new sequence identical to the input sequence except that the element in the i-th position is removed. Additionally, an action called `TreatDeferredEvents` is declared; this action traverses the list `deferred`, and sends it to the state machine's internal process; if the event is consumed it is removed from the list of deferred events, otherwise nothing changes. The recursive action inside the while statement guarantees that whenever an event is ready to be sent to the internal process, the block can also add an event to the the pool of events.

Next, the main action is a recursive action that allows the block to add new events to the pool, or offers a random event in the event pool `events`. Three possibilities can be observed at this point. If the events are successfully consumed, a change in the state may occur, and the deferred events may become active, therefore the process attempts to communicate them for treatment. This is accomplished by calling the action `TreatDeferredEvents`. If the event was consumed but not successfully (e.g. the event is sent to the state machine but it cannot be treated or deferred by any active states), no state change has occurred, thus the deferred events cannot become active. The last case that need to be treated is when the event is deferred by the state machine. In this case, the event is simply added to the end of the sequence `deferred`.

Finally, the channel `hasevent` is offered to activities in order to check if a specific event is available in the event pool: in case it is, then it is removed from the pool `events` and returned to the activity by the channel `getevent`. The event is also searched in the pool of deferred events, and just when it is not available anywhere, the `getevent` returns the value `<NOEVENT>`. The controller for our example is as follows.

```
channels
  Device_inevent:ID*E
  Device_consumed:ID*DL
  Device_hasevent:ID*token
  Device_getevent:ID*(E | <NOEVENT>)
types
  controller_Device_E = nat*ID*ID*MSG
  inv mk_(-,-,-,m) == m.$id in set Device_I union Device_S
process controller_Device = $id: ID @ begin
state
  events: seq of controller_Device_E
  deferred: seq of controller_Device_E
functions
  remove: (seq of E) * nat -> seq of E
  remove(s,i) == s(1,...,i-1)^s(i+1,...,len s)
  pre i <= (len s)
actions
  TreatDeferredEvents = (dcl i: nat := 1 @
    while (i <= len deferred) do
      let ev = deferred(i) in (
        Device_inevent?o!ev -> (
          Device_consumed!o?b:(is_bool(b)) ->
            deferred := remove(deferred,i)
          []
          Device_consumed.o.<defer> -> i := i+1
        )
        []
        mu X @ (Device_addevent?n?o!$id?e ->
         events := events union {mk_(n,o,$id,e)}; X)
     )
  @ mu X @ (
    Device_addevent?n?o!$id?e -> (events := events union
      {mk_(n,o,$id,e)}); X
      []
    [(card events) > 0] & (
      |~| ev in set events @ (
        Device_inevent?o!ev -> (events := events\{ev};(
          Device_consumed.o.true -> TreatDeferredEvents;X
          []
          Device_consumed.o.false -> X
          []
          Device_consumed.o.<defer> -> deferred :=
            deferred^[ev];X
          ))
      )
    )
    []
    Device_hasevent?o?t -> (
        if (exists e in set events @ e.#4.$id = t)
        then (|~| e in set {x | x in set events @ x.#4.$id = t} @
          events := events\{e}; Device_getevent!o!e -> X)
        elseif (exists i in set inds deferred @ (deferred(i)).#4.$id = t)
        then (|~| i in set {x | x in set inds deferred @
          (deferred(x)).#4.$id = t} @
          Device_getevent!o!(deferred(i)) ->
          deferred := remove(deferred,i);X)
        else Device_getevent!o!<NOEVENT> -> X
    )
    )
end
```

The details of how the state machine process and activity process communicates with the events described in the controller process are discussed in sections 5.1 and 5.2 in Chapter 5.

4.6 Composite blocks

Composite blocks are those that are formed of other blocks through the composition relation. For example, the model of the leadership election problem on Figure 2.3 on page 26 has a composite block, namely SoS.

Our guidelines require that a composite block does not have behaviours or attributes. Therefore we must only treat ports in the composite system boundary. Notice that whilst we do not allow composite blocks with behaviours, it is still possible to add a new part to the composite block, move all behaviours to the new block, and connect that part to any port or part. So, there is a simple way to overcome this restriction without any loss of expressiveness.

A composite block is modelled by the alphabetised parallel composition of the processes of its parts, including the outer ports (i.e., the ports of the composite state). The alphabets are defined by the connectors from the part and its ports. The process that models a part is that derived from the block that types the part. The external channels of the ports of the parts must be renamed to the names of its connectors (see Rule 4.23).

Rule 4.23: t_composite_block_process

```
t_composite_block_process(b: Block): process declaration =
   for p in set b.Ports
      t_port(p)
   end for
   "process "name(b)" = $id: ID @ "
      define_alphabetised_parallel(
         ({
            ("("name(p.Type)"($id^"id(p)"))"
               if p.Connectors.size() > 0 or p.PortsWithConnectors.size()
                  > 0
               then "[[" t_rename_ext_ports(p) "]]"
               end if
            ,
            t_chanset_part(p))
            | p in set b.Parts
         } union {
            ("(port_"name(p.topleveldefinition)"(
                  $id^"id(p.topleveldefinition)"))"
               if p.Connectors.size() > 0 or p.PortsWithConnectors.size()
                  > 0
               then "[[" t_rename_int_ports(p.topleveldefinition) "]]"
               end if
            ,
            t_chanset_port(p.topleveldefinition))
            | p in set b.Ports
         })
      )
   "\\ ("
      for sb in set {p.Type | p in set b.Parts} do
         if sb.visibility = #private then
            "{|"name(sb)"_op,"name(sb)"_sig|}"
            for a in set sb.AllAttributes sep "union" do
            if a.visibility = #public then
            "{|"name(sb)"_get_"a.name".x.y | x: ID, y: ID @ prefix($id,x)
               and prefix($id,y)|}"
            "union {|"name(sb)"_set_"a.name".x.y | x: ID, y: ID @
               prefix($id,x) and prefix($id,y)|}"
            end if
            end for
         end if
      end for
   ")"
where
   1. topleveldefinition of a port p is the port of the block of which p
      is an instance.
```

The parallel action is defined by the rule, `define_alphabetised_parallel`, that takes a sequence of pairs formed of an action name and a channel set name, and recursively constructs the alphabetised parallelism of all actions (see Rule 4.24). The notation `pairs(1)` means an access to the first element of the sequence `pairs`. The numbers after the dot in `pairs(1).1` and `pairs(1).2` are projections on the pair `(Name*Chanset)`, where `1` is an access to the value of type `Name` and `2` is an access to the value of type `Chanset` (channel set).

Rule 4.24: define_alphabetised_parallel

```
define_alphabetised_parallel(pairs: seq of (Name*Chanset)): action =
   if len pairs = 0 then "Skip"
   elseif len pairs = 1 then pairs(1).1
   else
      "("pairs(1).1
      "["pairs(1).2"||" define_union_chansets(tl pairs)"]"
      define_alphabetised_parallel(tl pairs)
      ")"
   end if
```

Since the alphabetised parallelism is a binary operator, we need to compose a number of alphabetised parallel operators hierarchically. To do so, we calculate the channel set associated with an alphabetised parallelism. This channel set is the union of the channel sets in the parallel operator. To calculate the channel set associated with the alphabetised parallelism formed of a sequence of action names and channel set names, we define a recursive rule that produces the union of all channel set names in the sequence of pairs of action and channel set names (Rule 4.25).

Rule 4.25: define_union_chansets

```
define_union_chansets(pairs: seq (Name*Chanset)): binary expression =
   if len pairs = 0 then"{||}"
   elseif len pairs = 1 then pairs(1).2
   else pairs(1).2 "union" define_union_chansets(tl pairs)
   end if
```

The translation rule that generates the renaming pairs for a part's ports is shown by Rule 4.26; for each port in the part, it obtains the set of connectors associated with the port and renames the port's external channel with the channels named after the connectors.

Rule 4.26: t_rename_ext_ports

```
t_rename_ext_ports(p: Part): seq of renaming pair =
   sep "," for each c in p.Connectors do
      name(p.Type)"_sig <- "name(c)"_sig,"
      name(p.Type)"_op <- "name(c)"_op"
```

```
    end for
    if p.Connectors.size() > 0 and (exists x in p.Ports |
       x.Connectors.size() > 0)
    then ","
    end if
    sep "," for each x in p.PortsWithConnector do
       sep "," for each c in x.Connectors do
          name(x.topleveldefinition)"_ext_sig <- "name(c)"_sig,"
          name(x.topleveldefinition)"_ext_op <- "name(c)"_op"
       end for
    end for


where
    1. topleveldefinition of a port p is the port in the block definition
       of which p is an instance.
```

The pairs generated by this rule when applied to the first instance of Device of our example in Figure 2.4 is shown below.

```
pD1_ext_sig <- c_pB[1]_pD1_sig, pD1_ext_op <- c_pB[1]_pD1_op
```

Since a port may have multiple connectors, the external channel may be renamed multiple times. This means that a communication on the external channel is substituted by a choice of communication over the renamed channels. We assume that each connector is uniquely identified, therefore, there is no possibility of naming conflicts.

The rule for renaming the internal channels of a port is similar, but simpler. It only considers its own channels, as a port does not contain other ports.

Rule 4.27: t_rename_int_ports

```
t_rename_int_ports(p: Port): seq of renaming pair =
    sep "," for each c in p.Connectors do
       name(p)"_int_sig <- " name(c)"_sig,"
       name(p)"_int_op <- " name(c)"_op"
    end for
```

For each connector associated with a port, the internal channels `int_sig` and `int_op` are renamed to the channels (operation and signal channels) associated with the connector. For example, the renaming pairs for the port **pD** of the first instance of Device in our examples are as follows.

```
pD1_int_sig <- c_pB[1]_pD1_sig, pD1_int_op <- c_pB[1]_pD1_op
```

The result of applying the Rule 4.23 to the block **SoS** for resultant model of figures 2.3 and 2.4 is the following process.

```
process SoS = $id: ID @ (
Device($id^[mk_token("Device1")])[[
pD1_ext_sig <- c_pB[1]_pD1_sig, pD1_ext_op <- c_pB[1]_pD1_op
]]
[ {|...|} || {|...|} ]
(Device($id^[mk_token("Device2")])[[
pD2_ext_sig <- c_pB[2]_pD2_sig, pD2_ext_op <- c_pB[2]_pD2_op
]]
[ {|...|} || {|...|} ]
(Device($id^[mk_token("Device3")])[[
pD3_ext_sig <- c_pB[3]_pD3_sig, pD3_ext_op <- c_pB[3]_pD3_op
]]
[ {|...|} || {|...|} ]
(Bus($id^[mk_token("Bus")])[[
pB[1]_ext_sig <- c_pB[1]_pD1_sig, pB[1]_ext_op <- c_pB[1]_pD1_op,
pB[2]_ext_sig <- c_pB[2]_pD2_sig, pB[2]_ext_op <- c_pB[2]_pD2_op,
pB[3]_ext_sig <- c_pB[3]_pD3_sig, pB[3]_ext_op <- c_pB[3]_pD3_op
]]
))))
```

Rule 4.28 determines the set of allowed events for a part; it takes a part (that is, an instance of a block that is part of a composite block), and builds the channel set that defines the possible interactions of the part with its environment. This set is built based on the connectors of the part or of its ports. The ends of a connector (`c.Ends`) may be parts or ports, and to identify the appropriate channel, we use the name of the block that types the part, or the name of the port.

Rule 4.28: t_chanset_part

```
t_chanset_part(p: Part): chanset =
   "{|" name(p.Type)"_op.n.($id^"id(p)").y | n: nat, y: ID |}"
   "union {|"name(p.Type)"_sig.n.($id^"id(p)").y | n: nat, y: ID |}"
   "union {|" name(p.Type)"_op.n.y.($id^"id(p)") | n: nat, y: ID |}"
   "union {|"name(p.Type)"_sig.n.y.($id^"id(p)") | n: nat, y: ID |}"
   for each c: p.Connectors do
      let x in set c.Ends such that x <> p in
         "union {|"name(c)"_op.n.($id^"id(p)").($id^"id(x)") | n: nat|}"
         "union {|"name(c)"_sig.n.($id^"id(p)").($id^"id(x)") | n: nat|}"
         "union {|"name(c)"_op.n.($id^"id(x)").($id^"id(p)") | n: nat|}"
         "union {|"name(c)"_sig.n.($id^"id(x)").($id^"id(p)") | n: nat|}"
   end for
   for each x: p.Ports do
      "union " t_chanset_port(x)
   end for
```

Rule 4.28 takes a part and an identifier, and produces a channel set that specifies the set of events on which the part can synchronise. This set includes:

- all the events associated with the channels `op` and `sig` of the part are in the channel set. These events allow anyone to communicate with the block;
- for each connector linked to the part, the events associated with the channels `op` and `sig` of the other end of the connector; the second and the third parameters (`"$id^"id(p)` and `"$id^"id(x)`) are the identifiers of both ends of the connector. Note that we include them in both orders (`id(p)` followed by `id(x)` and vice-versa);
- for each port of the part, the events associated with the channels `op` and `sig` of the connector are added; these events are restricted to those where the second and the third parameters are the identifiers of both ends of the connector.

This channel set does not restrict which elements can make requests to the part or its ports, nor does it restrict which elements can receive responses from the part or its ports. It does, however, restrict the elements to which the part and its ports can make requests. A similar restriction exists regarding which elements can respond to the part and its ports. The intersection of this channel set with the channel set associated with a connected element determines the allowed communication between both ends.

In our example, the channel set associated with the first part of `Device`, which has the identifier `$id^[mk_token("Device1")]` is shown below.

```
{|
  c_pB[1]_pD1_op.n.($id^[mk_token("Device1")]).($id^[mk_token("Bus")]),
  c_pB[1]_pD1_op.n.($id^[mk_token("Bus")]).($id^[mk_token("Device1")]),
  c_pB[1]_pD1_sig.n.($id^[mk_token("Device1")]).($id^[mk_token("Bus")]),
  c_pB[1]_pD1_sig.n.($id^[mk_token("Bus")]).($id^[mk_token("Device1")])
  | n: nat
|}
```

This channel set essentially allows the part Device1 to communicate on the channels associated with the connector between ports pD1 and pB[1], where the second and third parameters are the identifiers of the parts Bus and Device1 in both orders.

Similarly to the channel set of parts, the channel set of ports is calculated by Rule 4.29; it takes a port, and builds the channel set that defines the possible interactions of the port with its environment.

Rule 4.29: t_chanset_port

```
t_chanset_port(p:Port): chanset =
    "{|" name(p.topleveldefinition)"_ext_op.n.(
      $id^"id(p.topleveldefinition)").y | n: nat, y: ID |}"
    "union {|"name(p.topleveldefinition)"_ext_sig.n.(
      $id^"id(p.topleveldefinition)").y | n: nat, y: ID |}"
    "union {|" name(p.topleveldefinition)"_ext_op.n.y.(
      $id^"id(p.topleveldefinition)") | n: nat, y: ID |}"
    "union {|"name(p.topleveldefinition)"_ext_sig.n.y.(
      $id^"id(p.topleveldefinition)") | n: nat, y: ID |}"
    for each c: p.Connectors do
       let x in set c.Ends such that x <> p in
          "union {|"name(c)"_op.n.($id^"id(p.topleveldefinition)").
            ($id^"id(x.topleveldefinition)") | n: nat|}"
          "union {|"name(c)"_sig.n.($id^"id(p.topleveldefinition)").
            ($id^"id(x.topleveldefinition)") | n: nat|}"
          "union {|"name(c)"_op.n.($id^"id(x.topleveldefinition)").
            ($id^"id(p.topleveldefinition)") | n: nat|}"
          "union {|"name(c)"_sig.n.($id^"id(x.topleveldefinition)").
            ($id^"id(p.topleveldefinition)") | n: nat|}"
    end for

where
   1. topleveldefinition of a port p is the port of the block of which p
      is an instance.
```

The following extract illustrates the channel set of port pD of the first part of Device whose identifier is `$id^[mk_token("Device1"),mk_token("pD1")]`. This port is connected to the port pB[1] of block Bus and its identifier is `$id^[mk_token("Bus"),mk_token("pB[1]")]`.

```
{|pD1_ext_op.n.($id^[mk_token("Device1"),mk_token("pD1")]).y |
  n: nat, y: ID|} union
  pD1_ext_sig.n.($id^[mk_token("Device1"),mk_token("pD1")]).y |
  n: nat, y: ID|} union
  pD1_ext_op.n.y.($id^[mk_token("Device1"),mk_token("pD1")]) |
  n: nat, y: ID|} union
  pD1_ext_sig.n.y.($id^[mk_token("Device1"),mk_token("pD1")]) |
  n: nat, y: ID|} union
  {|c_pB[1]_pD1_op.n.($id^[mk_token("Device1"),mk_token("pD1")]).
  ($id^[mk_token("Bus"),mk_token("pB[1]")]) | n: nat |} union
  {|c_pB[1]_pD1_sig.n.($id^[mk_token("Device1"),mk_token("pD1")]).
  ($id^[mk_token("Bus"),mk_token("pB[1]")]) | n: nat |} union
  {|c_pB[1]_pD1_op.n.($id^[mk_token("Bus"),mk_token("pB[1]")]).
        ($id^[mk_token("Device1"),mk_token("pD1")]) | n: nat |} union
  {|c_pB[1]_pD1_sig.n.($id^[mk_token("Bus"),mk_token("pB[1]")]).
        ($id^[mk_token("Device1"),mk_token("pD1")]) | n: nat |}
```

4.7 Related Work

We have presented a behavioural model of SysML blocks that includes simple and composite blocks, generalisation, association and composition relations, standard ports and connectors, interfaces, operations, attributes and signals. To the best of our knowledge, this is the first formalisation of the semantics of a comprehensive subset of the block notation. In particular, it is also unknown to us any treatments of SysML blocks that take into account state machine diagrams and activity diagrams.

Graves proposes a representation of a restricted subset of SysML block diagrams in the web ontology language (OWL2), a language for knowledge representation based on a description logic (GRAVES, 2009). Description logics are subsets of first-order logic that possess better computational properties (e.g., some description logics are decidable and others have efficient inference procedures). Ding and Tang propose a representation of SysML block diagrams directly in a description logic (DING; TANG, 2010). In both cases, the semantics only cover simple blocks, composite blocks and associations.

Graves and Bijan extend the work by (GRAVES, 2009) by encoding SysML diagrams into a type theory that axiomatises block diagram notions of types, properties and operators (GRAVES; BIJAN, 2011b). Block definition diagrams and internal block diagrams are covered, but dynamic aspects of SysML block diagrams are not.

All these works focus on generating a set of axioms that specify a system based on a SysML diagram, and then using the existing techniques for the underlying logic to check properties such as consistency. Although Graves and Bijan describe model refinement as theory refinement (that is, modification of the knowledge base aiming at achieving consistency), it does not elaborate on the topic, and it is not clear what properties are preserved by this notion of refinement.

Table 4.1 presents a comparison of the coverage of our formalisation with some of the available literature. The first row indicates the related works, the first column contains the features that we cover: simple blocks (SB), composite blocks (CB), standard ports (SP), interfaces (Int), operations (Op), signals (Sig), properties (Prop), generalisation (Gen), association (Assoc), and dynamic aspects of block diagrams (Dyn). The symbol ✓ indicates that the feature is covered by the work on the column, and $\times$ indicates it is not.

Table 4.1: Coverage comparison.

	Graves (2009)	Ding and Tang (2010)	Graves and Bijan (2011b)	Our Work
SB	✓	✓	✓	✓
CB	✓	✓	✓	✓
SP	×	×	×	✓
Int	×	×	×	✓
Op	×	×	✓	✓
Sig	×	×	×	✓
Prop	×	×	✓	✓
Gen	×	×	✓	✓
Assoc	✓	✓	✓	✓
Dyn	×	×	×	✓

Source: Author's ownership.

5 Behavioural Diagrams

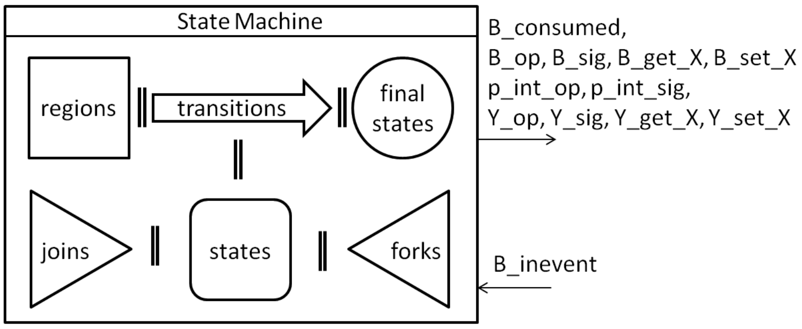
In this chapter we present the diagrams used to describe the behaviour of blocks. According to our guidelines in Chapter 3, the behaviour of blocks can be described in terms of a state machine, and activity diagrams can be used to model the dynamics of an operation or some specific behaviour of the block. The semantics of state machine diagrams are not a contribution of this thesis, however, as they are part of a whole strategy for defining an integrated SysML semantics in CML, its semantics is briefly described in Section 5.1 to make our presentation as self-contained as possible. For more details on the semantics of state machine diagrams see (MIYAZAWA et al., 2013). Next, Section 5.2 presents the activity diagram semantics in CML. Despite a sequence diagram also being considered an element to describe behaviour in SysML, they are used for specific purposes in our approach. Its semantics is presented in Chapter 6.

5.1 State Machine Diagram

The CML model of a state machine is defined by a single process whose actions model the elements of the state machine as shown in Figure 5.1. Regarding the channel names, `B` is the block that owns the state machine, `X` is an attribute of a block, `p` is a port and `Y` is another block that is associated with `B`. Each state, region, final state, transition (starting from a state), join and fork pseudostate is modelled by a CML action, and all these actions are composed in parallel to define the overall behaviour of the CML process.

As previously indicated in Section 3.4, the process that models a state machine is defined by the application of the function `t_statemachine` to the state machine and its block, which is presented in Rule 5.1.

Figure 5.1: Overview of the model of state machines.



Source: Author's ownership.

Rule 5.1: t_statemachine

```
t_statemachine(stm: StateMachine,b:Block):   process declaration =
   "chansets"
      "internal_chanset_"name(stm)" = {|enter,entered,exit,exited,"
         "enabled,fire,fired|}"
         "union {|"name(b)"_consumed.x | x: ID @ x<>"id(stm)"|}"
         "union {|"name(b)"_inevent.x | x: ID @ x<>"id(stm)"|}"
   "process stm_"name(stm)" = $id: ID @ begin"
   "values"
      "topregions={"sep ","{id(r) | r in set stm.immediateregions}"}"
   t_element_actions(stm,b)
   "actions"
      t_machine(stm,b)
   "@" t_internal_main_action(stm)
   "end"
   "chansets"
      "chanset_"name(stm)" = {|enter."id(stm)".x | x in set {"sep
         ","{id(r) | r in set stm.immediateregions}"} |}"
         "union {|entered."id(stm)".x | x in set {"sep ","{id(r) | r in
            set stm.immediateregions}"}|}"
         "union {|fire,fired|}"
         "union {|"name(b)"_inevent.x | x in set {"sep ","{id(r) | r in
            set stm.immediateregions}"}|}"
         "union {|"name(b)"_consumed.x | x in set {"sep ","{id(r) | r in
            set stm.immediateregions}"}|}"
```

This rule defines the process that organises the internal structure of the state machine. This process is parametrised by an identifier for the instance of the block to which the state machine belongs. Its behaviour is defined by a recursion that, at each iteration, receives an event through the channel `inevent`, communicates it to its active states and regions, and indicates through `consumed` the result of processing the event. We remind that the communications through channels `inevent` and `consumed` happen according to synchronisation with the controller process of the block, as discussed in Section 4.5.4. Thus, the state machine only reacts in case the expected events of its active states are available in the event pool of the block. This processing may lead to actions being executed, transitions being triggered, and states being activated and deactivated. The model of the execution of actions and the verification of conditions of transitions may involve communications over the channels `get_` (to read the state of the block's process), `set_` (to modify that state), `⋆_op` (to answer and send operation calls) and `⋆_sig` (to send signals), where ⋆ stands for any block name.

The CML actions that model elements of the state machine (states, transitions, and so on) define protocols that specify how they interact with each other. They are defined by rule `t_element_actions`. This level of granularity allows us to focus on particular elements when analysing the CML model, and to trace back any issues to the original SysML model. For instance, the six states of state machine for the dwarf signal shown in Figure 3.4 on page 48 have their CML actions defined by the function `t_element_actions`. If an error occurs in the Stop state, then the event is marked by the name of the corresponding action of this state allowing the traceability to the state in the SysML model.

The actions that model the elements of the state machine are coordinated by a `machine` action, which is defined by the invoking Rule 5.2. It defines the iterative behaviour described above; it initialises the state machine and controls the processing of events. In the initialisation, `machine` requests the actions that model the top regions of the state machine to carry out the behaviour corresponding to entering the regions. This leads to a request for substates to be entered, and so on, until all regions of an active state are active, and exactly one substate of each active region is active.

Rule 5.2: t_machine

```
t_machine(stm: StateMachine,b:Block): action declaration =
    "machine_"name(stm)" ="
        "(||| r in set topregions @ [{}] enter!"id(stm)"!r -> "
            "entered!"id(stm)"!r -> Skip);"
        "mu X @ ("name(b)"_inevent!"id(stm)"?e ->"
            "(dcl aux: bool @ aux := false;"
                "("
                    "(||| r in set topregions @ [{}]"
                        name(b)"_inevent!r!e ->"
                        name(b)"_consumed!r?y -> Skip)"
                    "[|{}|{|"name(b)"_consumed|}|{aux}|]"
                    "(for all i in set topregions do"
                        name(b)"_consumed?x?y -> "
                        "aux:=DL_or(aux,y)"
                    ")"
                ");"name(b)"_consumed!"id(stm)"!aux -> Skip"
            "); fire -> fired -> X"
        ")"
```

The processing of events is defined by a loop in which, at each iteration, the `machine` action accepts an event from the block (process) synchronising on channel `inevent`, sends it to the top-region actions, sends the result back to the block process on channel `consumed`, executes the actions for the transitions and waits for the transition actions to finish executing, before recursing. The response to the block is stored in the auxiliary variable `aux`: if the event can be treated, it yields true, otherwise, it yields false and the event is discarded. If it cannot be treated but the state can defer such an event, then `aux` yields `<defer>` and the event is stored in the sequence of deferred events of the controller process, as described in Section 4.5.4. The interactions between the actions that model states and regions are similar.

After the acknowledgement of an event, a transition from the current state may be fired. The action compartment of a transition may describe several types of behaviour. For instance, invoking operations, sending signals, manipulating attributes of the block, and even invoking an activity of the block. All of these behaviours must follow the CML action language as described in Section 3.1.

For instance, consider the state machine of Figure 2.5 on page 27 for block Device of the leadership election problem. When the block is in the state Off and it receives a turnOn event, the block process synchronises on the `Device_inevent` channel with the state machine process communicating this event. As there is only one region and the event can be consumed at the current state, the state machine process communicates the value **`true`** on the `Device_consumed` event and the `fire` event synchronises with the CML action of the transition enabling it to execute the action compartment, which in this case is the assignment petition := petition-1. Next, the `machine` and the transition actions, which are both of the state machine process, synchronise on the `fired` channel enabling the `machine` action to recurse to wait for the next event of the block process. More details on the integration of state machines and other diagrams are provided in Chapter 7.

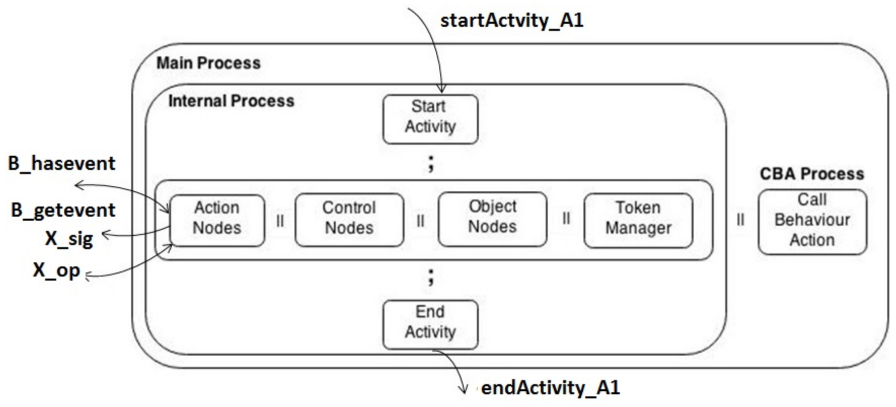
5.2 Activity Diagram

In this section, we formalise the translation of activity diagrams and illustrate it with a few examples. Section 5.2.1 gives an overview of the translation of activity diagrams. As the examples presented in Chapter 3 do not illustrate all translation rules we propose, some other examples are introduced in Section 5.2.2. The translation rules are presented in Section 5.2.3. Finally, we discuss related work in Section 5.3.

5.2.1 Overview

Figure 5.2 depicts how an activity process is structured. Each activity diagram is described by the Main Process, whose behaviour is specified as the parallel composition of the (internal) behaviour (Internal Process) of the activity itself with other processes for activities

Figure 5.2: Overview of the representation of activities in CML.



Source: Author's ownership.

that may be used inside this activity as call behaviour actions (CBA Process). If there is no call behaviour action, then the main process is simply the internal process. A call behaviour action is an action node in the activity diagram that invokes the behaviour of another activity, hence, to use such behaviour we compose the main activity process in parallel with other activity processes that are used in call behaviour actions.

In the internal process of an activity, we have a CML action for each node of the diagram (action, object, control). Due to the token semantics of activity diagrams, there is also a CML action Token Manager for managing tokens; this is the action that models the control of ending an activity according to available tokens. All CML actions are composed in parallel to define the behaviour of the activity process.

The main action of the internal process is recursive. Each iteration runs one execution of the activity via a sequence of three actions. The first, Start Activity, synchronises on the `startActivity_A1` channel, where `A1` is the name of the activity. This channel starts the activity flow and can possibly take any value as input when needed. For instance, the state machine Leadership Election of Figure 2.5 (page 27) calls the activity ActBroadcast via the action `call ActBroadcast` passing the claim of the Device as argument. In the CML model, the state machine and the activity processes synchronise on the channel `startActivity_ActBroadcast`. The Start Activity action is followed by the parallel composition of the CML actions for all nodes of the diagram along with the Token Manager. The third action, End Activity, communicates via the `endActivity_A1` that the activity has finished along with any output values it may have. For instance, when the activity ActBroadcast ends its flow, it synchronises with the process for the state machine LeadershipElection on `endActivity_ActBroadcast`.

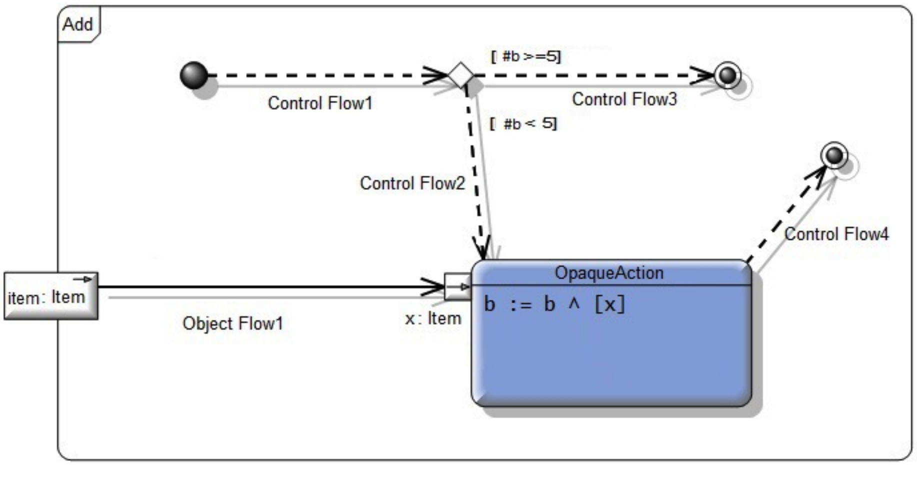
Control and object flows are established via synchronisations. Actions that model nodes have channels for this purpose, so that the alphabetised parallelism of these actions enforce the order of execution of nodes depicted in the diagram. We provide a CML representation for object nodes, control nodes, and several actions including call operation, send signal, accept event, opaque, value specification, call behaviour, read self, and read structural feature. These representations are presented in Section 5.2.3.

5.2.2 Examples

We depict two examples of activity diagrams, where they describe operations of a producer-consumer problem. They illustrate the operations *add* and *rem*, which correspond to adding and removing an item to/from the block Buffer, respectively.

The first diagram displayed in Figure 5.3 represents the **Add** activity. It has all kinds of activity nodes: action, control and object nodes. Its behaviour starts at the initial node and the activity parameter node **item**, which is of the type **Item**. A decision node checks if the size of the buffer **b**, which is a sequence of items, is less than five (the size of the Buffer). If so, another item can be added, otherwise the activity ends. When it is less than five, an opaque action is fired and it receives as input (through an input pin) the item to be added. An opaque action is a node whose behaviour is described in terms of another language (e.g. a programming language). In our case, we assume that the content of opaque actions is described in terms of the CML action language (the same that is used in the action compartment of state machine transitions). As the content of the opaque action is a CML code, it is executed when the action is ready to be performed (i.e, once all input tokens have been provided to the action). Such code just adds to the buffer the element received as parameter. After this action is performed, the control edge leaving this action leads to an activity final node, which ends the diagram execution.

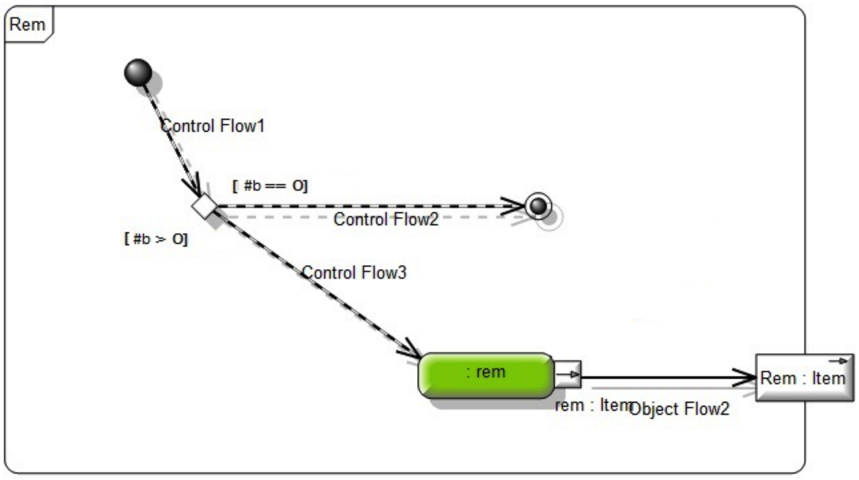
Figure 5.3: Activity Diagram: Adding an element to the Buffer



Source: Author's ownership.

The **Rem** activity diagram removes an item from the buffer and returns it as output. Figure 5.4 shows the behaviour of such an activity. It has no inputs, and after the initial node, a decision node checks if there is at least one element to be removed, otherwise it ends the activity. In case it has elements, a call operation action is fired, which should deal with the task of removing an element of the buffer and returning it as output. This call operation action returns an output, which is put in its output pin. Next, this data is made available as output of the activity by means of the activity parameter node **Rem** of the type **Item**. The edge leaving the action represents an object flow that transports the data to the activity parameter node.

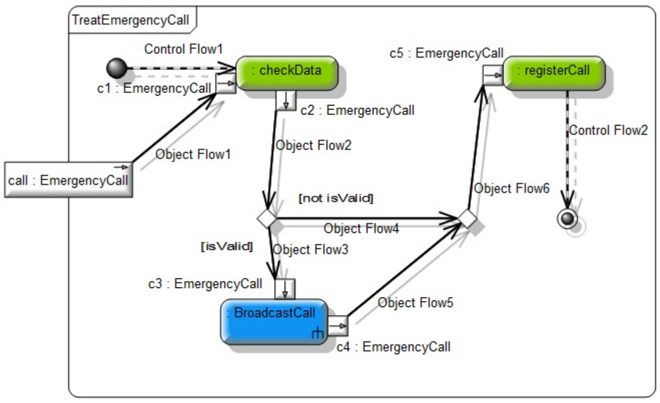
Figure 5.4: Activity Diagram: Removing an element of the Buffer



Source: Author's ownership.

Finally, we illustrate the usage of call behaviour actions by using two diagrams showing scenarios of an emergency call system. Figure 5.5 shows the process to treat an emergence call. First, it verifies the information of the call (see the **checkData** call operation action), then, if the data is valid the call is broadcasted. The act of broadcasting is defined by another activity, which is depicted in Figure 5.6, thus, the **TreatEmergencyCall** activity calls the activity **BroadcastCall** using a Call Behaviour action. Eventually, the call is registered and the activity finishes.

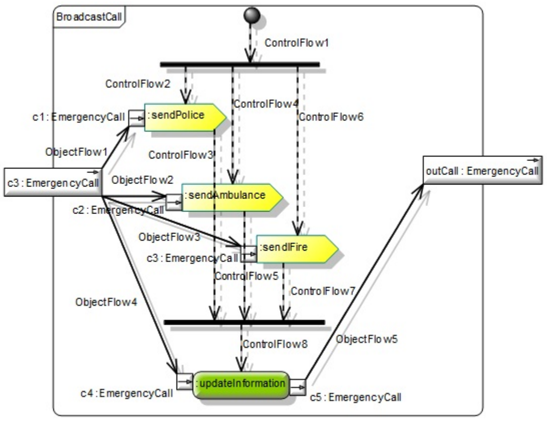
Figure 5.5: Activity Diagram: Treat Emergency Call



Source: Author's ownership.

Figure 5.6 shows the activity that is invoked by the Call Behaviour action. It broadcasts the call by sending three signals in parallel (**sendPolice**, **sendAmbulance** and **sendFire**) before updating the information of the call and sending it as an output parameter. In each one of the send signal actions we omit the target pin, which defines to whom the signal must be sent, because we can infer it from the name of the signal. However, each one must have a target pin indicating the destination block of the signal.

Figure 5.6: Activity Diagram: Broadcasting a Call



Source: Author's ownership.

We use these examples to illustrate how the translation functions generate the respective CML code.

5.2.3 Formal semantics

In this section we present some of the translation rules for activity diagrams. We first present the translation rule for activity diagrams of a block (Section 5.2.3.1), then we focus on the translation of a single activity (Section 5.2.3.2). We define the rule for the internal activity diagram in Section 5.2.3.3. The main action that composes all elements involved in a activity diagram is presented in Section 5.2.3.4. The remaining rules for activities are available in the Section A.1 of the Appendix A.

5.2.3.1 Block Activity Diagrams

The first translation function, which is defined in Rule 5.3, is `t_activity_diagrams`. This is the root function for activity diagrams. The parameters of this function provide a sequence of activity diagrams and a block to which these activities belong, resulting in a sequence of program paragraphs. This function introduces channels related to control flow, and a process.

The channel `control` is used to deal with the flow of the control tokens in an activity diagram. The execution of an action or control node linked to a control flow is only allowed with the availability of the control token. The type of the channel `control` is **`nat`**. We assume that each control edge is given a unique identifier of type **`nat`**. So, through `control` we define which action node can be executed. The channel `endDiagram` communicates that a diagram has finished its execution either because it has reached a final activity node or because there is no active token in the diagram. The channel `interrupted` is used to indicate that an interrupting edge of an interruptible region has been traversed. An interruption is related to a diagram, a block instance, and an interruptible region index. As activity diagrams use a token semantics, we use two channels to keep the amount of active tokens running inside a diagram. The channel `update` changes the number of active tokens and the channel `clear` sets the amount of active tokens to zero. Both are indexed by the ID of the activity node that communicates the event. We assume that this ID is a CML token that uniquely identifies a node in the context of an activity diagram. The channels `inc` and `dec` are used to control the possibility of interruption of a diagram as explained in Rule A.15. The channel `wait` is used to block a node while it waits for termination of the diagram.

By calling the function `t_ad_channels(ad, block)` for every activity diagram of the sequence `ads`, more channels are introduced to deal with communication that occurs among diagrams and inside diagrams. All these channels are internal to an activity diagram process and they will be hidden by using the channel sets `_Hidden` prefixed by the name of the diagram. Also, the iterative call to function `t_activity_diagram` provides a translation of every activity diagram in `ads`.

The name of the process that the function `t_activity_diagrams` introduces is defined by the name of the block instance received as argument (`name(block)`) appended with `_ads`. The process definition is given by the interleaving of all activity diagram processes of that block.

Rule 5.3: t_activity_diagrams

```
t_activity_diagrams(ads: seq of Activity, block: Block): seq of program
    paragraph =
    "channels
       control: nat
       endDiagram: ID
       interrupted: ID*ID*nat
       update: ID*int
       inc, dec, clear: ID
       wait"

    for ad in ads do
       t_ad_channels(ad, block)
    end for

    "chansets"
    for ad in ads do

       name(ad)"_Hidden = {|control, endDiagram, interrupted, update,
           clear, wait, inc, dec"

                 if ad.edges(ObjectFlow.Type).size > 0 then
                    ", "
                 end if
```

```
                for ObjEdge in seq ad.edges(ObjectFlow.Type) sep "," do
                   "obj_"name(ad)"_"ObjEdge.index
                end for

                if ad.Nodes(Pin.Type) > 0 then
                   ","
                end if
                for action in seq ad.Nodes(Action.Type) do
                   for inPin in seq action.inputPins sep "," do
                      "in_"name(ad)"_"name(action)"_"inPin.edge.index
                   end for
                   for outPin in seq action.outputPins sep "," do
                      "out_"name(ad)"_"name(action)"_"outPin.edge.index
                   end for
                end for
                "|}"
   for ad in ads do
      t_activity_diagram(ad, block)
   end for

   "process "name(block)"_ads = val $id: ID @" sep "|||"
      {"ad"_name(ad)"($id^[mk_token(\""name(ad)"\")])" | ad in seq ads}
```

The respective CML code generated by this rule for the **Add** and **Rem** activity diagrams described earlier is displayed in the following extract.

```
channels
  control: nat
  endDiagram: ID
  interrupted: ID*ID*nat
  update: ID*nat
  inc, dec, clear: ID
  wait
...

chansets
  Add_Hidden = ...
  Rem_Hidden = ...

process ad_Add = ...
process ad_Rem = ...

process Buffer_ads = val $id: ID @
ad_Add($id^[mk_token("Add")]) ||| ad_Rem($id^[mk_token("Rem")])
```

In this extract, there are two processes (`ad_Add` and `ad_Rem`) for the activities **Add** and

Rem. The process `Buffer_ads` combines these two processes in parallel. The channels deal with control flow, indication of finalisation of a specific diagram, end of a flow, and indication that an interrupting edge has been traversed.

The function `t_ad_channels` (Rule 5.4) introduces channels for a specific activity `ad` of a Block instance `block`. An activity may be interrupted by communicating the channel `interruptActivity`. To start an activity, we must provide values for its input parameter nodes in the channel `startActivity`. The parameters of this channel are the ID followed by the types of the activity parameter nodes that do not have incoming edges, but have outgoing edges. In other words, they are used for the input of an activity. After an activity finishes, it provides values in the channel `endActivity` whose type is defined in a similar way to that of `startActivity`. On the other hand, parameter nodes related to `endActivity` have no outgoing edge, but have at least one incoming edge. Notice that the channel names `startActivity`, `interruptActivity` and `endActivity` are appended by an underscore (_) followed by the activity name.

Similarly to an activity, a call behaviour action has channels for providing input values and for obtaining output values. The channel named `startActivity_CBA_` is appended to the diagram name given by `name(ad)`; its type is given by the diagram identifier, the Call Behaviour Action identifier (natural), and the types of the input activity parameter nodes. The channel `endActivity_CBA_` is similar to `startActivity_CBA_`, except that it gets values that result from a Call Behaviour Action execution.

Every object edge of an object flow is a channel whose name is `obj_` followed by the activity `name(ad)`, and an edge index through which the object flows. The type of this channel is the type of the object flow source.

We introduce channels for the input and the output pins of each action node. Communication through an input pin uses a channel named `in_` followed by the activity name, the action name and the input pin index. The type is given by the input pin type. Similarly, for communication through an output pin, we define channels named `out_` followed by the same pattern as for input pins.

Rule 5.4: t_ad_channels

```
t_ad_channels(ad: Activity, block: Block): seq of channel =

   "interruptActivity_"name(ad)": ID"

   "startActivity_"name(ad)": ID"
     if (card {param | param in seq ad.Nodes(ActivityParameterNodes.Type)
        and param.IncomingEdges.size() == 0 and
        param.OutgoingEdges.size() > 0}  > 0) then
       "*"
     end if
     sep "*" {t_types(param) | param in seq
        ad.Nodes(ActivityParameterNodes.Type) and
        param.IncomingEdges.size() == 0 and param.OutgoingEdges.size() >
        0}
    "endActivity_"name(ad)": ID"
      if (card {t_types(param) | param in seq
          ad.Nodes(ActivityParameterNodes.Type) and
          param.OutgoingEdges.size() == 0 and param.IncomingEdges.size() >
          0} > 0 then
        "*"
      end if
      sep "*" {t_types(param) | param in seq
          ad.Nodes(ActivityParameterNodes.Type) and
          param.OutgoingEdges.size() == 0 and param.IncomingEdges.size() >
          0}

    for cba in seq ad.Nodes(CallBehaviour.Type) do
        "startActivity_CBA_"name(ad)": ID*nat"
          if (cba.IncomingEdges(Object.Type).size > 0) then
            "*"
          end if
          sep "*" { t_types(edge) | edge in cba.IncomingEdges(Object.Type) }
        "endActivity_CBA_"name(ad)": ID*nat"
          if cba.OutgoingEdges(Object.Type).size > 0 then
            "*"
          end if
          sep "*" { t_types(edge) | edge in cba.OutgoingEdges(Object.Type) }
    end for

    for edge in seq ad.ActivityEdges do
       if edge instanceof ObjectFlow then
          "obj_"name(ad)"_"edge.index": " t_types(edge.source)
       end if
    end for

    for action in seq ad.Nodes(Action.Type) do
       for inPin in seq action.inputPins do
          "in_"name(ad)"_"name(action)"_"inPin.index":" t_types(inPin)
       end for
       for outPin in seq action.outputPins do
          "out_"name(ad)"_"name(action)"_"outPin.index":" t_types(outPin)
       end for
    end for
```

The respective CML code generated by Rule 5.4 regarding the additional channels of the **Add** and **Rem** activity diagrams is displayed in the following extract (we assume that the IDs of the diagrams are their own names):

```
channels
  ...
  interruptActivity_Add: ID
  startActivity_Add: ID*Item
  endActivity_Add: ID
  obj_Add_1: Item
  in_Add_OpaqueAction1_1: Item

  interruptActivity_Rem: ID
  startActivity_Rem: ID
  endActivity_Rem: ID*Item
  obj_Rem_1: Item
  out_Rem_rem_1: Item
  ...
```

In the next section, we introduce the function to translate a single activity diagram.

5.2.3.2 Single Activity Diagram

The function `t_activity_diagram` (Rule 5.5) defines the structure of an activity shown in Figure 5.2 in Section 5.2.1. It gives as results two process definitions. The first process is produced by `t_ad_internal_process` and is related to the nodes and edges of an activity diagram. The second process is named `ad_` followed by the name of the activity diagram. It is parametrised by a block identifier. The process definition is given by a generalised parallelism of two processes: the one introduced by `t_ad_internal_process` and the other for Call Behaviour actions. The diagram that requests an execution of a Call Behaviour action must synchronise in the channel that allows the Call Behaviour action to start its execution. After this, the action that was called synchronises with the diagram in a channel to indicate that it has finished. In this way, the diagram process gets control back. The function `t_ad_cba_parallel` is used to define the parallelism of an activity with call behaviour actions. This rule is defined in the Appendix A.1.

Rule 5.5: t_activity_diagram

```
t_activity_diagram(ad: Activity, block: Block): seq of program paragraph =
   t_ad_internal_process(ad, block)
   "process ad_"name(ad)" = val $id: ID @"
      "ad_internal_"name(ad)"($id)"
   t_ad_cba_parallel(ad, block)
```

5.2.3.3 Internal Activity Diagram

Rule 5.6 translates the nodes and edges of an activity. The function `t_ad_internal_process` takes as arguments an activity and a block. The function introduces a process with name `ad_internal_` appended with the activity name. It receives the block instance as parameter and has as state the attribute `nTokens` that keeps the number of active tokens inside an activity diagram. Finally, the function `t_ad_actions` determines the actions used by this process.

Rule 5.6: t_ad_internal_process

```
t_ad_internal_process(ad: Activity, block: Block): seq of process =
   "process ad_internal_"name(ad)" ="
   if block != null then
      "val $id: ID"
   end if
   "@ begin
      state
         nTokens: nat := 0"
   t_ad_actions(ad, block)
   "end"
```

The next extract shows the application of Rule 5.6 regarding the **Add** activity diagram. The translations of the other diagrams are similar.

```
...

process ad_internal_Add = val $id: ID @ begin
  state
    nTokens: nat := 0
  actions
    ...
end
...
```

The function `t_ad_actions` (Rule 5.7) receives as arguments an activity and a block instance. The translation of activity actions is performed by the translation of actions of an activity diagram that belong to a block and by introducing a CML main action. Notice that the function `t_ad_actions` just introduces CML actions, as indicated by the use of the reserved word **`actions`**.

Rule 5.7: t_ad_actions

```
t_ad_actions(ad: Activity, block: Block): seq of action =
"actions"
   if block != null
      t_block_actions(ad, block)
   end if
   t_main_ad_action(ad,block)
```

To introduce CML actions that correspond to actions of an activity diagram, we call the function `t_block_actions` (Rule 5.8) that receives an activity and a block instance as arguments. This function calls other functions to translate activity nodes. The function `t_start_activity` introduces a CML action that defines how an activity diagram starts its execution. The CML action `END_DIAGRAM` is used by every CML action related to an activity node to synchronise the termination of the diagram. It communicates via the channel `endDiagram` that this diagram has finished and then behaves as **`Skip`**. We use the function `t_interruptible_regions` to introduce models of interruptible regions. For each node in an activity diagram we select an adequate function based on the node type. There are separate functions for action nodes, control nodes, and object nodes. The action `Nodes` is defined as the alphabetised parallelism of all actions that correspond to the nodes of an activity diagram. If the activity has parameter nodes, then we define value-result parameters (**`vres`**) for the CML action `Nodes` so that when such a parameter is updated by one CML action, the other actions can use this updated value. According to the type of the node a different function is called to recover the synchronisation alphabet of the node (channel set): `t_channels_action_node` for actions, `t_channels_control_node` for control nodes and `t_channels_object_node` for object nodes. These functions return all channels used by each node in their translations. For example, a control node that has two edges, one incoming and one outgoing edge both from control flow with indexes 1 and 2, respectively, has the following channel set as alphabet: `{|control.1, control.2|}`. The functions `t_token_manager` is called to introduce CML actions that establish the termination rules for an activity diagram according to its active tokens. The function `t_interrupt_activity_manager` controls the availability of the diagram to be interrupted by the external environment. Most of the rules related to the functions called inside Rule 5.8 are presented in Appendix A.1.

Rule 5.8: t_block_actions

```
t_block_actions(ad: Activity, block: Block): seq of action =
   t_start_activity(ad,block)
   "END_DIAGRAM = endDiagram."id(ad)" -> Skip"

   t_interruptible_regions(ad,block)

   for node in seq ad.Nodes do
      switch(node.Type)
         case Action.Type: t_action_node(node, ad, block,
            node.inInterruptibleRegion)
         case ControlNode.Type: t_control_node(node, ad, block,
            node.inInterruptibleRegion)
         case ObjectNode.Type: t_object_node(node, ad, block,
            node.inInterruptibleRegion)
      end switch
   end for

   "Nodes = "
```

```
    if (card {param in seq ad.Nodes(ActivityParameterNodes.Type)} > 0) then
          "vres " sep ", vres " { name(param)": "t_types(param) | param in
             seq ad.Nodes(ActivityParameterNodes.Type) } "@"
    end if
       define_alphabetised_parallel({
          for node in seq ad.Nodes sep ","
             switch(node.Type)
                case Action.Type:
                   (name(node)"_"node.index,
                      t_channels_action_node(node,ad,block,
                       node.inInterruptibleRegions))
                case ControlNode.Type:
                   ("CNode_"node.index,
                      t_channels_control_node(node,ad,block,
                       node.inInterruptibleRegions))
                case ObjectNode.Type:
                   if (node.type == ActivityParameterNode.Type) then
                      ("ObjNode_"node.index"("name(node)")" ,
                         t_channels_object_node(node,ad,block,
                          node.inInterruptibleRegions))
                   else
                      ("ObjNode_"node.index,
                         t_channels_object_node(node,ad,block,
                          node.inInterruptibleRegions))
                   end if
             end switch
          end for
          })

    t_token_manager(ad,block)
    t_interrupt_activity_manager(ad,block)
```

The next extract shows the application of Rules 5.7 and 5.8 to **Add**. We assume that the indices for the opaque action, initial node, decision node, the two activity final nodes, the input pin and the input activity parameter node are, respectively, 1, 1, 2, 3, 4, 1 and 2. This information comes from the underlying SysML model.

```
process ad_internal_Add = val $id: ID @ begin
...
actions
  START_ACTIVITY = ...

  END_DIAGRAM = endDiagram.[mk_token("Add")] -> Skip

  OPAQUEACTION_1 = ...
  CNode_1 = ...
  CNode_2 = ...
  CNode_3 = ...
  CNode_4 = ...
  ObjNode_1 = ...
  ObjNode_2 = ...

  Nodes = (OPAQUEACTION_1 [ {|...|} || {|...|} ]
  (CNode_1 [ {|...|} || {|...|} ]
  (CNode_2 [ {|...|} || {|...|} ]
  (CNode_3 [ {|...|} || {|...|} ]
  (CNode_4 [ {|...|} || {|...|} ]
  (ObjNode_1 [ {|...|} || {|...|} ] ObjNode_2 ))))))

  TOKEN_MANAGER = ...

  INT_ACT_MANAGER = ...
  ...
end
```

5.2.3.4 Main Action

The function `t_main_action` (Rule 5.9) introduces the main CML action for an activity. It receives an activity and a block instance as arguments. The main action is recursive and it is composed of a block declaration (for variables) and an action. The first variable (`$source`) is used to store the entity ID that called the activity in the `START_ACTIVITY` action. Then, its value is used in the `endActivity` channel that represents the return of the activity to the caller. The other variables have the same name as the activity parameter nodes that are used to hold activity input or output values. The types of these variables are defined as the types of the parameter nodes. These variables are assigned default values defined by the function `default(t_types(param))`.

The action begins with `START_ACTIVITY`, which represents the beginning of the diagram and has as start event the channel `startActivity_`. This action is sequentially composed by a generalised parallelism of actions `Nodes` and `TOKEN_MANAGER`, and interruptible regions (if they exist in the diagram). The `Nodes` action synchronises with the `TOKEN_MANAGER` action on the channels used to change the number of active tokens (`update` and `clear`), to block a CML action (`wait`), and to terminate the diagram (`endDiagram`). The `InterruptibleRegions` action has another function to get the channel set of synchronisation (`t_chanset_int_regions(ad)`), which returns the channels used in the interruptible region actions.

This parallelism runs until the diagram finishes its execution. Then, the main action communicates over the channel named `endActivity_`, appended with the diagram name. The output data of this channel is defined by the names of the activity parameter nodes that have no outgoing edges, just incoming edges (output parameter nodes), that is, the return values of the activity. After the `endActivity` is communicated, the main action recurses. The main action can be interrupted by an external source communicating over the `interruptActivity` channel. However, due to its parallelism with the `INT_ACT_MANAGER`, it is only available according to the behaviour defined in this latter action. Notice that we use the hiding operation `\\` to hide internal channels in set `[name(ad)]_Hidden`, so that they are not visible outside the main action.

Rule 5.9: t_main_ad_action

```
t_main_ad_action(ad: Activity, block: Block): action =
   "@
   mu X @ ( (dcl $source: ID " { ", "name(param)": "t_types(param) |
      param in seq ad.Nodes(ActivityParameterNodes.Type) } "@"
   " (
      (START_ACTIVITY($source" {", "name(param)| param in seq
         ad.Nodes(ActivityParameterNodes.Type) and
         param.OutgoingEdges.size > 0 } ")"
      "; ((Nodes"
      if (card {param in seq ad.Nodes(ActivityParameterNodes.Type)} > 0)
         then
         "(" sep "," { name(param)| param in seq
            ad.Nodes(ActivityParameterNodes.Type) } ")"
      end if
      " [|{|update, clear, wait, endDiagram|}|]TOKEN_MANAGER)"
      if ad.InterruptibleRegions.size > 0 then
         "[|"t_chanset_int_regions(ad, block)"|] InterruptibleRegions);"
      else
         ");"
      end if
      "endActivity_"name(ad)"!$source" {"!"name(param) | param in seq
         ad.Nodes(ActivityParameterNodes.Type) and
         param.OutgoingEdges.size() == 0 and param.IncomingEdges.size()
         > 0 }
      " -> Skip));X /_\ interruptActivity_"name(ad)"?x:(prefix($id,x)) ->
         X)"
      "[|{|inc,dec|} union {|interruptActivity_"name(ad)".x | x: ID @
         prefix($id,x) |}|] INT_ACT_MANAGER) \\ "name(ad)"_Hidden"
```

Finally, the next extract details the application of Rule 5.9, which represents the main action of the activity diagram. Here we depict the two main actions of each one of the activity diagrams **Add** and **Rem**. Notice that the **Add** diagram has an input activity parameter node, and then a local variable is created (*item*, same as the name of the node) to store the input data. The **Rem** diagram returns an item, which is represented by the local variable of the same name of the node, as can be seen in the output activity parameter node from Figure 5.4 on page 104.

```
...
process ad_internal_Add = val $id: ID @ begin
chansets
...
actions
...
@
mu X @ ( (dcl $source: ID, item: Item := null @ (
(START_ACTIVITY($source,item);((Nodes(item)
[|{|update,clear,wait,endDiagram|}|]
TOKEN_MANAGER));endActivity_Add!$source -> Skip));X /_\
interruptActivity_Add -> X)
[| {|inc,dec|} union {|interruptActivity_Add.x | x:ID @
prefix($id,x)|} |] INT_ACT_MANAGER) \\ Add_Hidden
end
...

process ad_internal_Rem = val $id: ID @ begin
actions
...
@
mu X @ ( (dcl $source: ID, Rem: Item := null @ (
(START_ACTIVITY($source);((Nodes(Rem)
[|{|update,clear,wait,endDiagram|}|]
TOKEN_MANAGER));endActivity_Rem!$source!Rem -> Skip));X /_\
interruptActivity_Rem -> X)
[| {|inc,dec|} union {|interruptActivity_Rem.x | x:ID @
prefix($id,x)|} |] INT_ACT_MANAGER) \\ Rem_Hidden
end
...
```

We recall that the termination of activities is controlled by the action `TOKEN_MANAGER`, which is detailed in Section A.1.5 of Appendix A. For instance, once a final node is reached, it clears all the tokens that are controlled by the `TOKEN_MANAGER` and this enables the termination of the diagram.

The presented set of rules provide mechanisms to represent meaningful activities in CML. A vast number of constructs is accepted in order to allow very expressive SysML models. Next we detail some similar works that provide formal semantics to UML/SysML activities.

5.3 Related work

Xu et al. (XU et al., 2008, 2009) formalise UML activity diagrams and define a set of mapping rules from the formal model for activity diagrams into CSP (HOARE, 1985). They introduce a formal meta-model for activity diagrams. This meta-model is given by a tuple of elements that represent the different nodes of an activity diagram, a set of directed edges, and the flow relationship between them. Translation functions are defined for each of the constructs, however, the strategy is not compositional; the activity nodes are not translated independently. Our semantics does not have this limitation as each node is translated independently of the others. Some of their mappings differ from ours. For instance, in their work a decision node is mapped into a guarded event. In our translation, we use Dijkstra's guarded commands for the translation of a decision node. The reason is that if two or more guards are true, the edge to be traversed is non-deterministically chosen. They deal strictly with control flow. There are no mapping rules for pins. Also, they do not deal with send signal or accept signal actions. They deal with the beginning of the execution of an activity diagram as an internal choice between initial nodes, but they do not take into account call behaviour action without incoming edges. They do not treat different kinds of action, for instance, call operation, call behaviour, and value specification. Other features of activity diagrams, like accept event action, are translated into CSP.

Table 5.1: Coverage comparison

	Xu et al.	Abdelhalim et al.	Bisztray et al.	Our Work
CBA	×	×	×	✓
IRg	✓	✓	×	✓
COA	×	×	×	✓
OpA	×	×	×	✓
AEv	×	✓	×	✓
SSA	×	✓	×	✓
VSA	×	✓	×	✓
IN	✓	✓	✓	✓
FF	✓	✓	✓	✓
DN	✓	✓	✓	✓
MN	✓	✓	✓	✓
FN	✓	✓	✓	✓
JN	✓	✓	✓	✓
AFN	✓	✓	✓	✓
ObjN	×	✓	×	✓
CtrF	✓	✓	✓	✓
ObjF	×	×	×	✓
APN	×	✓	×	✓

Source: Author's ownership.

Abdelhalim et al. (ABDELHALIM et al., 2010) propose the use of a subset of fUML (Foundational Subset for Executable UML) that is mapped into CSP (HOARE, 1985). Their focus is on the analysis of dynamic behaviours. As control flow has been addressed by (XU et al., 2008, 2009); they concentrate on mapping Send Signal action, Accept Event action and signals (ABDELHALIM et al., 2010). They deal with decision node as an internal choice. Also, they map expansion regions into CSP processes. They deal with signals by means of an asynchronous buffer, whereas in our translation we use a one-place synchronised buffer that could receive data from an asynchronous buffer. Their communication model allows storing of signals.

Bisztray et al. (BISZTRAY et al., 2007) define translation rules that relate edges in an activity diagram to a process in CSP (HOARE, 1985). They do not deal with object nodes or object flow, just with the translation of control flow. They translate a join node separately from the fork node. In other words, they have distinct translation rules for these control nodes. A consequence is that a synchronisation event appears only in the process that reaches the join node, but not in the parallel operator that is introduced in the fork node. Also, the translation of the join node results in processes that are not similar: only one will behave as the process after the join, all the others will terminate in `Skip`. In our case, we treat each node separately and the nodes synchronise according to the links between them. We do not have to worry about how several flows created from a fork node will terminate in nodes that consume tokens (join, flow final, activity final and output parameter nodes).

Table 5.1 presents a comparison of the coverage of our formalisation with related work. The left column contains the features we formalised. The $\checkmark$ indicates that the feature is covered by related work (column), whereas $\times$ indicates it is not. We cover the following features: Call Behaviour Action (CBA), Interruptible Region (IRg), Call Operation Action (COA), Opaque Action (OpA), Accept Event Action (AEv), Send Signal Action (SSA), Value Specification Action (VSA), Initial Node (IN), Flow Final Node (FF), Decision Node (DN), Merge Node (MN), Fork Node (FN), Join Node (JN), Activity Final Node (AFN), Object Node (ObjN), Control Flow (CtrF), Object Flow (ObjF), and Activity Parameter Node (APN).

6 Sequence Diagram

In this chapter we present the CML semantics for sequence diagrams. This kind of diagram is used for describing scenarios of the system in terms of block instances and the messages exchanged between them. It has a distinct characteristic of representing interactions between blocks.

This chapter is organised as follows. Section 6.1 gives an overview of the translation of sequence diagrams. Our rules for sequence diagrams are presented in Section 6.2. Related work is presented in Section 6.3.

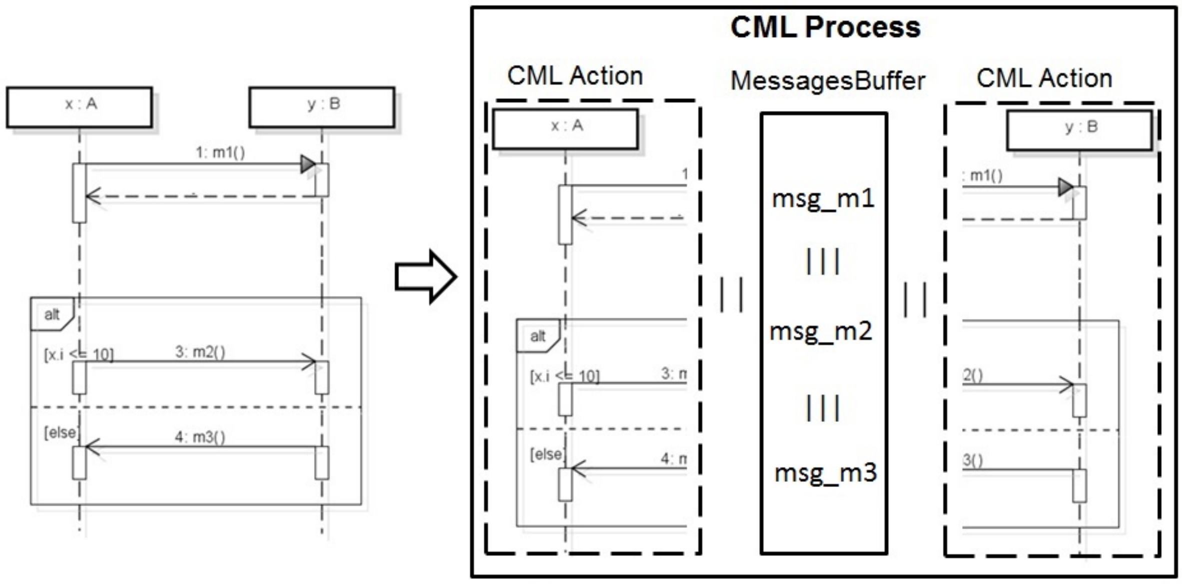
6.1 Overview

A sequence diagram is defined in terms of a CML process. Figure 6.1 shows how the semantics of sequence diagrams is captured by CML elements. Each lifeline is represented by a CML action defined by the sequential composition of other CML actions that represent fragments that happen in the lifeline: message occurrences, combined fragments, state invariants and interactionUse elements.

The CML process that models a sequence diagram is parametrised by the identifiers (of type `ID`) of the block instances that either are used in a lifeline or send messages through gates. This makes the model of the diagram as generic as the diagram itself, which is valid for any instances of the block types used in the diagram.

Each message exchange is represented in CML by two communications, one corresponding to the point where the message is sent and another to the point where it is received. The channels used are `mOP` and `mSIG`. They are similar to `op` and `sig` presented in Chapter 4 except that they carry one extra information on the channel regarding the event type of the message, which can be `s` for a sending event, or `r` for a receiving event. Another difference is the meaning of the index of the message. While in the `op` and `sig` channels they are used for differing messages with the same signature, for the `mOP` and `mSIG` channels, this value is used to uniquely identify each message inside the sequence diagram. This difference is irrelevant when relating the events because it is used for maintaining internal consistency of the models. That is why

Figure 6.1: Semantic representation of sequence diagrams in CML.



Source: Author's ownership.

these values are hidden in the complete model, as discussed in Chapter 8.

However, changing the message exchanging mechanism has impacted the definition of the semantics in comparison with the other diagrams. One of them requires an internal entity to control the flow of messages between the lifelines because we now differentiate the sending and receiving of messages. Also, the asynchronous nature of the sending and receiving of the messages requires an intermediate component to model the environment through which the messages flow. This component is modelled as a CML action called `MessagesBuffer`.

The `MessagesBuffer` action coordinates the message exchanging between lifelines by relaying messages from one lifeline to another. For each message there is a CML action that synchronises on a sending event of the sender then communicates the receiving event with the receiver. The behaviour of the `MessagesBuffer` is the interleaving of all these message communications.

In the case of a synchronous message, the sending communication is followed by the reply communication, so that the sender stays blocked until the reply is received. For asynchronous messages, the sender is ready to proceed after the sending communication.

The main action of the sequence diagram CML process composes in parallel the lifeline actions together with the `MessagesBuffer` action.

Next, we detail the translation rules for sequence diagrams.

6.2 Formal semantics

Concerning the translation rules, Rule 6.1 is the root function that translate all sequence diagrams. The function `t_sequencediagram` defines the translation of a set of sequence diagrams. First it defines the internal control channels used by the sequence diagram processes and the channel set used to hide these channels from external entities processes. Other channel sets are defined for the synchronisation channels of each lifeline. These channel sets are composed by the events used inside the lifeline actions. Next, the process for each sequence diagram is defined in terms of two other functions.

Rule 6.1: t_sequencediagram

```
t_sequencediagram(sds: set of Interaction): seq of program paragraph =
   "channels
      join, break, interrupt: nat
      strict: nat.nat
      beginCR, endCR: nat
      startRef, endRef: nat
      beginInteraction, endInteraction: ID
      inv, block: ID
      gate_ev: ID.ID.MSG"



   "chansets
      Hidden = {|join, break, invalid_trace, endInteraction,
      strict, beginCR, endCR|}"
   for sd in sds do
      for lf in seq sd.Lifelines do
         name(sd)"_cs_"t_lifeline_name(lf)" =
            "lf_channels(lf.InteractionFragments, lf)
      end for
      name(sd)"_Events = " sep "union" {name(sd)"_cs_"t_lifeline_name(lf)
         | lf in seq sd.Lifelines}
   end for
   let generated = {} within
      for sd in sds do
         t_create_sd_process(sd,generated)
      end for
   end let
```

If the diagram has interactionUse elements, we need to generate the interactions that are referred by them. For that, we use `generated` inside the function `t_create_sd_process` to store which interactions have been generated to avoid the generation of duplicated processes. Any sequence diagram referred by an interactionUse element is translated first. It is important to note that there can be no mutual dependencies, via interactionUse elements, in the construction of sequence diagrams. This possibility is not explicitly ruled out in SysML, but it is in our guidelines as discussed in Chapter 3.

Similarly to activities, a sequence diagram is translated in terms of a internal process and a main process that composes in parallel its internal process with the other processes related to sequence diagrams referred by interactionUse elements. The function `t_create_sd_process` is defined in the Appendix A. It defines the processes for all sequence diagrams invoking function `t_simple_sd`, which is also defined in Appendix A. The latter defines the process for the internal representation of a sequence diagram. Along the definition of this process function `t_sd_actions` is invoked to provide the CML actions of a sequence diagram as presented next.

Rule 6.2 presents the actions that are defined in the internal representation of a sequence diagram. We define CML actions for each lifeline in `t_lf_interaction_fragments`. The Loop combined fragment requires repetition, which is translated to CML in terms of recursion. Thus, we have to define the CML action that recurses separately in the function `t_loop_actions`. The break and the critical combined fragments also require the definition of auxiliary actions, which are defined by functions `t_critical_actions` and the `BREAK` action. The latter simply waits for `break` events to fire the interruption of the enclosing operand of the break fragment communication the `interrupt` event, and when the interaction terminates then this action also terminates because they synchronise on the `endInteraction` event. Finally, functions `t_messages_buffer` and `t_main_action` specify an auxiliary action to control message events and the main action of the sequence diagram process, respectively.

Rule 6.2: t_sd_actions

```
t_sd_actions(sd: Interaction): seq of program paragraph =
   "actions"
      for lf in seq sd.Lifelines do
         "lf_"t_lifeline_name(lf)" =
            "t_lf_interaction_fragments(lf.InteractionFragments, lf)
         if lf.hasLoopFragments() then
            t_loop_actions(lf.getLoopCombinedFragments, lf)
         end if
      end for
      if sd.hasCriticalFragments() then
         t_critical_actions(sd.getCriticalCombinedFragments, sd)
      end if

      if sd.hasBreakCombinedFragment() then
         "BREAK = (break?i -> interrupt!i -> BREAK)
            [] (endInteraction.sd_id -> Skip)"
      end if

      t_messages_buffer(sd)
```

```
       t_main_action(sd)
```

We use the example shown in Figure 6.1 to illustrate the Rule 6.2. This diagram has two lifelines x of block A and y of block B, hence, we define CML actions for each lifeline. As it does not have break, loop nor critical combined fragments, their respective auxiliary CML actions are not generated in this process. As illustrated in Figure 6.1, we define a CML action for the MessagesBuffer. All these actions are used in the main action, which is added after the @ character. We omit the generated CML for this action now because it is detailed further in Rule 6.9.

```
...
process sd_internal_example = val sd_id: ID, xA_id: ID, yB_id: ID @
begin
actions
        lf_xA = ...
        lf_yB = ...
        MessagesBuffer = ...
@ ...
end
...
```

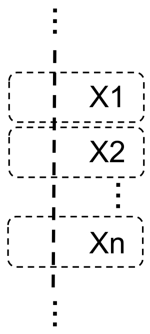
Next, we detail the translation of a lifeline.

6.2.1 Lifeline

A lifeline represents the execution of a sequence of events of the block it represents. The elements that appear along the lifeline vertical axis are the interaction fragments. Figure 6.2 illustrates a lifeline and its interaction fragments (X1, X2, ..., Xn). Interaction fragments can be message occurrences, combined fragments, interaction use and state invariants. These elements must occur in a top-down order along the vertical axis, however, this order can be diverted by the use of combined fragments like break.

Rule 6.3 shows how we represent a lifeline in CML. We sequentially compose each one of the possible occurrences in a lifeline. The interaction fragment related to the occurrence is translated by invoking the corresponding translation function: `t_message` for a message occurrence, `t_combined_fragment` for combined fragments, `t_state_invariant` for state invariants and `t_interaction_use` for interaction use elements. If we have break combined fragments among the sequence of interaction fragments translated, then we have to add the possibility of interruption, which is translated using the interruption operator of CML `/_\` and the event `interrupt`.

Figure 6.2: Illustration of a lifeline.



Source: Author's ownership.

Rule 6.3: t_lf_interaction_fragments

```
t_lf_interaction_fragments(ifs: seq of InteractionFragment, lf:
    Lifeline): action =
    "("
    for intf in seq ifs sep ";" do
       "("
       switch(intf.Type)
          case MessageType: t_message(intf)
          case CombinedFragmentType: t_combined_fragment(intf, lf)
          case StateInvariantType: t_state_invariant(intf, lf)
          case InteractionUseType: t_interaction_use(intf)
       end switch
       ")"
    end for
    ")"
    for intf in seq ifs do
       if intf.Type == CombinedFragmentType and intf.cfType == BREAK.Type
          then
          "/_\ interrupt."intf.index" -> Skip"
       end if
    end for
```

The next extract shows an example of the CML actions for the lifelines shown in Figure 6.1. Both have two main occurrences: the synchronous message m1 and the alternative combined fragment. The latter is divided in two operands with other message occurrences. Therefore, the corresponding CML action is divided in two groups one for each occurrence. Each group is enclosed by parentheses and they are sequentially composed. We use ellipses to represent the content of each occurrence because they are detailed in other rules.

```
...
lf_xA = ((...);(...))
```

```
lf_yB = ((...);(...))
...
```

6.2.2 Messages

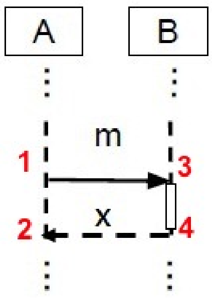
We provide the semantics for synchronous and asynchronous messages. Synchronous messages are operation calls and they are composed of two events: the call to the operation and the reply to the call. They are modelled exactly as described in Section 4.2 of Chapter 4 with the addition of the event type of the message (sending or receiving event). Thus, events with names suffixed by `_I` corresponds to the input of the operation call while events with names suffixed by `_O` represent the output of the call, in other words, the reply to the call. For simplicity, we consider that all operation calls have reply messages as they are synchronous.

Rule 6.4 shows the semantics for synchronous calls. If the message occurrence is a sending event, then we call function `t_statecopy` to retrieve all attributes of the block because they can be used as arguments of the message. Next, we generate the sending event `mOP.s` with similar values as described for operation events in Chapter 4. Each message in the interaction is indexed by a natural number (`mos.Message.index`). We use the auxiliary function `t_idMessageEnd` to provide the proper identifier value given the message end be either a lifeline or a gate. The latter case may happen when a message crosses an interactionUse element. In this case, the referred diagram has messages starting at the borders of the diagram and these points are called gates. For instance, the messages transmitPack() of the sequence diagram shown in Figure 2.10 on page 32 have gates as one of their message ends.

The sending event is followed by the receiving event `mOP.r` with a similar event structure. The receiving event is fired by another lifeline that replies to the call, hence, this lifeline receives the parameter `out`, which is the reply to the call. When the message occurrence is the receiving event, then the receiving lifeline just synchronises on the receiving event of the operation. Differently from the sender lifeline, the receiving lifeline can exchange messages before replying to the synchronous call. Note that this is the receiving event of the first part of a synchronous call, therefore, we use `mOP.r`. The reply call also has two events, one for sending the reply, which is shown in Rule 6.5, and another for receiving, which we presented as the second event in the if compartment of Rule 6.4.

Figure 6.3 illustrates a synchronous message from lifeline of a block A to lifeline of block B. Rule 6.4 covers the message ends 1, 2 and 3. While Rule 6.5 details the event at message end 4.

Figure 6.3: Illustration of a synchronous message.



Source: Author's ownership.

Rule 6.4: t_synch_call

```
t_synch_call(sd: Interaction,mos: MessageOccurrenceSpecification, sender,
    receiver: MessageEnd): action =
   if mos.isSendEvent() then
      t_statecopy(receiver.represents, mos.Message)
      name(receiver.represents)"_mOP.s."mos.Message.index"."
      t_idMessageEnd(sender,sd)"."t_idMessageEnd(receiver,sd)"!
      mk_"mos.Message.signature"_I(
      mk_token(\""mos.Message.signature"_I\")"
      sep "," mos.Message.Arguments") ->
          "name(receiver.represents)"_mOP.r?n."
      t_idMessageEnd(sender,sd)"."t_idMessageEnd(receiver,sd)"?out:(
      out.$id in set "name(receiver.represents)"_O and out.$id = mk_token(
      \""mos.Message.signature"_O\"))) -> Skip"
   else
      name(receiver.Lifeline.represents)"_mOP.r?n."
      t_idMessageEnd(sender,sd)"."t_idMessageEnd(receiver,sd)"?oper:(
      oper.$id in set "name(receiver.Lifeline.represents)"_I and
      oper.$id = mk_token(\""mos.Message.signature"_I\") -> Skip"
   end if
```

Rule 6.5 shows the event of sending the reply event from the receiver lifeline. This event uses the mID variable declared and assigned in Rule 6.4.

Rule 6.5: t_reply_call

```
t_reply_call(sd: Interaction,mos: MessageOccurrenceSpecification, sender,
    receiver: Lifeline): action =
   if mos.isSendEvent() then
      t_statecopy(receiver.represents, mos.Message)
      name(receiver.represents)"_mOP.s."mos.Message.index"."
      t_idMessageEnd(sender,sd)"."t_idMessageEnd(receiver,sd)"!
      mk_"mos.Message.signature"_O("
      mk_token(\""mos.Message.signature"_O\")"sep ","
      mos.Message.Arguments") -> Skip)"
   end if
```

The following extract shows the CML content of the synchronous message m1 depicted in Figure 6.1. Firstly, the lifeline `lf_xA` declares the event of sending message m1. Before declaring the event, it invokes function `t_statecopy` in order to get the values of attributes of block x. In this case, there is only one attribute (`i`). Next, it defines the event of sending message m1 to other lifeline `B_mOP.s.1.xA_id.yB_id!mk_m1_I(mk_token("m1_I"))`, where `s` is the mark for a sending event, `1` is the index of the message, `xA_id` is the identifier of the source block, `yB_id` is the identifier of the target block, and `mk_m1_I` creates a record type for the first event of a synchronous call m1 (`_I`). This event is followed by the receiving of the reply event `B_mOP.r?n.xA_id.yB_id?out`, where `out` is a record type whose field `$id` is part of the reply events of block B (`B_O`) and its content is `m1_O`. After these events, the lifeline has a combined fragment whose content is omitted for now (`(...)`).

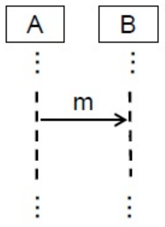
The CML action of the second lifeline `lf_yB` has the complementary behaviour regarding message m1. Its first events is the receiving (`.r`) of the synchronous call (`m1_I`) and its second event is the sending (`.s`) of the reply (`m1_O`). We also omit the content of the combined fragment for now.

```
...
lf_xA = ((
  (dcl x_i: int @ (A_get_i.xA_id^[mk_token("m1")].xA_id?x -> x_i := x);
  B_mOP.s.1.xA_id.yB_id!mk_m1_I(mk_token("m1_I")) ->
  B_mOP.r?n.xA_id.yB_id?out: (out.$id in set B_O and
  out.$id = mk_token("m1_O")) ) -> Skip);(...))
lf_yB = ((
  B_mOP.r?n.xA_id.yB_id?oper: (oper.$id in set B_I and
  oper.$id = mk_token("m1_I")) -> Skip;
  Skip;B_mOP.s.2.xA_id.yB_id!mk_m1_O(mk_token("m1_O")) -> Skip);
  (...))
...
```

Regarding asynchronous calls, they are modelled in CML in terms of signals. We use the corresponding signal event as described in Section 4.2. As explained earlier, while the sender lifeline is blocked waiting for the reply event for operation calls, it is ready to proceed to the next event of the lifeline after sending a signal.

Figure 6.4 shows an example of an asynchronous message. Events for sending and receiving an asynchronous message are detailed in Rule 6.6. The events are similar to the synchronous message. The channel used is `mSIG`, and the record types of the message are appended with `_S`.

Figure 6.4: Illustration of an asynchronous message.



Source: Author's ownership.

Rule 6.6: t_asynch_call

```
t_asynch_call(sd: Interaction,mos: MessageOccurrenceSpecification,
    sender, receiver: Lifeline): action =
   if mos.isSendEvent() then
      t_statecopy(receiver.represents, mos.Message)
      name(receiver.represents)"_mSIG.s."mos.Message.index"."
      t_idMessageEnd(sender,sd)"."t_idMessageEnd(receiver,sd)"!mk_"
      mos.Message.signature"_S(mk_token(\""mos.Message.signature"_S\")"
      sep "," mos.Message.Arguments") ) -> Skip"
   else
      name(receiver.represents)"_mSIG.r?n."t_idMessageEnd(sender,sd)"."
      t_idMessageEnd(receiver,sd)"?signal: (signal.$id in set "
      name(receiver.represents)"_S and signal.$id = mk_token(
      \""mos.Message.signature"_S\")) -> Skip"
   end if
```

The following extract shows the CML code of asynchronous message m2 shown in Figure 6.1 for both lifelines. Similarly to the operation, it first recovers the values of attributes, and then it communicates the sending event of the signal. The differences when compared to the events of the operation m1 are: the name of the channel `B_mSIG`, the index of the message (`3`), and the record type used (`mk_m2_S(mk_token("m2_S"))`), which corresponds to the signal event m2. Regarding the receiving event, now the channel only communicates events whose record types identifiers are in the set of signals of block B (`B_S`) and that match the name `m2_S`.

```
...
lf_xA = (...
        (dcl x_i: int @ (A_get_i.xA_id^[mk_token("m1")].xA_id?x ->
                x_i := x);
        B_mSIG.s.3.xA_id.yB_id!mk_m2_S(mk_token("m2_S")) -> Skip)
        ...)
lf_yB = (...
        B_mSIG.r?n.xA_id.yB_id?signal: (signal.$id in set B_S and
        signal.$id = mk_token("m2_S")) -> Skip
        ...)
```

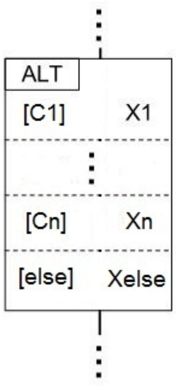
...

6.2.3 Combined fragments

Since UML 2, the flow of messages in a sequence diagram can be described using operators (called combined fragments) in order to express more complex traces of messages. Combined fragments can alter the interpretation of the flow according to the operator used. We provide the semantics of the following combined fragments: parallelism (PAR), alternatives (ALT), option (OPT), strict order (STRICT), loops (LOOP), break of flow (BREAK), and critical region (CRITICAL). To illustrate their semantics, we show the representation of the alternatives combined fragment. The remaining operators are discussed in the Appendix A.

Figure 6.5 shows a generic alternative combined fragment for a lifeline. Although combined fragments usually span over several lifelines, we translate each lifeline separately in terms of its occurrences, as shown in Rule 6.3. In this example, the operands of the fragment are X1, X2, ..., Xn and Xelse. Each operand can have message events and other fragments inside. The guards of the operands are C1, C2, ..., Cn, respectively. If none of these guards yields true, then the Xelse operand should be performed. The else guard is optional in this type of fragment. If none of the operands has a guard that evaluates to true, none of the operands are executed and the remainder of the enclosing fragment is executed.

Figure 6.5: Illustration of an alternative combined fragment for a lifeline.



Source: Author's ownership.

For operators that require the evaluation of guards (ALT, LOOP, OPT, BREAK), before the evaluation of the guard, each CML action of a lifeline involved in the fragment gets the value of attributes of the blocks involved in the diagram using function `t_statecopy` (Rule A.6), which uses the `get_` channels as described in Chapter 4. Also, they synchronise on these channels to make all of them get the same state at the same time. Otherwise, they could get these values in different times, which could result in an inconsistent evaluation because the values could change during this time lapse.

Rule 6.7 shows the translation functions for alternatives. They are translated as a nested if-then-else statement where each operand is an if statement. When the constraint is true, the operand is executed. When more than one constraint is true, the first one is executed. When the operands have no guards, a non-deterministic choice is performed in order to decide which operand should be executed.

Rule 6.7: t_alt_combined_fragment

```
t_alt_combined_fragment(alt: CombinedFragment, lf: Lifeline): action =
   if alt.OperandsHaveGuards then
      for operand in seq alt.Operands do)
         t_statecopyConstraint(operand.Constraint, alt)
      end for
      for operand in seq alt.Operands sep "else" do
         if operand.InteractionConstraint != "else" then
            "if "operand.InteractionConstraint.specification" then"
               t_lf_interaction_fragments(
               operand.InteractionFragmentsFromLifeline(lf), lf)
         else
               t_lf_interaction_fragments(
               operand.InteractionFragmentsFromLifeline(lf), lf)
         end if")"
      end for
   else
      "(" sep "|~|" {t_lf_interaction_fragments(
                  op.InteractionFragmentsFromLifeline(lf), lf)
                     |op in seq alt.Operands}")"
   end if
```

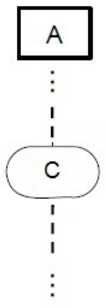
The following extract shows the CML generated from the alternative combined fragment displayed in Figure 6.1. Both lifelines get the same value of the attribute in order to evaluate the constraint of the combined fragment. We guarantee that they acquire the same value because they also synchronise on this event. Then, if the constraint yields true, then the behaviour referred for message m2 in each lifeline (as shown in the previous extract) should be performed, otherwise, the content for message m3 is executed.

```
...
lf_xA = (...
  (dcl x_i: int @ (A_get_i.xA_id^[mk_token("m1")].xA_id?x -> x_i := x);
  (if x_i <= 10 then
    (B_mSIG.s.3.xA_id.yB_id!mk_m2_S(mk_token("m2_S")) -> Skip)
   else
        (A_mSIG.r?n.yB_id.xA_id?signal: (signal.$id in set A_S and
         signal.$id = mk_token("m3_S")) -> Skip)
  ))
)
lf_yB = (...
  (dcl x_i: int @ (A_get_i.xA_id^[mk_token("m1")].xA_id?x -> x_i := x);
  (if x_i <= 10 then
    (B_mSIG.r?n.xA_id.yB_id?signal: (signal.$id in set B_S and
     signal.$id = mk_token("m2_S")) -> Skip)
   else
     ((Skip;A_mSIG.s.4.yB_id.xA_id!mk_m3_S(mk_token("m3_S")) -> Skip))
  ))
)
...
```

6.2.4 State invariant

Besides combined fragments, we also defined rules for state invariants, which are constraints evaluated along a lifeline that determines if the trace should be valid or not. These mechanisms can be used to verify properties of the system along a scenario. Figure 6.6 illustrates a state invariant along a lifeline. Unlike combined fragments, state invariants are linked to one lifeline. The semantics of this construct states that when the flow of events arrives in the state invariant, its constraint is evaluated. In case it yields false, the current trace should be considered invalid, otherwise, nothing happens.

Figure 6.6: Illustration of a state invariant.



Source: Author's ownership.

Rule 6.8 details the translation of state invariants. As a constraint needs to be evaluated, the `t_statecopy` function is invoked to collect the values of the attributes used in the constraint. If the constraint of a state invariant yields false, a synchronisation on the extra channel **`inv`**, which is used specifically in our semantics for sequence diagrams, marks an invalid scenario. This notion of marking a scenario that is not valid is important for validation purposes. Traces with such an event cannot be refined as it is an specific event of this type of diagram and does not exist in the traces of the system model. More details regarding this verification strategy and examples of application of this constructor are detailed in Chapter 8.

Rule 6.8: t_state_invariant

```
t_state_invariant(si: InteractionFragment, lf: Lifeline): action =
   t_statecopy(lf.represents, si)
   "if "si.InteractionConstraint.specification" then
      Skip)"
   "else
      inv.sd_id -> Skip)
```

6.2.5 Main action

Once all lifelines have been translated, they are composed in parallel in the main action of the sequence diagram. Rule 6.9 describes the content of the main action of a sequence diagram process. A sequence diagram starts with the communication of the event `beginInteraction`. Next, the flow of events is represented by a parallelism of the actions of the lifelines and the composition of this parallelism with the `MesssagesBuffer` action, which simulates the environment that the messages are transmitted. This action acts as a relay that receives a sending message event from a sender lifeline (`.s`) and communicates the corresponding receiving event to the receiver lifeline (`.r`). Other parallelism may exist according to the existence of break and critical combined fragments. `BREAK` and `CRITICAL` are auxiliary actions that may control the flow of messages according to the rules of their respective fragments. The former may interrupt the trace of a sequence diagram, while the latter may restrict parallel flows along a sequence diagram.

Rule 6.9: t_main_action

```
t_main_action(sd: Interaction): action =
"@"
"((( beginInteraction.sd_id -> (("define_alphabetised_parallel(
         {(t_lifeline_name(s), name(sd)"_cs_"t_lifeline_name(s))|
             s in set sd.Lifelines})
             ");;endInteraction.sd_id -> Skip [|{|endInteraction.sd_id"
             sep "," {name(block)"_mOP, "name(block)"_mSIG" |
                     block is lf.represents, lf in seq sd.Lifelines}
             " |}|] MessagesBuffer))"
if sd.existsBreakCombinedFragment() then
"[|{|break, interrupt, block, endInteraction|}| ] BREAK"
end if
")"
if sd.existsCriticalFragments() then
"[|"name(sd)"_Events|] CRITICAL"
end if
")\\ Hidden"
```

The next extract show the main action of the CML process for the sequence diagram displayed in Figure 6.1. It first communicates the event `beginInteraction` to signalise the beginning of the interaction. Next, it composes in alphabetised parallelism both lifelines. The alphabets of the lifelines `xA` and `yB` are detailed in the channel sets `example_cs_xA` and `example_cs_yB`, respectively. This parallelism is composed in parallel with the `MessagesBuffer` action, which controls the message exchanging between the lifelines. It simply synchronises with the lifelines on sending events and then communicates the respective receiving event for each message in the interaction. Finally, once the flow of events terminates, this action communicates the `endInteraction` event to represent the termination of the interaction. This process hides the channels that are used for internal control of the sequence diagram semantics. These channels are listed in the channel set `Hidden`.

```
...
process sd_internal_example = val sd_id: ID, xA_id: ID, yB_id: ID @
begin
actions
  lf_xA = ...
  lf_yB = ...
  MessagesBuffer = ...
@ ((( beginInteraction.sd_id ->
        ((lf_xA [{|example_cs_xA|} || {|example_cs_yB|}] lf_yB )
        [|{| B_mOP,B_mSIG,A_mOP,A_mSIG |}|]
          MessagesBuffer);endInteraction.sd_id -> Skip))) \\ Hidden
end
...
```

The rules for the remaining constructors of our semantics for sequence diagrams are presented in Appendix A.2. Next we present other formal semantic definitions for sequence diagrams and we compare them in terms of the number of constructs provided.

6.3 Related work

There are numerous works providing semantics for sequence diagrams; most of them are related to UML instead of SysML. As they have the same semantics, the work done for UML fits well for SysML sequence diagrams. Usually there are two kinds of approaches: (i) the definition of a semantic model to formalise diagrams (LI et al., 2004; SHEN et al., 2008), and (ii) the translation to an existing formalism such as Z, B, CSP, and Petri-Nets (EICHNER et al., 2005; RASCH; WEHRHEIM, 2005; DAN; DANNING, 2010). The main advantage of the latter is the existing tool support used to apply reasoning on the translations. Few related works allow the check of the consistency among diagrams. This is extremely relevant to us as one of our objectives is to provide such consistency verification across the various SysML diagrams.

Storrle presents an exhaustive work on formalising sequence diagrams using trace semantics (STORRLE, 2004b). Many constructs used in UML 2, including combined fragments, are covered. Storrle's semantics allows one to reason about refinement, concurrency and time restrictions. Some basic features are not covered such as Gates and arguments. Haugen et al. present another work that covers some of the Sequence Diagram elements we are interested in (HAUGEN et al., 2005). They also propose an approach based on a trace semantics in which refinement is used as a foundation for compositional analysis, verification and testing. Lund gives an operational semantics for the Haugen's denotational semantics (LUND, 2007). In both semantics, loop with constraints and the BREAK fragment are not covered.

Although the approaches by Storrle and by Haugen et al. do not use a semantic model similar to CML, they are inspiring works as they provide some important discussions over complete and partial traces, global versus local view, and so on.

Dan and Danning (DAN; DANNING, 2010) present an approach to semantic mapping specified using the language QVT (OMG, 2005) relations to CSP (HOARE, 1985). Their approach uses a notation similar to CML, so we could benefit from some of the ideas of their work. However, very few constructs of UML 2 are covered.

Cavarra and Bowles proposed a technique using Object Constraint Language (OCL) templates to express liveness properties in UML sequence diagrams (CAVARRA; BOWLES, 2005). They give examples showing that certain liveness properties cannot be expressed with assert or negate. Abstract state machines are used to enrich the sequence diagram in order to express such properties.

Cengarle et al. gives an operational semantics for sequence diagrams (CENGARLE et al., 2005). The authors define a semantic of negative fragments. Rules are given for each of the operators specifying whether a trace positively or negatively satisfies a fragment with that operator. The authors point out that with the basic interpretation of negative fragments it is easy to construct overspecified Interactions, i.e., an Interaction that can be positively and negatively satisfied from the same trace.

Knapp and Wuttke provide an operational semantics based on automaton (KNAPP; WUTTKE, 2007), while Eichener et al. use multivalued nets, which are a specific kind of Petri nets that allow parametrisation of messages and interactions (EICHNER et al., 2005). However, the latter does not provide enough information about the formalisation of some constructs. The intuitions are only described textually.

Shen et al. propose a formalisation using template semantics for UML 2 Sequence Diagrams (SHEN et al., 2008). The approach gives an operational semantics for which the basic computation model is hierarchical transition systems (HTS).

Most of these works differ regarding three main aspects: the number of constructions they cover, the semantics of constructions whose official meaning is vaguely defined, and the semantic domain used to formalise the semantics. The interesting aspect to notice is that when defining a semantics for sequence diagrams, some semantic decisions must be taken in

Table 6.1: Coverage comparison

	IU	GD	PA	ST	AL	OP	LP	BK	CR	SI	As	Gt
Storrle	✓	×	✓	✓	✓	✓	✓	✓	✓	×	✓	×
Haugen et al.	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓
Cavarra and Bowles	×	✓	✓	✓	✓	×	×	×	×	✓	✓	×
Dan and Danning	×	✓	×	✓	✓	✓	✓	×	×	×	✓	×
Cengarle	×	×	✓	✓	✓	✓	✓	×	×	×	✓	×
Knapp and Wuttke	×	✓	✓	✓	✓	✓	✓	×	×	✓	✓	×
Eichner et al.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Shen et al.	✓	✓	✓	×	✓	✓	✓	✓	✓	×	✓	×
Our work	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Source: Author's ownership.

order to allow its use. Micskei and Waeselynck have provided an excellent survey on these semantic choices (MICSKEI; WAESELYNCK, 2011). We developed our work according to their classification and categorisation of semantic meanings for sequence diagrams. Whenever the meaning of an operator is vaguely defined, we have chosen one that is more convenient for modelling and for checking diagram consistency. For instance, the choice on synchronisation before evaluating guards of combined fragments in order to avoid inconsistency, and the unique identification of the messages.

Table 6.1 presents a comparison (partially based on (MICSKEI; WAESELYNCK, 2011)) of the coverage of our formalisation with some of available literature. The left most column contains the features that we cover: interactionUse (IU), Guards (GD), the CombinedFragments for parallel (PA), strict sequencing (ST), alternatives (AL), option (OP), loop (LP), break (BK), critical region (CR). Moreover, other elements like state invariant (SI), asynchronous message (As) and gates (Gt) are compared for coverage. The ✓ indicates that the feature is covered by the work on the column, and × indicates it is not.

As far as we know none of these works aim to perform consistency verification among structural and behavioural diagrams.

As for the other diagrams, we propose a set of rules covering a large number of constructs in order to provide to users a language expressive enough for creating their models. The remaining rules for sequence diagrams are presented in Section A.2 of Appendix A.

7 Model Integration

Whilst a SysML model can be visualised via diagrams, its actual representation is a set of interconnected elements that respect the metamodel specified in by the OMG (OMG, 2012, 2011). For instance, a block definition diagram is not stored explicitly in the SysML model; it is just a means of declaring the blocks that form the model. Furthermore, different diagrams may contribute to the same model element. For example, if a block definition diagram introduces a block `B` with a property `n`, and an internal block diagram adds a port `p` to the block `B`, in the model, `B` contains both the property `n` and the port `p`.

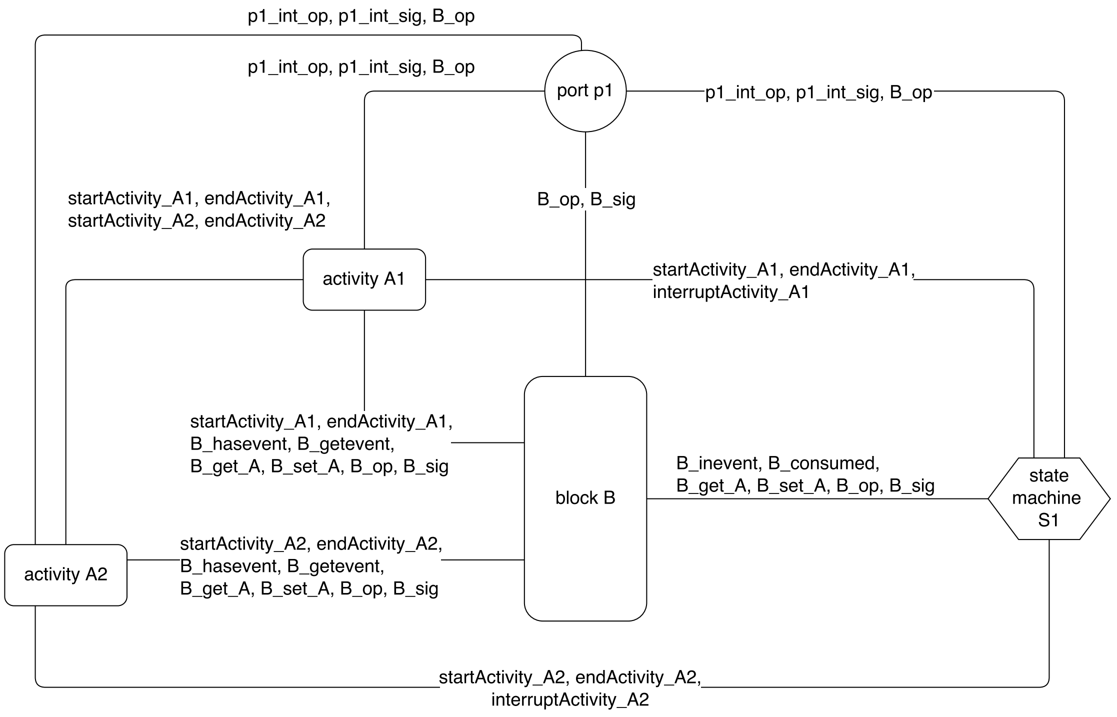
Accordingly, a SysML model that follows our guidelines is, essentially, a collection of blocks, state machines, activities and interactions (visualised by sequence diagrams). The parts of composite blocks are structured using connectors, simple blocks have state machines specifying their behaviours, operations are specified either by the block's state machine or by an associated activity, and activities and state machines use CML data operations.

Typically, a model contains several simple and composite blocks as components, and the system as a whole is modelled by a block. Since the CML construct used to represent systems and their components is a process, then blocks, state machines, activities and ports are modelled as CML processes.

The services defined by a SysML model are the operations, signals and public attributes of its blocks. These are all represented in CML by channels that model the system communication with the environment.

In this section, we present our approach to define the CML model that corresponds to a SysML model integrating several elements as described above. We first address in Section 7.1 the case of a system defined just by one simple block; we call such basic models non-hierarchical. The abstract model of the leadership election protocol is an example of a non-hierarchical model. Models containing several simple blocks can be handled by translating each block (and its associated diagrams) in isolation. Next, we explain how our approach extends to consider the integration of diagrams in hierarchical models, which include composite (as well as simple) blocks. The modelling approach in this case builds on that for non-hierarchical models. Finally, in Section 7.3, we discuss the models related to sequence diagrams, and how they define properties

Figure 7.1: The integration points of the semantics of a single block.


p1_int_op, p1_int_sig, B_op
p1_int_op, p1_int_sig, B_op
port p1
p1_int_op, p1_int_sig, B_op
startActivity_A1, endActivity_A1,
startActivity_A2, endActivity_A2
B_op, B_sig
activity A1
startActivity_A1, endActivity_A1,
interruptActivity_A1
startActivity_A1, endActivity_A1,
B_hasevent, B_getevent,
B_get_A, B_set_A, B_op, B_sig
B_inevent, B_consumed,
B_get_A, B_set_A, B_op, B_sig
state machine S1
block B
startActivity_A2, endActivity_A2,
B_hasevent, B_getevent,
B_get_A, B_set_A, B_op, B_sig
activity A2
startActivity_A2, endActivity_A2,
interruptActivity_A2

Source: Author's ownership.

of the overall model of a system.

7.1 Non-hierarchical models

Figure 7.1 gives an overview of the CML models that we use to capture the semantics of SysML models containing a single simple block. They are defined by a composition of interacting processes, each of which models the individual elements of the SysML model.

Figure 7.1 depicts these processes as nodes. The required cooperation between them is depicted as lines. The complete model is defined by the parallel composition of the processes, which synchronise on the channels indicated in Figure 7.1 as the labels of the lines. This modelling strategy allows diagrams to be modelled and analysed independently.

The services of a SysML block are characterised by its operations, signals, and attributes. The interface of these services are represented in CML by channels, which in turn define the interface of the process that models the block. In the previous chapters we presented channels that represent operation calls, signal transmissions, and get and set operations that allow access to attributes. We also introduced other channels that support the evaluation by state machines and activities of a SysML event, which can be related to an operation call or a signal reception.

A state machine accepts requests to process operations and signals of its block. So, the interface of the CML process of a state machine includes two types of channels: a channel that accepts a request to react to events corresponding to an operation call or a signal reception and another to provide responses regarding the realisation of the event. For instance, in the CML model of the example displayed in Figure 2.5 (on page 27), the communications between the Device block and its state machine happen through the channels `Device_inevent` and `Device_consumed`. In general, as shown in Figure 7.1, a channel `B_inevent` is used by the block process `B` to send an event to be processed by the state machine, and `B_consumed` is used by the state-machine process to indicate the result of evaluating such an event.

Additionally, in order to handle these SysML events, the state machine may access the block's attributes. This interaction between a block B and a state machine S1 is represented by the channels `B_get_A` and `B_set_A` shown in Figure 7.1. They allow the state machine to read from and write to the attribute A of B. For example, in Figure 2.5 (on page 27) the Device state machine accesses the block's attributes in the action petition:=petition-1. In the corresponding CML process, these attribute accesses are carried out through the channels `Device_get_petition` and `Device_set_petition`.

A state machine can also call operations and send signals to its block. This is achieved via the channels `B_op` and `B_sig` shown in Figure 7.1 for communication between B and S1. In general, these channels can be used by the environment to request the services of a block. Such services can also be requested by other blocks, state machines, or activities.

We observe that the channels detailed so far are in the interface of both the block and the state machine processes. This ensures that when they are composed in parallel, the block process can send events to be treated by the state machine process and, similarly, the state machine can request services of the block.

The interface of a CML process that models an activity contains channels that can be used to start the execution of the activity, interrupt it, signal its termination, call other activities, wait for another activity termination, access block operations, signals and attributes, and check the occurrence of an event in a block. For instance, the state machine LeadershipElection in Figure 2.5 (on page 27) calls the activity ActBroadcast with Leader as a parameter. This is modelled by communications on the channels `startActivity_ActBroadcast`, for initialisation, `interruptActivity_ActBroadcast`, for interruption, and `endActivity_ActBroadcast`, for termination.

Activities can also interact with a block to access its attributes, send signals, and call operations. In CML, these interactions take place using the same channels used by state machines as described above. As shown in Figure 7.1, the process for an activity A1 interacts with a block B via the channels `B_hasevent` and `B_getevent` to search and obtain events from the block. We remind that these events are used by the activity to verify the availability of a certain event in the event pool of the block and to consume an event of the pool, respectively. The channels `B_op` and `B_sig` access operations and signals of the block, `B_get_A` and `B_set_A` access an attribute A, and `startActivity_A1` and `endActivity_A1` allow the block to call the activity and wait for its termination. Additionally, an activity may call another activity A2 through the channels `startActivity_A2`, `endActivity_A2` as defined in Chapter 5 by using the call behaviour action constructor. Another possibility is the activity A1 starting the activity A2 and the block B starting A2 too. This is feasible because the block B and activity A1 manipulate different instances of the process for activity A2.

The interface of a port p1 has channels that allow sending and receiving operation calls and signals. These channels are divided in two sets, which we call internal interface and external interface. The channels in the internal interface are used to interact with the block that contains the port as well as with its parts, state machines and activities. The names of the channels in the internal interface are of the form `p1_int_op` and `p1_int_sig` as shown in Figure 7.1. The external interface allows other blocks to interact with the port via their own ports. It contains channels named `p1_ext_op` and `p1_ext_sig`, which are omitted in Figure 7.1, and further illustrated in Figure 7.2.

Note that, in the interface of a block `B`, the names of the channels used to model events corresponding to operation calls and signals are of the form `B_op` and `B_sig`. In the case of a port, they are `p1_int_op` and `p1_int_sig`. We use different names because a port can restrict the operations and signals that a block may accept or require. For instance, in Figure 2.4 (on page 26) the port pD communicates only calls to operations and signals described in the interfaces DeviceInterface and BusInterface. So, the CML process that models pD only includes events that correspond to these calls.

On the other hand, in the composition of processes for a port and its block, events corresponding to the same operation calls or the same signals need to be identified. For this purpose, we use CML renaming, which allows the renaming of the channels used in a process or action. For instance, the action `c?x -> **Skip**` waits for a value on the channel `c` and terminates, while the renamed action `(c?x -> **Skip**)[[c <- a, c <- c]]` is equivalent to `(a?x -> **Skip** [] c?x -> **Skip**)`, that is, it waits for a value on either the channel `c` or `a` before terminating. The renaming applied to ports of simple blocks is described in Section 4.5 in Chapter 4. This operator is used to rename the channels `p1_int_op` and `p1_int_sig` to `B_op` and `B_sig` to restrict the possible communications through these channels whenever p1 is realising a subset of all SysML interfaces. This mechanism allows the block and the port processes to interact. Figure 2.3 (on page 26) illustrates that a Device block has a pD port. Hence, the port channels `pD_int_op` and `pD_int_sig` are renamed to `Device_op` and `Device_sig` restricting the events that should be transmitted through the port pD, which corresponds only to the operations and signals described in the DeviceInterface and BusInterface.

The interaction between two ports is also modelled using renaming, but in this case the channels in the external interfaces of the ports are renamed to new channels that model the connector between the ports. This is discussed in the next section.

Finally, the interaction between a port p1, state machines, and activities take place through the channels `p1_int_op`, `p1_int_sig` and `B_op`, where B is the block that contains the port p1. The first two channels are used for the state machines and activities to call operations and send signals via the port, and the third channel is used to indicate the completion of an operation call.

Some of the presented channels are just for internal communication, that is, they are not relevant for the external environment of a block. For example, a block should not access the state machine or the activities of another block directly. The hiding operator of CML is used to make these channels internal. We write `A \\ {|e|}` to make `e` not visible externally from `A`. In Figure 7.1, the channels `B_inevent` and `B_consumed`, related to state-machines, the channels `B_hasevent`, `B_getevent`, `startActivity_A1`, `interruptActivity_A1`, and `endActivity_A1`, related to the activity `A1`, the similar channels for the activity `A2`, and the channels `p1_int_op` and `p1_int_sig`, related to the port `p1`, are hidden from the environment of `B`. Finally, any channels `B_get` and `B_set` related to private attributes of B are also hidden, as are the communications on `B_get` and `B_set` channels that are related to public attributes, but whose source and target are elements of the diagram themselves. These communications correspond to internal uses (via accesses and assignments) of the attributes of a block, rather than external observations described, for example, in a sequence diagram.

Conversely, the services of a block are offered by the remaining channels, which correspond to signals, operations, attributes with public visibility, and the external interface of ports. In the example of Figure 3.5 (on page 49), they are `LE_SoS_get_Active`, `LE_SoS_set_Active`, `LE_SoS_get_Elected`, `LE_SoS_set_Elected`, `LE_SoS_op`, and `LE_SoS_sig`. In general, for a block `B` like the one in Figure 7.1, the services are provided by the channels `B_op`, `B_sig`, `p1_ext_op`, `p1_ext_sig`, and, for attributes of B that have public visibility, we can use the channels `B_get_A` and `B_set_A`, where `A` is the name of the attribute.

These channels communicate at least three pieces of control data relevant to the interactions. Two of them are the identifiers of the sender and the receiver of the interaction. As described earlier, they have the type `ID`. For instance, in the leadership election example, we have three instances of the Device block interacting with one instance of a Bus block, as depicted in Figure 2.4 (on page 2.4), and each of the devices can send transmitPack signals to the bus. Each channel `Bus_sig` that communicates the transmitPack signal carries the information of which device sent the signal and the target of the message, which is the instance of the block Bus.

This control information is important for several reasons. It is used to ensure the correct communication between the CML processes for block instances, which run in parallel and synchronise on these channels. Additionally, with the control information in the communications, the traces of the CML processes represent accurately communications between the different block instances of a system. Finally, it is used to ensure that the reply of an operation call is returned to the specific block process that requested it. The same call can be sent by different elements of behaviour (state machine or activity) in a block. For example, the block B from Figure 7.1 can call an operation of another block through its state machine S1 or through its activity A1. The event corresponding to the return of this operation call must be sent exactly to the element of behaviour that requested it to allow it to continue. This is achieved in CML by treating an identifier as a sequence of tokens that describes the position in the SysML model hierarchy of the element that originated the call. Thus, when the state machine S1 process makes the call, the identifier of the sender is `[mk_token("B"),mk_token("stm")]`, where `B` is the token representing an instance of the block B and `stm` represents the state machine of B. Alternatively, when the call comes from the activity process A1, the identifier is `[mk_token("B"),mk_token("acts"),mk_token("A1")]`, where `acts` means that it comes from an activity and `A1` is the activity that originated the call. We do not need to specify a specific identifier for the state machine because the simplification guideline 1 on page 44 requires that each block has at most one state machine.

A third piece of control data is needed when the same interaction occurs concurrently inside the same element of behaviour: for instance, two calls to the same operation of the same target, each originating from two different parallel regions (r1 and r2) of a state machine S1. In order to differentiate the calls, each channel communicates a unique index of the call represented by a natural number. Therefore, the communication corresponding to the call from r1 has a natural number index, while for the call from r2 we use a different index. These indices avoid that the reply to the call from r1 is delivered to the r2 action and vice-versa. This third piece of control data is not relevant for the environment of the block process that represents the entire system. The index of an interaction must be part of an internal protocol to ensure the correct communication between processes and actions. The top-level block that represents the system, therefore, exposes a different version of the channels without this index. This is achieved through the use of the renaming operator of CML. For instance, the LE SOS (Figure 3.5 on page 49) block process has its operations renamed by `LE_SOS[[LE_SOS_op.m <- LE_SOS_OP | m: nat]]` to define the system model. The communications `LE_SOS_op.m`, where `m` is a natural number representing the index that identifies the communication context, are renamed to `LE_SOS_OP`, which is a version of `LE_SOS_op` without the index. The same strategy is applied to signal channels.

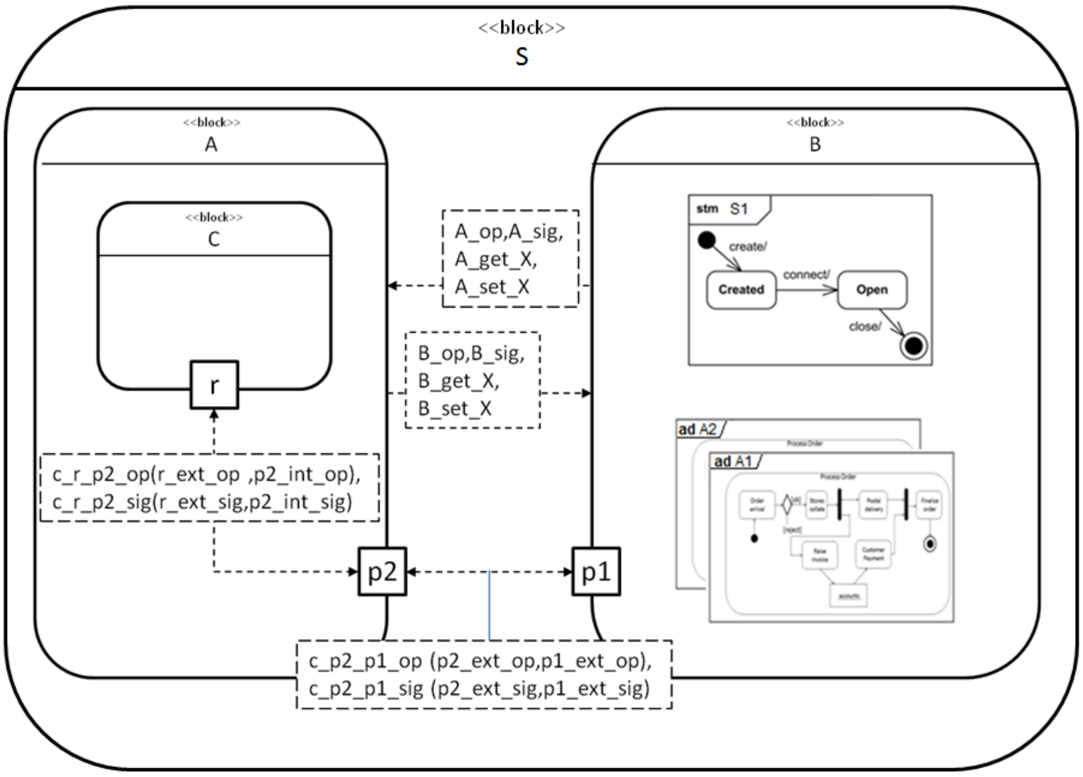
7.2 Hierarchical models

We now address blocks that are structured in hierarchies. Figure 7.2 depicts the CML processes that model such blocks, and their ports, activities and state machines. In this figure, processes are represented by solid boxes, and the sets of channels that allow them to interact are included in dashed boxes associated with arrows connecting the relevant processes.

In Figure 7.2, the block B is simple (and its CML model has the structure illustrated in Figure 7.1), A is composite and contains a block C as a part, and S, which is the root block of the model, is also composite and contains both A and B. The SysML model to which it belongs is, therefore, hierarchical. The block C may also have a state machine and activities, but C can only communicate with B through the interface of block A. Although it is not shown in this example, our semantics caters for models with hierarchies of any depth; for example, C could have other blocks as its parts.

As indicated in the previous section, a block such as Device in Figure 2.3 (on page 26)

Figure 7.2: Overview of the communication between the models of multiple blocks.



Source: Author's ownership.

interacts with the block Bus through operations, signals and attributes. These interactions are presented in the CML model by communications on the channels `Bus_op` and `Bus_sig`, `Bus_get_packs` and `Bus_set_packs`. The first is used to call the operations of the Bus, the second to send signals to Bus, and the third and fourth channels are used to read and write values to the attribute packs of Bus.

In general, as shown in Figure 7.2, processes for blocks can communicate through channels `A_op`, `A_sig`, `A_get_X`, `A_set_X`, `p_ext_op` and `p_ext_sig`, where `A` is the name of the block, `X` is the name of an attribute of `A`, and `p` is the name of a port of `A`. As explained in the previous section, for simple blocks, the channels `p_int_op` and `p_int_sig` corresponding to a port `p` are renamed to model communication between the block and its port.

In the case of a composite block, a port is used for communication with its sub-blocks rather than with state machines or activities as described in Section 4.6. We recall that our guidelines require that composite blocks do not have state machines and activities. So, in this case, the channels `r_ext_op` and `p2_int_op` used by the processes for the ports r and p2 in Figure 7.2 are renamed to `c_r_p2_op`, and similarly the `r_ext_sig` and `p2_int_sig` channels are renamed to `c_r_p2_sig`. By using this strategy, the block process `A` can relay a message received by the port process `p2` by sending it to the port process `r` and vice-versa.

The same happens between blocks A and B. When they communicate with each other using ports p2 and p1, the channels `p2_ext_op` and `p1_ext_op` are renamed to `c_p2_p1_op`, and channels `p2_ext_sig` and `p1_ext_sig` are renamed to `c_p2_p1_sig`. In this case, the renaming is performed by the block S that owns A and B.

Finally, the renaming applied to operation and signal channels to hide the control data used to identify internal calls is applied to the root block of the model. However, for composite blocks, this renaming is applied for every channel in the hierarchy of the root block process that is part of the interface of the system. Therefore, for the model of Figure 7.2, it is defined as `S[[A_op.m <- A_OP, B_op.m <- B_OP,... | m: nat]]`.

7.3 Scenarios

According to our guidelines described in Section 3, sequence diagrams are used to validate the system.

Since a SysML model is a composition of several elements, to maintain consistency in an integrated view is not an easy task. In addition, finding undesired behaviours is error-prone due to the lack of tools that check consistency between these elements. To address this issue, our guidelines cater for analysis of the SysML model against scenarios described by sequence diagrams. By using the CML models of both the SysML system model and of a sequence diagram, we can compare the traces specified by the sequence diagram against those of the system model through refinement.

To perform this verification, we must relate the messages depicted in sequence diagrams to the services of the SysML system model. As said before, these services are the operations and signals of the blocks, which are represented in CML by communications on channels `B_OP` and `B_SIG`, where `B` is a block of the model. Similarly, a sequence diagram defines scenarios in terms of the messages exchanged by the blocks, which correspond to operation calls and signals. Hence, we can relate the events of sequence diagrams to the services of the SysML system model.

The messages of a sequence diagram are modelled in CML by communications similar to those used in the CML specification of a SysML system model. They are represented by the channels `B_mOP` and `B_mSIG`, which need to be renamed to `B_OP` and `B_SIG` by following the same strategy presented earlier for the system model. An additional channel **`inv`** is used to identify invalid traces of scenarios defined by a sequence diagram. This channel is present in the interface of the CML model of a sequence diagram, but not in that of the CML system model.

The channels used to represent messages communicate the identifiers of the source and the target of the messages, an index to identify a message in the diagram, and an extra tag to differentiate the communication corresponding to the point where the message is sent (tag `s`) from that corresponding to the point where it is received (tag `r`). The definition of the CML process for a sequence diagram, like that for a system model, uses renaming to eliminate the index and the tag. Since indices and tags are used for internal control, with different purposes in the models, they are not relevant when we compare these models. In this case, we have two events that must become one, because both the sending and receiving event is represented by a single event in the system model. For that we choose the receiving event as the one that is renamed, while the sending event is hidden in the CML model of the sequence diagram. For example, the CML process for the sequence diagram in Figure 2.8 (on page 31) is defined by renaming as `sd_LeadershipElection[[Bus_mSIG.m.r <- Bus_SIG | m: nat ]]`, where `m` is a natural number used to represent the index of a message in the diagram. One possible concern is the introduction of divergence when we hide this event. A divergence happens when a process enters in a infinite sequence of hidden events. However, in this case, there are always two consecutive events where one of them is always visible, that is, the hidden event could not recurse or enter in an infinity sequence of hidden events because it is always followed by a visible event.

This strategy allows the CML specification of both the system model and the sequence diagram model to use the same representation for operation calls and signals, and, thus, be compared. If a trace from a sequence diagram is not in the set of traces of the system model, then, either we have a flaw in the system model as it cannot perform a scenario that should be valid, or the sequence diagram has not been written properly. The refinement-based analysis strategy for comparison is further discussed in Chapter 8.

This section has provided an integrated view of how the CML model of the different elements (blocks, activities, state machines and interactions) of the SysML model relate to each other. This uses the individual semantics provided for each element discussed in chapters 4, 5 and 6.

7.4 Related Work

In this section we discuss approaches that propose integrated semantics with two or more model elements that can interact aiming one common goal.

Breu *et al.* have proposed a formal language called System Model to specify object-oriented systems in the style of UML (BREU et al., 1997). A System Model specification has a pre-defined mathematical structure comprising object identifiers, message passing, behaviour, communication histories, states, and so on. A UML diagram is modelled as a projection of a System Model, which is regarded as a complete and unified model of the entire system. Class diagrams, state-machine diagrams and sequence diagrams can be translated to a System Model. On the other hand, although the semantics of these diagrams is defined in a single formalism, the verification of the consistency among the diagrams and the development of tools has not been reported.

The project Precise UML (pUML) started the development of a precise semantic model for UML diagrams. Lano and Evans have proposed a systematic development process using UML and a mix of syntactic checks and formal verification for consistency, enhancements and refinements among class diagrams, state machine, sequence, and collaboration diagrams (LANO;

EVANS, 1999). Modelling and verification are carried out by hand, using first-order set theory. No general translation strategy like the one presented here has been developed.

Kuske *et al.* have proposed an integrated semantics for UML class, object, and state-machine diagrams using graph transformation (KUSKE et al., 2002). A state machine is modelled as a transformation of an object diagram. The integrated semantics allows us to visualise the evolution of a particular object diagram with respect to the state machine. This is, however, the only consistency check supported. A consistency check similar to ours, based on sequence diagrams, is proposed as future work.

Baresi *et al.* propose a formal semantics based on a metric temporal logic for UML class and object diagrams, state machine diagram, sequence diagrams and interaction overview diagrams (BARESI et al., 2012). The formal models are used to analyse satisfiability of the model and to check properties of the system. Sequence diagrams are not used to describe valid or invalid scenarios. They represent behaviours of the system and they are composed in interaction overview diagrams. The semantic domain used is considerable different from ours, for instance, every object must have an associated clock and the events are linked to *tick* events of this clock. Moreover, the semantics of sequence diagrams does not seem to follow the OMG standard specification (OMG, 2011) based on traces because some events may happen at the same time. Another difference from our work is that, while our focus is SysML, their focus is on UML diagrams.

Rasch and Wehrheim (RASCH; WEHRHEIM, 2003) have presented an extended UML class diagram in which the body of a method, its pre and postconditions, and the initialisation of an attribute are specified in Object-Z (SMITH, 2000). An integrated semantics in CSP is proposed for these extended diagrams and a subset of state-machine diagrams. Five notions of consistency are used: at least one trace does not deadlock (satisfiability); the model is deadlock free (basic consistency); every method is called at least once (executability); and every method becomes enabled infinitely often (availability). We do not consider all these checks, but they can be done in CML. Additionally, our approach does not extend the SysML diagrams, and so hides the formalism from modellers. In particular, bodies of methods are specified by activity diagrams and state machines.

Davies and Crichton have also proposed a formal semantics in CSP for UML class, object, state machine, sequence and collaboration diagrams (DAVIES; CRICHTON, 2003). Both inter and intra-object concurrency are addressed. Inter-object concurrency allows objects to run in parallel, while intra-object concurrency allows concurrent calls of operations of a single object. The semantics is used to verify both refinements and the consistency between sequence and the remaining diagrams. The translation from UML to CSP has been illustrated via examples. No general transformation rules are introduced.

A semantics for SysML is proposed by Hamilton *et al.* in terms of axioms of the Universal Systems Language 001AXES (HAMILTON et al., 2007). This language is based on a set of axioms and rules for applying function mappings and type mappings. Three primitive structures specify a mapping via sequential and parallel composition, and choice. Hamilton *et al.* have provided examples of how to model block, internal block, parametric and activity diagrams. No formal verification is proposed in this work; the focus is on the benefits of using a formal semantics to prevent the introduction of bugs.

Graves and Bijan have proposed a semantics for SysML block and internal block diagrams using a knowledge-based model for UML class diagrams (GRAVES; BIJAN, 2011b; GRAVES, 2012, 2011). Both diagrams are formally specified using the logic ALCQI (BERARDI et al., 2005); this encoding is proved to capture the part-decomposition relation correctly as a tree-like structure. No other diagrams of SysML have been formalised in this work.

Café *et al.* (CAFÉ et al., 2013) have proposed a semantics for SysML block, internal block, and state machine diagrams as a SystemC-AMS (VACHOUX et al., 2003) program, an extension of SystemC (PANDA, 2001) for heterogeneous systems. Their main contribution is an interpretation of SysML diagrams for systems that mix continuous and discrete signals. Once the SysML diagrams are translated to SystemC-AMS, they can be simulated on standard tools. No formal verification is employed in that work.

Broy *et al.* propose one of the first foundational semantics for a subset of UML2, which is called the *system model* (BROY et al., 2006, 2007a,b). It is defined in terms of state machines that describe the behaviour of objects and their data structures. The system model is formalised using mathematical theories instead of existing formalisms. It is claimed that the semantics of any UML model can be represented in terms of the system model. Classes, actions and activities are mapped to the system model representation. Although the approach is significant in providing a unambiguous UML semantics, it lacks automatic support for analysing the models.

The fUML standard (OMG, 2013) provides a precise semantic for UML classes, activities and actions, and an extension to fUML is being developed to cover composite structures (OMG, 2014). It has an executable semantics described in pseudo Java-code, formally defined using PSL (Process Specification Language) (GRÜNINGER; MENZEL, 2003), an axiomatic foundational language. Despite providing a reliable semantics for a subset of UML, fUML lacks tools for formal reasoning. Some works have proposed transformations to other formal languages to enable analysis (ABDELHALIM et al., 2012; LAURENT et al., 2014), and fUML provides a basis for validation of these transformations. Besides the constructs covered by fUML, our work considers state machines and interactions.

Table 7.1 summarises the works described above. The columns bdd, obj, coll, ibd, stm, act, sd and par refer to block-definition or class, object, collaboration, internal block or composite structure, state machine, activity, sequence and parametric diagrams. A tick ✓ indicates that the corresponding diagram is formalised by the work of the authors named. Parametric diagrams are not available in UML, while object and collaboration diagrams are not used in SysML, so their coverage is not applicable (N/A) in some works. The purposes of the formalisations are classified as Spec (specification), Cons (consistency), Ref (refinement), Well-Form (well-formedness), and Sim (simulation).

Table 7.1: Summary of related works.

	Diagram										
	bdd	obj	iod	coll	ibd	stm	act	sd	par	Formalism	Purpose
Breu *et al.*	✓					✓		✓	N/A	System Model	Spec
Lano and Evans	✓			✓		✓			N/A	Set Theory	Cons, Ref
Kuske *et al.*	✓	✓				✓			N/A	Graph Theory	Cons
Baresi *et al.*	✓	✓	✓			✓		✓	N/A	Graph Theory	Cons
Rasch and Wehrheim	✓					✓			N/A	CSP	Cons
Davies and Crichton	✓	✓		✓		✓		✓	N/A	CSP	Cons, Ref
Hamilton *et al.*	✓	N/A	N/A	N/A	✓		✓		✓	001AXES	Spec
Graves	✓	N/A	N/A	N/A	✓					Description Logic	Well-Form
Café *et al.*	✓	N/A	N/A	N/A	✓	✓				SystemC-AMS	Sim
Broy *et al.*	✓					✓			N/A	Set Theory	Well-Form
fUML + PSCS	✓				✓		✓			PSL	Well-Form
Our work	✓	N/A	N/A	N/A	✓	✓	✓	✓		CML	Cons, Ref

Source: Author's ownership.

Our work is distinctive in its definition of an integrated semantics for a substantial subset of SysML. We have a comprehensive result in terms of both the amount of diagrams covered (five diagrams — tied with Davies and Crichton (DAVIES; CRICHTON, 2003) and Baresi *et al.* (BARESI et al., 2012)) and the amount of constructors covered per diagram. Our semantics cover 10 block-definition diagram constructs, 12 state-machine constructs, 21 activity-diagram constructs, and 17 sequence-diagram constructs. Only (LILIUS; PALTOR, 1999; MENG et al., 2004; EICHNER et al., 2005) cover as many constructs as we do per diagram and none, to our knowledge, covers more constructs than we do. Moreover, our semantics is mechanised, that is, it is implemented in a tool in order to generate CML specifications from the SysML model automatically, and can be used as part of a tool set for reasoning about SysML models.

8 Model Analysis and Validation of the semantics

This chapter describes applications of our semantics and some validations to assess its soundness. One of the benefits of describing a semantics in terms of a language based on a process algebra like CML is that we can apply the techniques for analysing specifications that are available for these languages, as analysis of deadlock-freedom, absence of non-determinism, and refinement of specifications. Such techniques can be applied to any of the specifications generated from the SysML models once a refinement notion is provided.

In order to perform an analysis, we need to derive formal specifications from the SysML models according to our semantics. Executing this task without tool support, besides being extremely time and effort consuming, is error-prone. To overcome this issue, most of the translation process has been mechanised. The automatic derivation of CML specifications from activities, blocks and state machines is implemented as a plug-in for the Atego's Artisan Studio (ATEGO, 2013) in the context of the COMPASS project. Thus, if designers create SysML models according to our guidelines of usage (see Section 3.1), they can automatically generate the corresponding CML specifications. From these specifications several methods can be applied to improve the Verification and Validation (V&V) process. We did not implement the plug-in, but we delivered the rules to the Atego team to implement them and we had regular meetings to discuss any issues that arose during this implementation.

As an example of an application for refinement, during the development of a SysML model, such a model changes to capture different levels of abstraction in which the system is described, and thus it is important to guarantee that an up-to-date version of the model captures the same behaviour of the previous version, that is, that the traces of the current model conforms to the same traces of the older model. This validation can be assessed through the use of refinement because a specification P is refined by another specification Q with respect to traces ($P \sqsubseteq_T Q$), if, and only if, the set of traces of Q are inside the set of traces of P. Assuming that Q is the specification of the current version of the SysML model (given that the newly introduced behaviours are hidden) and P is the specification of the older version of the SysML model, we could define a refinement notion using our semantics based on the refinement at the CML level. Note that this analysis can be performed at several levels of granularity, for instance, by considering the whole SysML model or just parts of the model, that is, a portion described by a specific diagram. Considering the latter, we could verify, for instance, that an activity or a state machine evolved as expected. In addition, due to the compositionality feature of our semantics, the refinement of a specific model element, like an activity or a state machine, would imply in a refinement of the whole model as well.

We can also check consistency between different diagrams used to describe the model. As a diagram interacts through synchronisation on specific channels, we can check if these communications are valid in all diagrams involved, in other words, we can check consistency of information used in different diagrams.

These applications of refinement are a byproduct of using a process algebra as the semantic domain for describing SysML models. Nevertheless, the benefits are not restricted to this kind of analysis. Another mechanism that can be applied to support the validation of the models is animation. By animating the CML specification, users can perceive what events are available at a given time and emulate them. This simple mechanism can help designers to identify the undesirable absence or availability of events.

In addition to the previous validation mechanisms, we propose a methodology based on model checking techniques to analyse the properties and the behaviour of the system. We verify if the traces of a system modelled using blocks, activities and state machines are consistent with the traces described by interactions designed in terms of sequence diagrams. Properties can be verified by the use of the state invariant construct presented in Chapter 2.

Therefore, this chapter discusses these applications with the support of the leadership election and dwarf signal examples presented in Chapter 3. Section 8.1 discusses how CML specifications can be automatic derived from SysML models. Once the CML specification is provided, some pre-processing on the model is necessary in order to allow animation and model checking of the model. This process is presented in Appendix B. We show how models can be animated in Section 8.2. Section 8.3 details our approach for analysing SysML models using refinement. Section 8.4 presents some validations we have performed to show the consistency of our semantics. Section 8.5 discusses the work related to analysis of UML/SysML models. Finally, in Section 8.6 we summarise the discussions and contributions of this chapter.

8.1 Mechanisation of the Translation Process

To enable automatic generation of CML models, we have supported the development of a model-to-text transformation plug-in for Atego's Artisan Studio. This tool produces code written in various languages using the Automatic Code Synchronizer (ACS) [1], which is a real-time synchroniser that keeps a model and its generated code consistent all the time.

[1] See http://www.atego.com/products/acs/

The CML generator has been developed using ACS to implement the SysML to CML transformation rules defined in (MIYAZAWA et al., 2013). When the CML generator is added as a plug-in, the CML model generation algorithm is offered to users alongside those for other languages. ACS can also generate code on demand. ACS uses a selected ACS generation scheme to generate code. ACS generation schemes are implemented using the TDK (Transformation Development Kit). Artisan Studio provides the capability to change the code generation algorithms and add new ones using the TDK. The TDK, like ACS itself, acts on a model to produce a generator. The implementation consists of a model of the SysML to CML transformation rules which the TDK then translates into a CML generator. The CML generator can be installed subsequently to Artisan Studio's own installation, after which the CML code generation is offered to users alongside those for other languages.

The CML generator implements all rules for block, state machine, and activity diagrams. This has allowed the CML tools to be used to validate the generated CML models. In addition to syntactic analysis and type checking, some important modelling issues have been identified. For instance, the encoding of interfaces as subsets of operations and signals had to be modified to cope with finiteness constraints imposed on sets (as opposed to types) by CML.

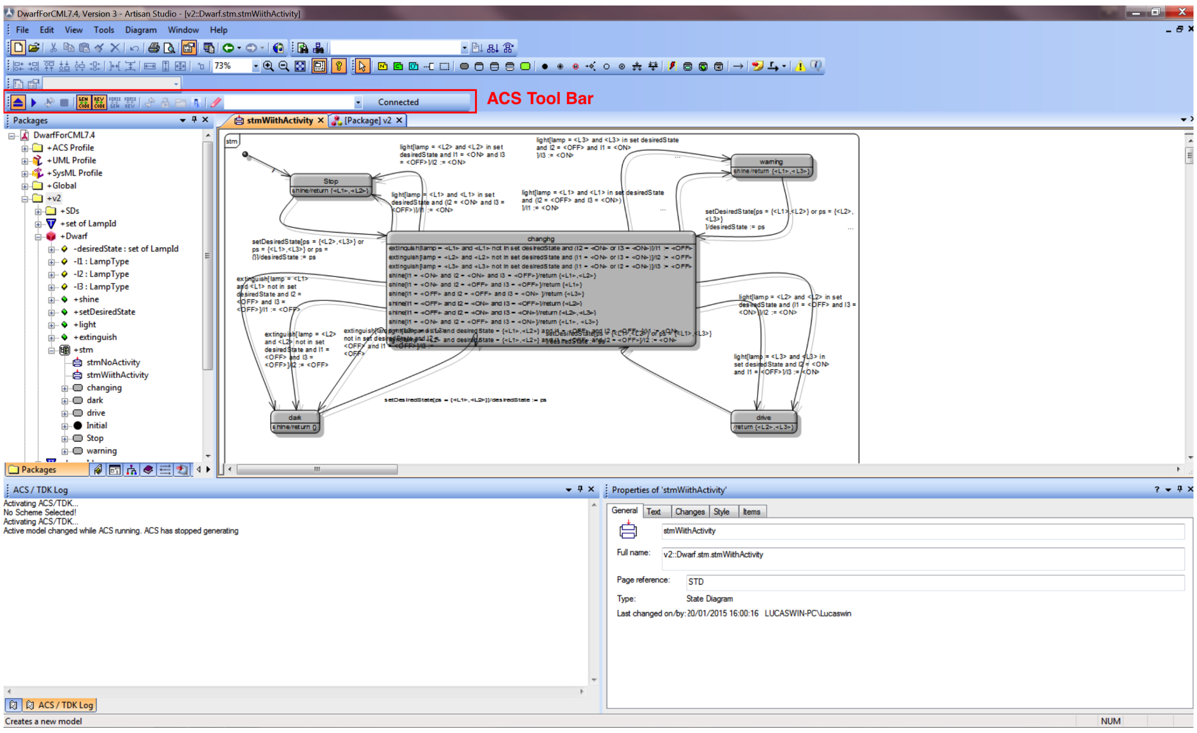
Other category of problems that has been addressed is related to gaps between the semantics of the different diagrams, which became apparent during the implementation of the rules and the validation of the models generated by them. For example, the event management processes (`controller_B` processes), which were part of the models of state machines, have become part of the models of blocks. The semantics of state machines was initially developed in isolation, and, whilst the integration of the models for state machines and blocks did not reveal any issues with the management of events, the incorporation of activities uncovered issues in the interaction of blocks, state machines and activities via events. It has been necessary for the `controller` process to become part of the parallelism that defines a block process to allow an activity process to search for particular events in a block's pool of events.

Figures 8.1, 8.2, 8.3, 8.4 show the steps to generate the corresponding CML specification of a SysML model. Figure 8.1 shows the ACS tool bar, which has the buttons related to the generation process. After selecting the package containing the SysML model chosen for generation, the user must click in the "play" button in the ACS tool bar highlighted in Figure 8.2. The tool opens a window, which is shown in Figure 8.3, where the user must inform the location of the DLL file corresponding to the CML generator, indicate the target folder where the CML specification must be generated and press the "Launch" button.

Afterwards, given that the SysML model respect our guidelines, the tool outputs in the chosen folder the CML files related to the specifications of the SysML model. The status of the generation process is detailed in the window on the left corner displayed in Figure 8.4.

Once we have the CML models, we need to adjust the specifications in order to perform animation and model checking. The steps for both are documented in procedures that are presented in Appendix B.

Figure 8.1: Artisan Studio: ACS tool bar.


DwarfForCML7.4, Version 3 - Artisan Studio - [v2::Dwarf.stm.stmWiithActivity]
File Edit View Tools Diagram Window Help
Connected
ACS Tool Bar
Packages
stmWiithActivity
[Package] v2
ACS / TDK Log
Properties of 'stmWiithActivity'
General Text Changes Style Items

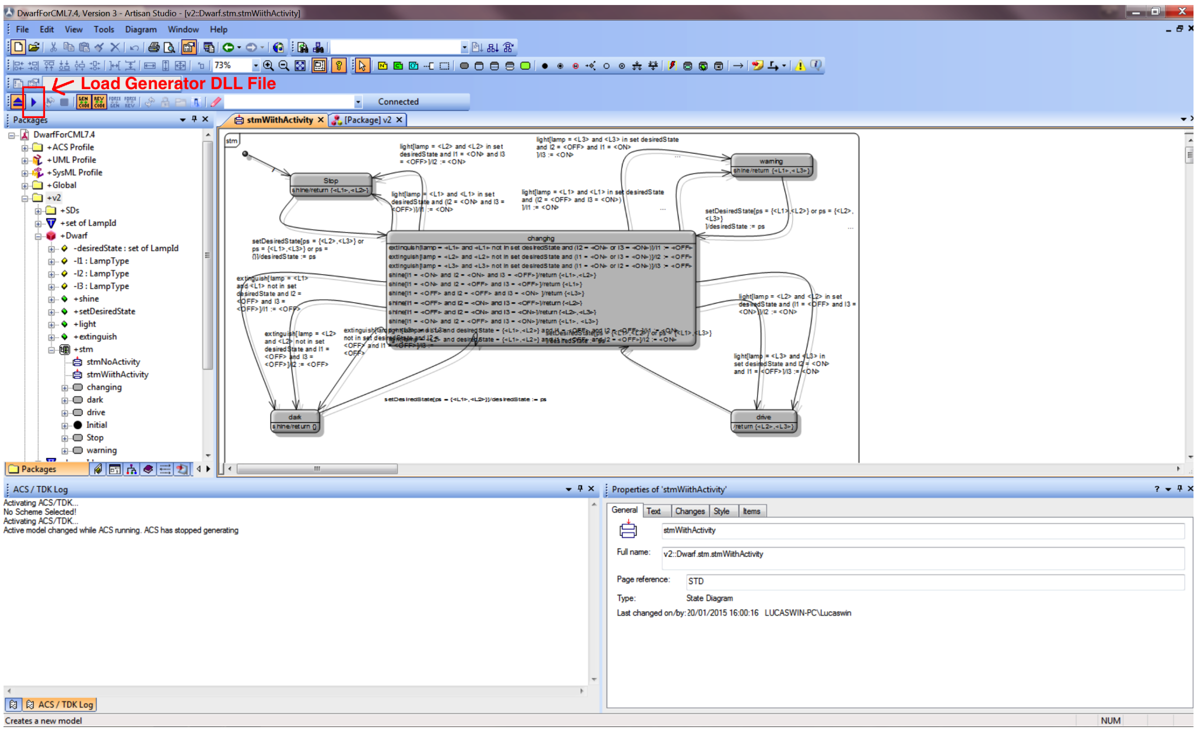
Activating ACS/TDK...
No Scheme Selected!
Activating ACS/TDK...
Active model changed while ACS running. ACS has stopped generating

stmWiithActivity
Full name: v2::Dwarf.stm.stmWiithActivity
Page reference: STD
Type: State Diagram
Last changed on/by: 20/01/2015 16:00:16 LUCASWIN-PC\Lucaswin

Creates a new model

Source: Author's ownership.

Figure 8.2: Artisan Studio: Button for loading the CML generator.


DwarfForCML7.4, Version 3 - Artisan Studio - [v2::Dwarf.stm.stmWiithActivity]
File Edit View Tools Diagram Window Help
Load Generator DLL File
Connected
Packages
stmWiithActivity
[Package] v2
ACS / TDK Log
Properties of 'stmWiithActivity'
General Text Changes Style Items

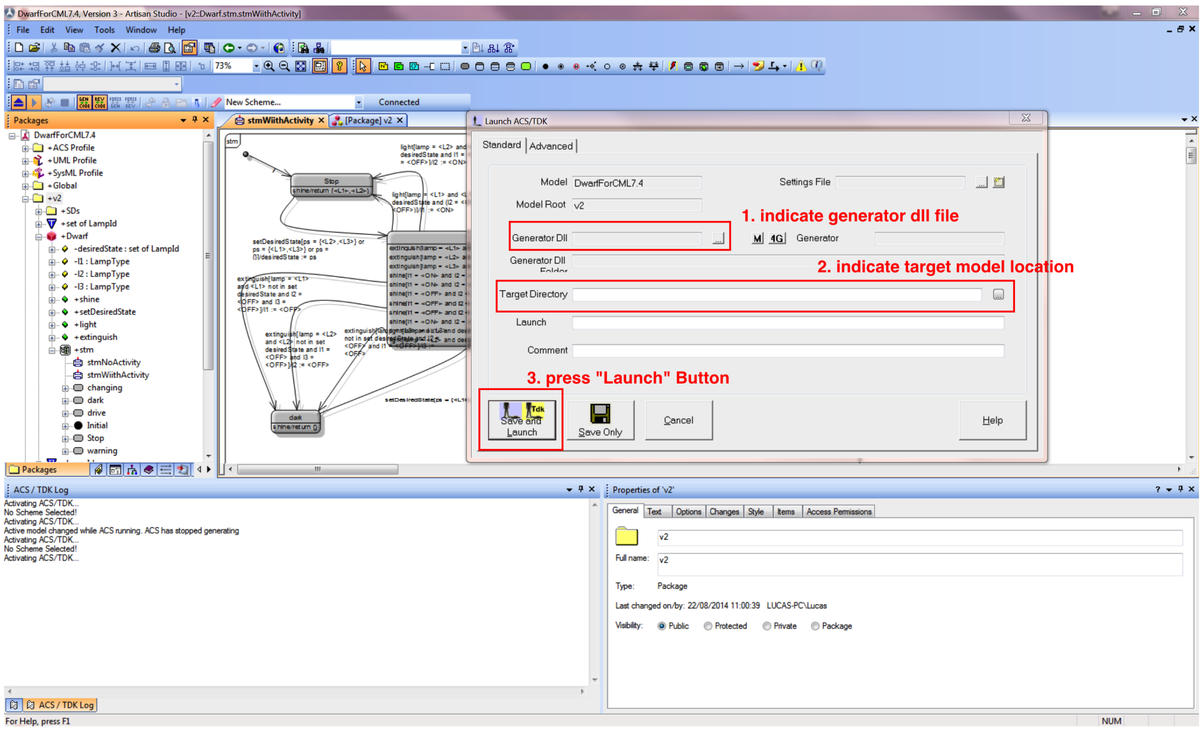
Activating ACS/TDK...
No Scheme Selected!
Activating ACS/TDK...
Active model changed while ACS running. ACS has stopped generating

stmWiithActivity
Full name: v2::Dwarf.stm.stmWiithActivity
Page reference: STD
Type: State Diagram
Last changed on/by: 20/01/2015 16:00:16 LUCASWIN-PC\Lucaswin

Creates a new model

Source: Author's ownership.

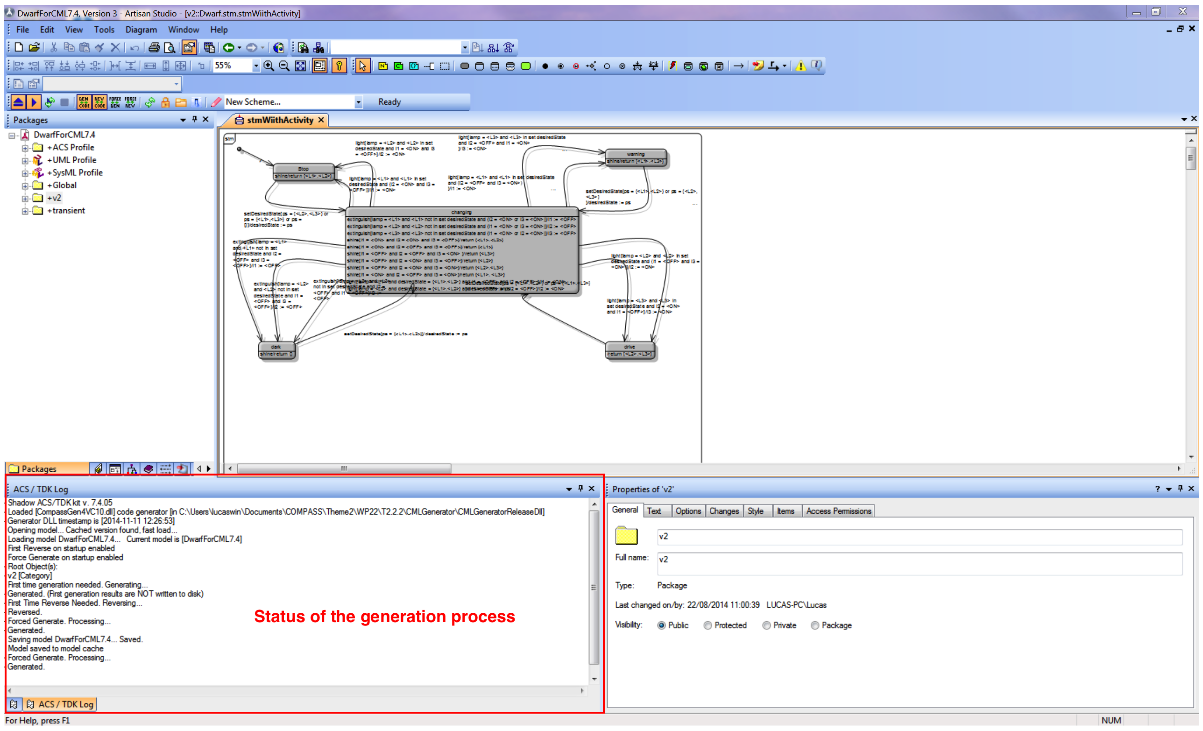
Figure 8.3: Artisan Studio: Configuring the generator DLL.



```
ACS / TDK Log
Activating ACS/TDK...
No Scheme Selected!
Activating ACS/TDK...
Active model changed while ACS running. ACS has stopped generating
Activating ACS/TDK...
No Scheme Selected!
Activating ACS/TDK...
```

Source: Author's ownership.

Figure 8.4: Artisan Studio: Status of the generation process.



```
ACS / TDK Log
Shadow ACS/TDK kit v. 7.4.05
Loaded [CompassGen4VC10.dll] code generator [in C:\Users\lucaswin\Documents\COMPASS\Theme2\WP22\T2.2.2\CMLGenerator\CMLGeneratorReleaseDll]
Generator DLL timestamp is [2014-11-11 12:26:53]
Opening model... Cached version found, fast load...
Loading model DwarfForCML7.4...   Current model is [DwarfForCML7.4]
First Reverse on startup enabled
Force Generate on startup enabled
Root Object(s):
v2 [Category]
First time generation needed. Generating...
Generated. (First generation results are NOT written to disk)
First Time Reverse Needed. Reversing...
Reversed.
Forced Generate. Processing...
Generated.
Saving model DwarfForCML7.4... Saved.
Model saved to model cache
Forced Generate. Processing...
Generated.
```

Source: Author's ownership.

8.2 Consistency Checking and Animation in Symphony

Here we describe how consistency checking and animation can be performed using the Symphony tool (COLEMAN et al., 2012). Symphony is an integrated development environment for CML. It has several tools for editing and executing CML specifications, including an editor and an animator.

One application of using Symphony is to check the consistency in the SysML model by checking errors on the corresponding CML model. Such a feature is considerably useful although most of the UML/SysML modelling tools do not provide such check. It comprises the checking of consistency between the different model elements (blocks, activities and state machines) used to describe the system. An example of lack of consistency is to use an operation in a transition of a state machine that is not defined in the corresponding block or using such an operation with a different protocol (e.g., different number of parameters). This is an easy mistake to make but a difficult problem to perceive when not supported by tools.

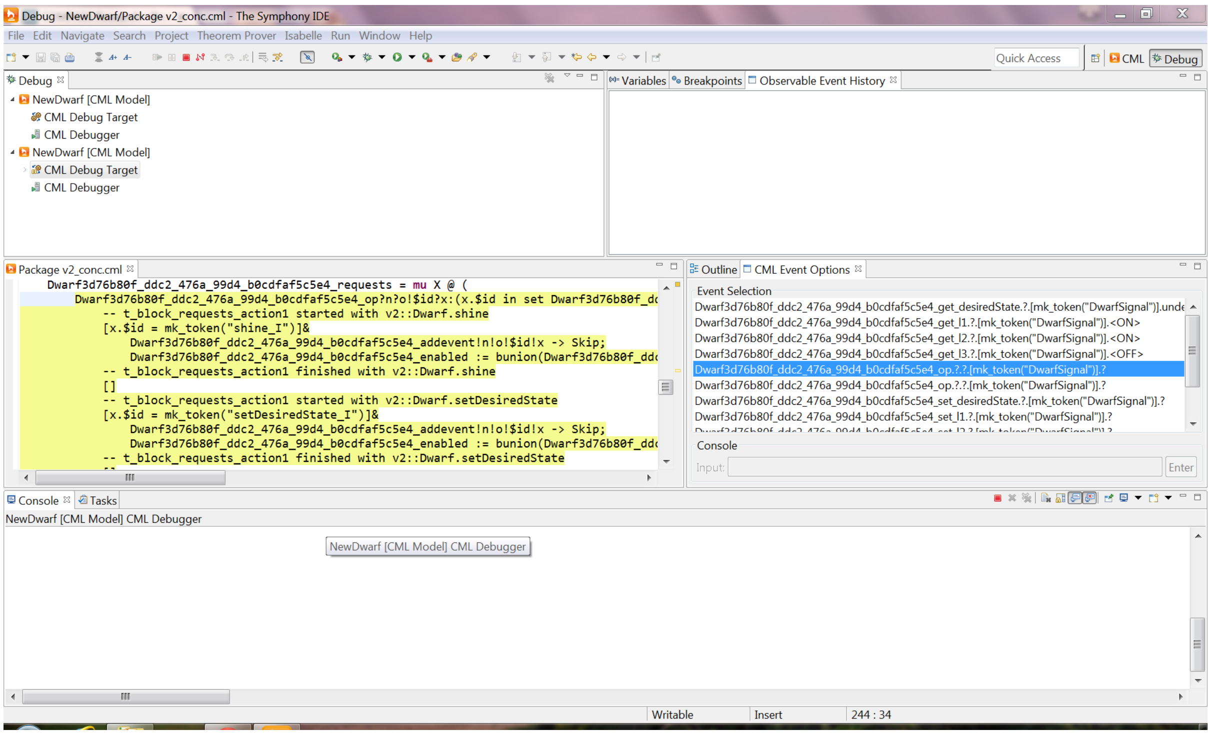
Some of these issues can be directly detected by syntactic errors that appears on the corresponding CML specification. For instance, in the case of using an operation with a different number of parameters, the Symphony's CML editor shows an error in the line where the event is communicated in the state machine because it does not conform with values of the corresponding channel declaration. Another example is invoking an activity that is not defined. Similarly, a syntactic error would be shown in Symphony. Therefore, problems related to undefinedness or incompatibility of messages between the different diagrams and model elements can be detected with the use of Symphony.

Animation can be used to exercise scenarios of the model. Although the user does not have to understand the details of the semantics, the user must comprehend the events syntax (operation calls and signals) of the CML model. As described in the previous chapter, the syntax is `[block name/connector]_op.[sourceID].[targetID].[operation protocol]` and `[block name/connector]_sig.[sourceID].[targetID].[signal protocol]` for operation calls and signals, respectively, where `[block name/connector]` is the name of the block or connector that links ports of blocks, `[sourceID]` is the identification of the source block, `[targetID]` is the identification of the target block, `[operation protocol]` is the signature of the call with the operation name and its parameters and `[signal protocol]` is the signature of the signal with its name and parameters.

By animating the model, a system engineer can perceive that certain scenarios are possible or do not depend on the availability of the events. Although it is a simple mechanism for assessing the model, most of the UML/SysML design tools do not support such a feature. Figure 8.5 shows the animation of a CML model of the dwarf signal example. A list with the current possible events is displayed on the CML Event Options window on the right-hand side. When the user selects (one click with the mouse left button) one of these events the CML action from where these event is made available is highlighted on the CML specification on the window shown on the left-hand side.

Figure 8.6 shows what happens when an event is selected for execution (double click with the mouse left button). The tool asks for values for each argument of the event on the console window on the bottom of the tool. The user must provide values in the declaration order

Figure 8.5: Animation in Symphony: selecting an event.

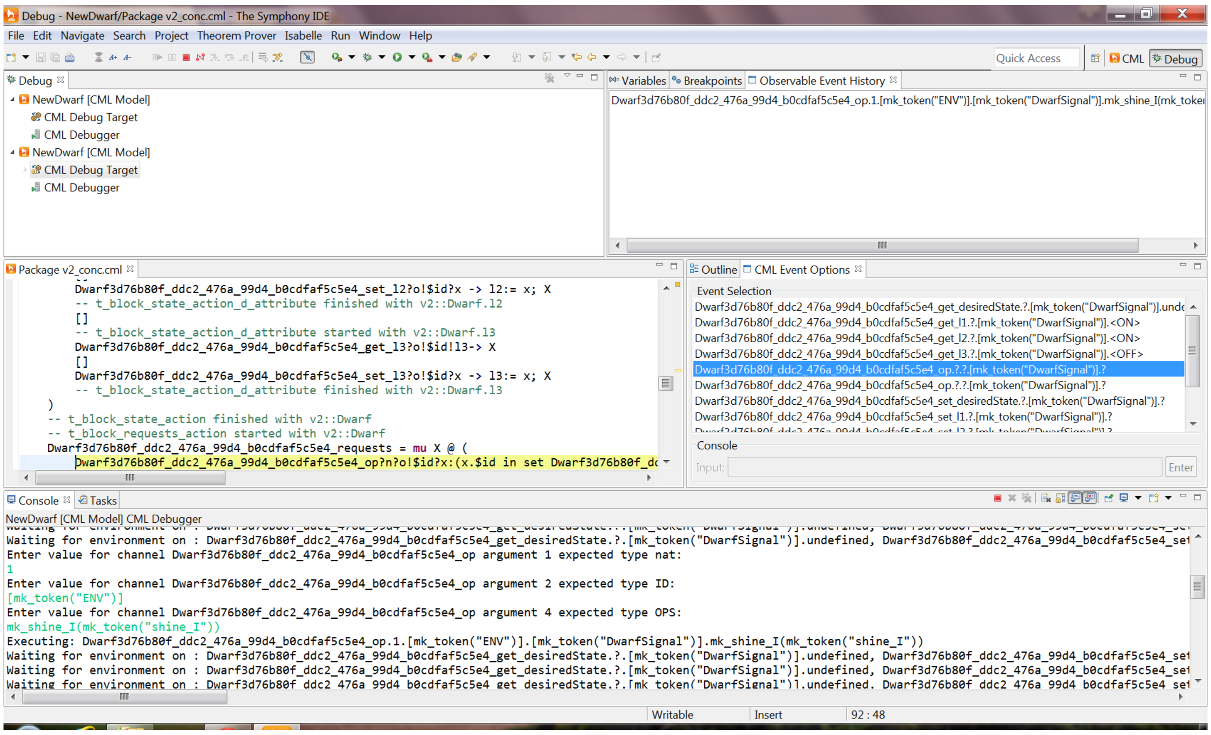


```
    Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_requests = mu X @ (
        Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op?n?o!$id?x:(x.$id in set Dwarf3d76b80f_dd
            -- t_block_requests_action1 started with v2::Dwarf.shine
            [x.$id = mk_token("shine_I")]&
                Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_addevent!n!o!$id!x -> Skip;
                Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_enabled := bunion(Dwarf3d76b80f_ddc
            -- t_block_requests_action1 finished with v2::Dwarf.shine
            []
            -- t_block_requests_action1 started with v2::Dwarf.setDesiredState
            [x.$id = mk_token("setDesiredState_I")]&
                Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_addevent!n!o!$id!x -> Skip;
                Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_enabled := bunion(Dwarf3d76b80f_ddc
            -- t_block_requests_action1 finished with v2::Dwarf.setDesiredState
```

```
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_desiredState.?.[mk_token("DwarfSignal")].unde
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_l1.?.[mk_token("DwarfSignal")].<ON>
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_l2.?.[mk_token("DwarfSignal")].<ON>
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_l3.?.[mk_token("DwarfSignal")].<OFF>
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op.?.?.[mk_token("DwarfSignal")].?
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op.?.?.[mk_token("DwarfSignal")].?
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_set_desiredState.?.[mk_token("DwarfSignal")].?
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_set_l1.?.[mk_token("DwarfSignal")].?
```

Source: Author's ownership.

Figure 8.6: Animation in Symphony: executing a valid event.



```
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op.1.[mk_token("ENV")].[mk_token("DwarfSignal")].mk_shine_I(mk_toke
```

```
        Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_set_l2?o!$id?x -> l2:= x; X
        -- t_block_state_action_d_attribute finished with v2::Dwarf.l2
        []
        -- t_block_state_action_d_attribute started with v2::Dwarf.l3
        Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_l3?o!$id!l3-> X
        []
        Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_set_l3?o!$id?x -> l3:= x; X
        -- t_block_state_action_d_attribute finished with v2::Dwarf.l3
    )
    -- t_block_state_action finished with v2::Dwarf
    -- t_block_requests_action started with v2::Dwarf
    Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_requests = mu X @ (
        Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op?n?o!$id?x:(x.$id in set Dwarf3d76b80f_dd
```

```
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_desiredState.?.[mk_token("DwarfSignal")].unde
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_l1.?.[mk_token("DwarfSignal")].<ON>
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_l2.?.[mk_token("DwarfSignal")].<ON>
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_l3.?.[mk_token("DwarfSignal")].<OFF>
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op.?.?.[mk_token("DwarfSignal")].?
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op.?.?.[mk_token("DwarfSignal")].?
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_set_desiredState.?.[mk_token("DwarfSignal")].?
Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_set_l1.?.[mk_token("DwarfSignal")].?
```

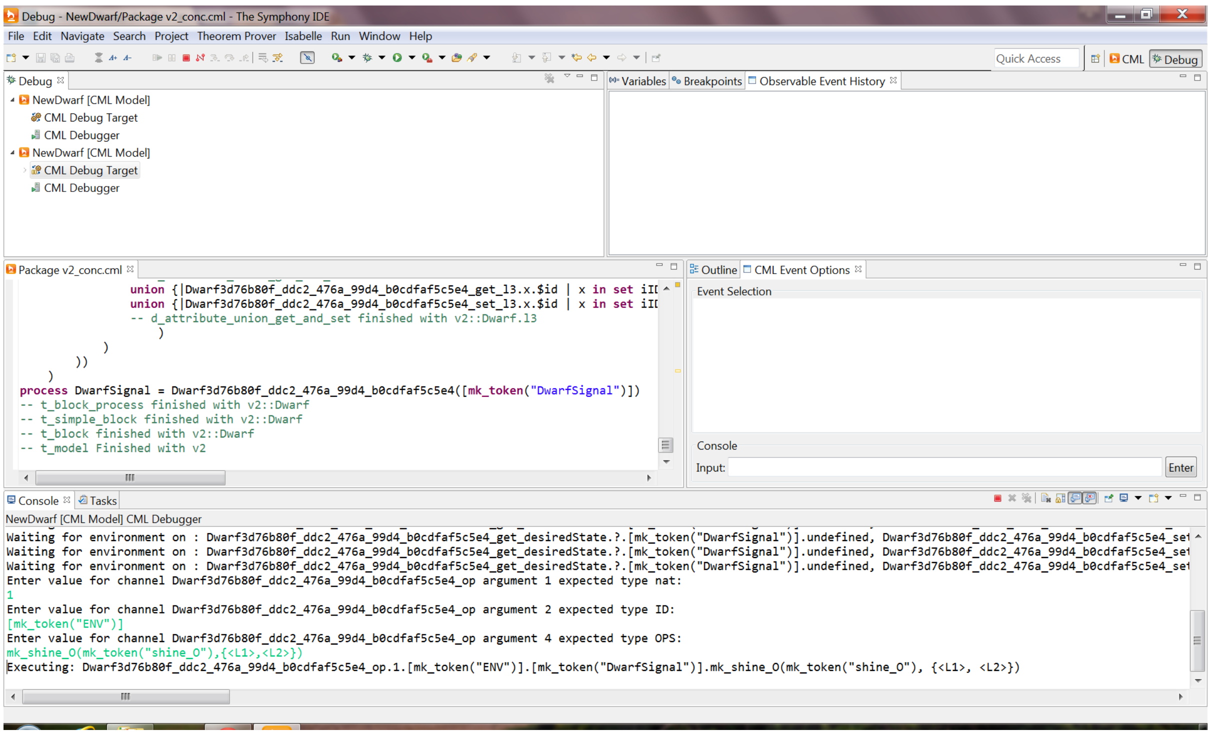
```
Waiting for environment on : Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_desiredState.?.[mk_token("DwarfSignal")].undefined, Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_set
Enter value for channel Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op argument 1 expected type nat:
1
Enter value for channel Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op argument 2 expected type ID:
[mk_token("ENV")]
Enter value for channel Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op argument 4 expected type OPS:
mk_shine_I(mk_token("shine_I"))
Executing: Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op.1.[mk_token("ENV")].[mk_token("DwarfSignal")].mk_shine_I(mk_token("shine_I"))
Waiting for environment on : Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_desiredState.?.[mk_token("DwarfSignal")].undefined, Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_set
Waiting for environment on : Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_desiredState.?.[mk_token("DwarfSignal")].undefined, Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_set
Waiting for environment on : Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_desiredState.?.[mk_token("DwarfSignal")].undefined, Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_set
```

Source: Author's ownership.

defined in the corresponding channel of the event. By the time of the writing, the Symphony animator does not provide support for the renaming operator, therefore, in this example we are animating the system model internally, that is, the same syntax for events described previously with the addition of a natural number before the source identification.

In this example the selected event is an operation call (channel `_op`), hence, the first argument according to the type of the channel is a natural number, where we provide the value `1`. The second argument is the identification of the source, `[mk_token("ENV")]`, which is a sequence with one token called `"ENV"` representing the environment. The third argument is the target identification. However, according to the definition of a block's request CML action in Rule 4.21 in Chapter 4, such an event is parametrised by the identification of the block (`$id`) that is receiving the call. Thus, we do not need to provide a value for this argument. The fourth argument is the operation call signature. Here, we provide the value (`mk_shine_I(mk_token("shine_I"))`) for a request of the shine operation call. After providing the last argument, the tool executes the event and process any internal event until reaching the next possible set of input events which are displayed on the CML Event Options window again.

Figure 8.7: Animation in Symphony: executing an invalid event.



```
                    union {|Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_l3.x.$id | x in set iII
                    union {|Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_set_l3.x.$id | x in set iII
                    -- d_attribute_union_get_and_set finished with v2::Dwarf.l3
                        )
                )
            ))
        )
process DwarfSignal = Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4([mk_token("DwarfSignal")])
-- t_block_process finished with v2::Dwarf
-- t_simple_block finished with v2::Dwarf
-- t_block finished with v2::Dwarf
-- t_model Finished with v2
```

```
Waiting for environment on : Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_desiredState.?.[mk_token("DwarfSignal")].undefined, Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_se
Waiting for environment on : Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_desiredState.?.[mk_token("DwarfSignal")].undefined, Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_se
Waiting for environment on : Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_get_desiredState.?.[mk_token("DwarfSignal")].undefined, Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_se
Enter value for channel Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op argument 1 expected type nat:
1
Enter value for channel Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op argument 2 expected type ID:
[mk_token("ENV")]
Enter value for channel Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op argument 4 expected type OPS:
mk_shine_O(mk_token("shine_O"),{<L1>,<L2>})
Executing: Dwarf3d76b80f_ddc2_476a_99d4_b0cdfaf5c5e4_op.1.[mk_token("ENV")].[mk_token("DwarfSignal")].mk_shine_O(mk_token("shine_O"), {<L1>, <L2>})
```

Source: Author's ownership.

Figure 8.7 illustrates the attempt to animate an event that leads to a deadlock. In this case, we are trying to animate a response (`mk_shine_O(mk_token("shine_O"),{<L1>,<L2>})`) to an operation call (messages of type `_O`) given that a request to that operation call has not been performed. After executing the event no other event is displayed on the CML Event Option window because the communication of such an event led to a deadlock.

This simple scenario shows how system engineers could use Symphony for exercising the flows of events of a SysML model according to the corresponding events in the CML specification. However, when there are several scenarios to be exercised, animating one by one is not a feasible task. Therefore, in the next section we provide a methodology for analysing scenarios and their properties using model checking.

8.3 Analysing SysML models

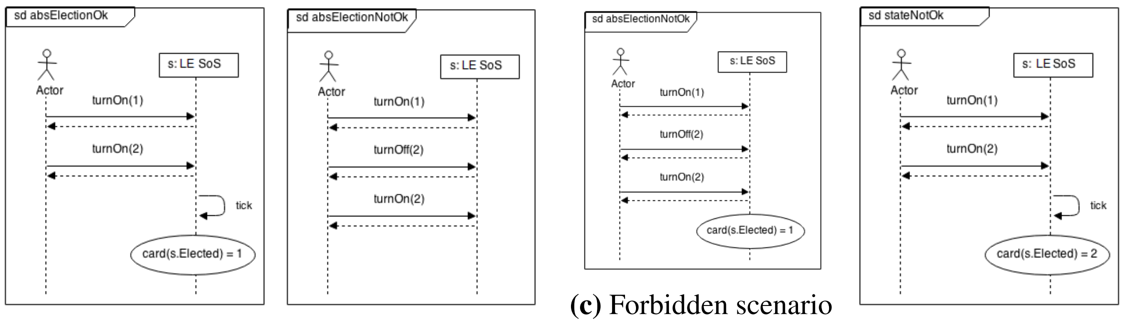
Here we use the refinement notions of CML to validate a SysML model. In particular, we focus on the use of model checking of the corresponding CML models. However, the models generated by our strategy can be considerably complex to analyse in reasonable time, and, by the time of writing, the CML model checker (MOTA et al., 2015) could not handle the models we were developing. Despite being extremely innovative due to the use of satisfiability modulo theories in order to support data with infinite domains, it is not mature enough to handle large models with a high degree of parallelism. Therefore, we decided to use the FDR3 model checker (GIBSON-ROBINSON et al., 2014), which is a model checker for CSP, one of the base languages of CML.

As described in Appendix B, we translate the CML specifications to CSP and perform several optimisations to allow the use of the FDR model checker, which is a considerably mature refinement checker for CSP specifications. This translation is not automated; it is described in terms of procedures that should be performed to the different elements of the CML specification generated from the SysML model in order to generate the corresponding CSP specification. Moreover, this translation is based on the mappings from Circus (WOODCOCK; CAVALCANTI, 2002) to CSP, which are duly proved (OLIVEIRA et al., 2014). Circus is another baseline language of CML and it has the process algebra concepts of CSP and the state features of the language Z (ISO, 2002). Due to the size of the models in terms of state, we needed a tool to verify properties in reasonable time. This factor, allied with the close relationship between CML and CSP, led us to choose the FDR model checker.

In this approach we use refinement checking to verify that the system can execute according to scenarios described by a sequence diagram. Using traces refinement, we can verify that the traces of a sequence diagram are in the set of traces of a SysML model defined by a collection of state machines, blocks and activities. So the scenarios defined by the sequence diagram are valid for the system.

More formally, if M is the CSP system model and SD is the CSP model of the sequence diagram, then $\mathtt{M} \sqsubseteq_T \mathtt{SD}$ asserts that the traces of SD are traces of M. This approach may seem strange for classical refinement checking practitioners because they usually have the process with the property at the left-hand side of the refinement. However, this approach fits our needs because we are checking traces containment between the two specifications, that is, the traces of M include

Figure 8.8: Sequence diagram examples.



(a) Valid scenario **(b)** Forbidden scenario **(c)** Forbidden scenario with state **(d)** Forbidden state

Source: Author's ownership.

the traces of SD. The CSP trace semantics is adequate for analysis based on sequence diagrams because, in this case, we are interested only on the services of the system. We recall that these correspond to operation calls, signals, and access to attributes of blocks. Divergences (livelocks) are not relevant for properties defined by sequence diagrams.

To illustrate the types of analysis we can do, we use the scenarios of our case studies, the leadership election and dwarf signal. Firstly, see the first two sequence diagrams in Figure 8.8 describing scenarios of the abstract leadership election model discussed in Section 3.2.2. Recall that, in this example, the system model is composed of a block (LE SoS) and a state machine that describes the behaviour of the block. Diagram 8.9a shows a valid scenario, where the devices 1 and 2 are turned on and the control event tick happens; we ignore the state invariant card(s.Elected = 1) for now. Diagram 8.9b describes a forbidden scenario: the second message is a turn off operation call for device 2, which has not been turned on. For Diagram 8.9a, the model checker does not return any counterexample, so the traces of the first sequence diagram model are valid. In the case of Diagram 8.9b, a counterexample shows a trace of the sequence diagram model that is not a trace of the system model, which is the trace `<[A].[B].turnOn.1, [A].[B].turnOff.2>`, where `[A]` is the identifier of the lifeline for Actor, and `[B]` of the lifeline for LE_SoS. This confirms that the scenario of Diagram 8.9b is not valid for the system.

Besides describing traces of communications, sequence diagrams are also capable of verifying properties of blocks by using the state invariant constructor, which is described in Section 2.1.5. With this constructor, a constraint can be assessed at a certain point of a block's lifeline. This constraint can be understood as a property that should be valid at a certain point of the scenario. Hence, besides checking the validity of traces, we can also verify properties described in state invariants. The corresponding semantics in CML of state invariants is briefly discussed in Section 6.1. We recall that if the state invariant yields false then an event `inv` is added to the trace that marks it as invalid. Nothing happens otherwise.

We note, however, that when we use state invariants like in Diagrams 8.9a, 8.9c, and 8.9d, the refinement $\mathtt{M} \sqsubseteq_T \mathtt{SD}$ may not hold even when the sequence diagram does specify a valid scenario of the system. This happens because some of the traces of SD may include a synchronisation on the extra channel inv that is not used in M. This is perhaps surprising, but as SD does not have a record of the state of M, and SD takes the value of the state (through _get_ channels) to evaluate the invariant, its traces cover all possible values that a state can have. On the other hand, the definition of M typically restricts the possible values of a state in a given scenario via the definition of the data operations and their behaviour.

For instance, in the Diagram 8.9a the state invariant defines a property of the attribute Elected. Hence, the CSP process SD for the sequence diagram takes an input on the channel LE_SOS_get_Elected to evaluate the state invariant. Since this input is not constrained, there are traces of SD for all values that Elected can take in this communication, including those for which the constraint does hold: the empty set and sets with more than one element. Such traces include a synchronisation on inv, which is never present in a trace of the CML process M for the leadership election protocol model. In addition, in M, the value that can be output on the channel LE_SOS_get_Elected after the operations turnOn in the sequence diagram, is restricted. So, traces for other communications on LE_SOS_get_Elected are not included in M. In summary, there are several traces of SD that are not traces of M, even though the sequence diagram presents a valid scenario of the model.

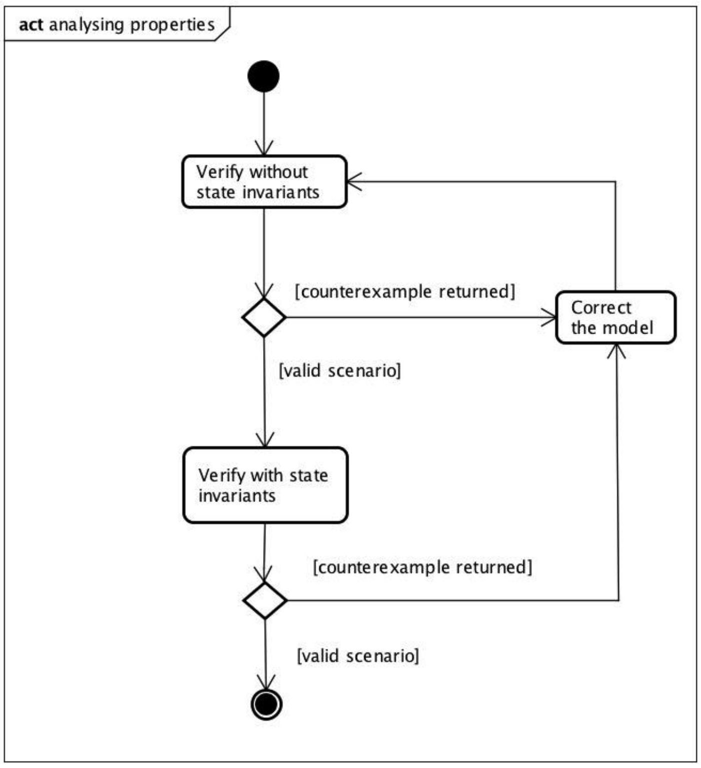
Consequently, to perform an accurate verification involving sequence diagrams with state invariants, we use another strategy performed in two steps. We note that the process $\mathtt{M} \underset{\Sigma \setminus \{\mathtt{inv}\}}{\|} \mathtt{SD}$, which composes M in parallel with SD, synchronising on all their common channels ($\Sigma \setminus \{\mathtt{inv}\}$ is the set-theoretic difference between the alphabet of the model and the inv event), captures the behaviour of the sequence diagram when it uses state information provided by the system model captured by M. The set Σ contains all the communications used in our CSP models. The traces of $\mathtt{M} \underset{\Sigma \setminus \{\mathtt{inv}\}}{\|} \mathtt{SD}$ are the traces of SD whose communications are also allowed by M. Consequently, the traces mentioned above, recording spurious values for the state components not allowed by M, are no longer included. On the other hand, traces may be eliminated due to deadlocks that arise when a trace of SD is not allowed by M. This may happen because the communications of the sequence diagram are not allowed by M. In this case, $\mathtt{M} \underset{\Sigma \setminus \{\mathtt{inv}\}}{\|} \mathtt{SD}$ is a traces refinement of M, even though the sequence diagram does not give a valid scenario of the system.

As an example, we consider Diagram 8.9c, which is a variation of Diagram 8.9b with an added state invariant. The sequence of messages defined by 8.9c is not valid for the reasons already discussed about Diagram 8.9b. In spite of that, the process $\mathtt{M} \underset{\Sigma \setminus \{\mathtt{inv}\}}{\|} \mathtt{SD}$ is a traces refinement of M. The parallelism $\mathtt{M} \underset{\Sigma \setminus \{\mathtt{inv}\}}{\|} \mathtt{SD}$ deadlocks after the communication corresponding to the first message turnOn(1). At this point, the sequence-diagram process SD is ready only for the second message turnOff(2), but such a communication is not available in the process M for the system model at this point. The traces of the parallelism are only the empty trace and the singleton trace with the communication for turnOn(1). Both of these are traces of M, and so

$\mathtt{M} \underset{\Sigma\setminus\{\mathtt{inv}\}}{\|} \mathtt{SD}$ is a trace refinement of M, although the scenario is not valid.

Therefore, our analysis strategy based on sequence diagrams with state invariants has two steps that are shown in the activity diagram of Figure 8.10. First, we verify the validity of the scenario in the sequence diagram, ignoring all state invariants. If a counterexample is returned, the practitioner must use this information to correct the model. Otherwise, we verify the properties defined in state invariants. Again, if a counterexample is returned, corrections should be made in the model and the process restarts. Otherwise, the scenario and the properties are valid in the system model.

Figure 8.10: Work flow of the analysis of properties.


act analysing properties
Verify without state invariants
[counterexample returned]
Correct the model
[valid scenario]
Verify with state invariants
[counterexample returned]
[valid scenario]

Source: Author's ownership.

Formally, in the first step, we check the refinement $\mathtt{M} \sqsubseteq_T \mathtt{SD} \setminus \{|\mathtt{inv}, *\_\mathtt{get}\_*|\}$, where only traces of SD without communications related to state invariants appear. We use $*\_\mathtt{get}\_*$ to refer to all $\_\mathtt{get}\_$ channels, which are used in SD only to access attributes for the evaluation of state invariants. If the refinement holds, then we check $\mathtt{M} \sqsubseteq_T (\mathtt{M} \underset{\Sigma\setminus\{\mathtt{inv}\}}{\|} \mathtt{SD})$ as suggested above.

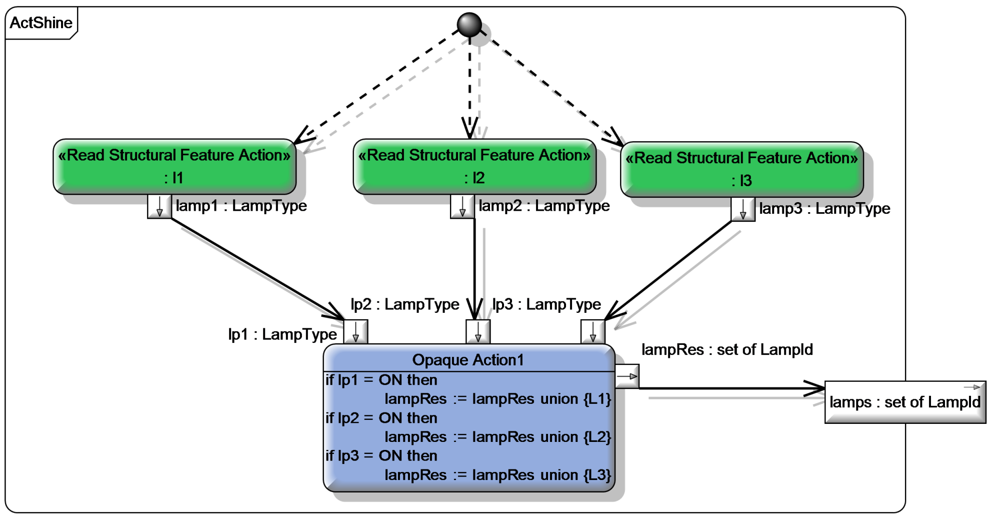
When using this strategy and executing the two steps, we find no counterexamples for Diagram 8.9a. As already explained, Diagram 8.9c describes a forbidden scenario with a state invariant. In this case, only the first step is necessary because, as the scenario is not valid, the refinement fails. The same counterexample presented above as part of the analysis based on Diagram 8.9b is relevant here. Diagram 8.9d, on the other hand, describes a forbidden scenario, where after two devices are turned on and the tick event takes place, the number of elected leaders is two: only one leader must exist when there are active devices after a tick event. According to our refinement checking strategy, the refinement in the first step holds, but a counterexample for the refinement in the second step is the trace <[A].[B].turnOn.1, [A].[B].turnOff.2,inv>, where inv indicates that card(s.Elected = 2) is violated. Of course this property is not desired for this system because after a tick event just one leader should be chosen among the active devices. We use this incorrect scenario to illustrate how the model checker proceeds when a property is not valid.

Regarding the leadership election case studies we discovered two flaws in the model using this verification approach. The first flaw corresponds to the absence of an event to represent the time passing. In the first version of the model, there was no tick event to represent the time. Therefore, the two transitions that are fired by tick on the state machine of the abstract leadership election (Figure 3.6 on page 50) did not have the tick trigger event; they only had guards. That leads to an ambiguous scenario where once the guards were true, the transitions could be performed or not. Hence, the model checker returned counterexamples where even after turning devices 1 and 2 on, the number of leaders was still zero. To fix that we had to explicitly add the tick event to represent the passage of enough time for the system to realise that it had a leader.

The second flaw is related to the visibility of modifiers of the block's attributes. In the first versions of the abstract leadership election, the visibility of the attributes were erroneously public. When we use public attributes on state invariants there is a possibility of other entities changing their values before the evaluation of the constraint. For instance, in Figure 8.9a, the Elected attribute of block LE SoS was set to {} before the evaluation of the state invariant because it had a public visibility. In order to correct this, the visibilities of the attributes are now private.

We have also applied this verification strategy to the dwarf signal example. In this example we exercised two versions of the SysML model, one composed of a block and a state machine, and another version composed of a block, a state machine and an activity. The former corresponds to the diagrams presented in Section 3.2.1. For the latter, the behaviour of the shine operation, which simply returns a set of lamps that are on, is being described in terms of an activity. The state machine shown in Figure 3.4 was modified as well to invoke the activity instead of deciding which set should be returned in the changing state. Therefore, instead of having six internal transitions in the changing state for the shine operation, in this new version only one transition is needed because it calls the corresponding activity. We recall that we have modelled the case studies and they were validated by the experts that have participated in the COMPASS project.

Figure 8.11: The ActShine activity.



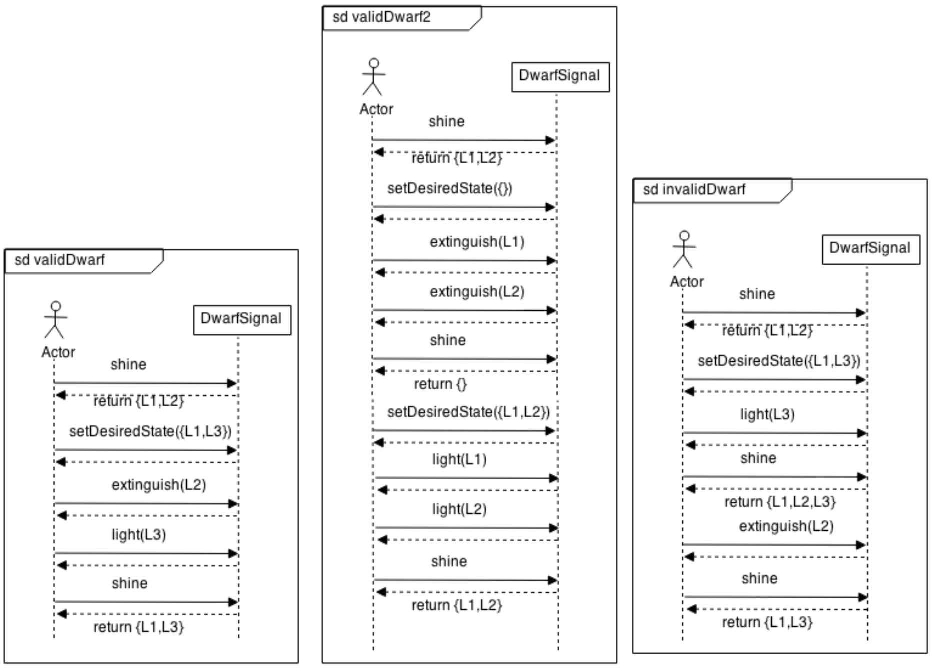
Source: Author's ownership.

Figure 8.11 shows the activity for the shine operation. It simply reads the values of the lamps through the Read Structural Feature actions and decides what the resultant set is according to these values in an opaque action. In order to validate these models we used the sequence diagrams that are displayed in Figure 8.12. The diagram displayed in Figure 8.13a describes a valid scenario where the lamps change from state stop ({L1,L2}) to warning ({L1,L3}). Figure 8.13b shows another valid scenario where the lamps change from stop ({L1,L2}) to dark ({}) and then returns from dark to stop. The last diagram shown in Figure 8.13c illustrates an invalid scenario because in order to go from stop to warning the signal turns the three lamps on, before extinguishing L2.

A flaw was detected in the first version of the dwarf signal state machine model when we tried to exercise the diagram shown in Figure 8.13b because the changing state did not have the last two internal transitions to treat light events (see Figure 3.4 on page 48). Thus, once the signal went dark it could not go back to the stop state because no light event could be treated. After inserting these two internal transitions the model checker did not provided any counterexample. The other sequence diagrams worked as expected. No counterexample is presented for the first sequence diagram and the counterexample `<[A].[B].shine,return {L1,L2}, [A].[B].setDesiredState.{L1,L3},[A].[B].light.L3>` is presented for the last sequence diagram because the three lamps cannot be concurrently on, where `[A]` is the identifier of the lifeline for Actor, and `[B]` of the lifeline for DwarfSignal. These results are valid for both versions of the model with the activity ActShine and the earlier version without it.

These examples illustrate how these verifications can be helpful in the development of safe and correct SysML models. When system engineers design complex systems with several possibilities of flows and concurrent events, the discovery of flaws are considerably difficult for manual verification. For example, in the dwarf signal case study we had lots of transitions to

Figure 8.12: Sequence diagram examples of the Dwarf Signal system.



(a) Valid scenario 1 **(b)** Valid scenario 2 **(c)** Forbidden scenario

Source: Author's ownership.

analyse in the state machine and some of them were missing. Therefore, we believe that such a strategy could help the development of complex systems in terms of detecting flaws during the design stage and to avoid their propagation in later stages of development or when systems are already deployed.

8.3.1 Analysis scalability

Although we have defined a analysis strategy for SysML models, an important concern still remains, which is how our analysis scale according to the size of the models. Early case studies have shown that the time and computational effort grow exponentially. Hence, we have searched for strategies that could improve the performance of the model checking.

We have studied how the FDR model checker works in order to discover ways of executing the refinement checking more efficiently. When a refinement assertion like $S \sqsubseteq I$ (the specification process S is refined by the implementation process I) is provided, the tool first tries to compile the specification and implementation, which means creating a labelled transition system (LTS) of the models and each of their sub-processes. Some optimisations are performed and then the tool tries to expand the LTS of the implementation process and verify if it has a sibling in the specification. Of course the level of compared information varies according to the semantic model used in the refinement, which can be traces, failures or failures-divergences. A refinement in the failures model checks if the traces of S contains the traces of I (traces refinement) and if the failures of S contains the failures of I, where a failure is a pair (s,X) such that s is a trace and X is a set of refused events after s. A refinement in the failures-divergences model extends the failures model in order to verify if the $divergences(I) \subseteq divergences(S)$, where $divergences(P)$ is the set of traces after which P can diverge. We say a process P diverges when it behaves like an infinite sequence of hidden events known as τ actions (ROSCOE, 2010).

The point here is that even if processes in the specification are not exercised by the implementation, the tool needs to compile them before the comparison of the LTSs. Therefore, the compilation time may require considerable effort of the analysis even if parts of this compilation are not necessarily relevant for the final result. Thus, when we have a large model in the specification and a small model in the implementation the refinement may take too long to conclude due to the size of the specification. In general, this is exactly what happens in our analyses given that the specification represents the whole model of the system and the implementation just exercises some scenarios of such a system.

Therefore, one possible way for improving performance of the refinement checking is transferring the complexity to the implementation (right-hand side of the refinement relation), rather than having a large specification. In addition, the failures-divergences model has a better performance than the others. Hence, we have defined a new refinement statement that captures these ideas, which is shown in Figure 8.14.

Figure 8.14: Refinement statement with optimisations for FDR.

$$\mathtt{end} \rightarrow \mathtt{DIV} \sqsubseteq_{FD} ((\mathtt{M} \underset{\Sigma \setminus \{\mathtt{inv},\mathtt{end}\}}{\parallel} \mathtt{SD};\mathtt{end} \rightarrow \mathtt{SKIP})\ \theta_{\{\mathtt{end}\}}\ \mathtt{DIV}) \setminus \{\Sigma \setminus \{\mathtt{inv},\mathtt{end}\}\}$$

Source: Author's ownership.

In this statement we have a failures-divergences refinement. On the left-hand side we have a process that communicates the event `end` and then behaves like `DIV`, which is the most simple divergence process. On the right-hand side we have the model of the system `M` in a synchronous parallelism with the model of the sequence diagram `SD` followed by the `end` event. This process can be interrupted once an `end` event occurs because we use the exception operator of CSP (θ_X) (ROSCOE, 2010). The semantics of $\mathtt{P}\ \theta_X\ \mathtt{Q}$ defines that when any event of X occurs in the process `P` such a process is halted and it starts to behave like the process `Q`.

In order to allow the refinement to happen we hide all events minus $\{\mathtt{inv},\mathtt{end}\}$. If an `inv` event occurs, then the state invariant has been falsified and the refinement is not valid because such an event is not on the alphabet of $\mathtt{end} \rightarrow \mathtt{DIV}$. If the traces of `SD` are not compatible with the traces of `M`, then a deadlock occurs and the event `end` and the divergence will never occur. In this case, a counterexample is returned showing that the event `end` is not present in the trace. Hence, this strategy only requires one refinement checking instead of two like the previous one. However, the drawback of this approach is that the events of the counterexample are not visible because we need to hide them all at the end of the process on the right-hand side, otherwise the refinement would not hold. Although it is possible to observe which are the hidden events, as several internal control events are also hidden, it may require considerable effort. We believe

Table 8.1: Specifications of the computer used in the evaluation.

Processor	Intel Xeon CPU E5-1650 3.50GHz (12 cores;15Mb Cache)
RAM	32 GB DDR4 2133MHz
Storage	512 GB SSD
Operation System	Windows 8.1 Pro

Source: Author's ownership.

that it is possible to have some tool support in order to make this treatment in the future.

We have exercised this strategy on all examples we had performed before with the previous approach, and it provided the same results as expected in considerably less time. Next we developed a scalability evaluation to compare the two approaches. We use the leadership election model varying the number of devices used in the model. The scenario exercised is the one illustrated in Figure 8.9a on page 157. Each refinement checking was executed in the same circumstances and in the same computer whose configuration is detailed in Table 8.1.

The evaluation consists in executing the scenario displayed in Figure 8.9a increasing the number of devices beginning with three devices. We take the average time of 5 executions for each number of devices with a given strategy. We defined a time limit of 24 hours for each analysis. When this limit is exceeded then we stop the analysis for that given strategy. Table 8.2 details the average time according to the number of devices for each strategy, where ST1 is our original strategy (performed in two steps), while ST2 is the approach detailed in this section. We can note that both strategies grow exponentially. The reason is that the model is a fully connected graph: for each inserted device, the possibilities of events grow exponentially according to the number of possible identifiers that may invoke operations and signals and according to the values of the channels of operations turnOn and turnOff. Moreover, the number of possibilities of leaders when a tick event happens also increases. Therefore, in general, the models will have an exponential feature. Nevertheless, models with a low number of identifiers and less possibilities of values in the channels may scale better. In any case, one can notice that ST2 provides considerably better results than ST1. For three devices the improvement is around 95%, for two 98% and for three 99%. While ST1 took more than twenty-four hours for six devices, ST2 finished in 89.96 seconds and it could also finish the analysis for up to eight devices in less than twenty-four hours. However, for nine devices our machine was not able to conclude the analysis in less than twenty-four hours. Considering the execution of ST2 for eight devices, FDR evaluated 293.275.785 processes.

Hence, we can conclude that ST2 reduces significantly the time of the analysis. However, the counterexample information is more difficult to read. Despite this improvement, the nature of SysML models allied to the several elements involved in our semantics can make the complexity of the corresponding formal models to grow exponentially. There are some strategies that can be applied in order to minimise the scalability issue:

- Data abstraction (LAZIC; NOWAK, 2003; FARIAS et al., 2004): these strategies

Table 8.2: Results of the scalability evaluation.

# of devices	ST1 average time (sec)	ST2 average time (sec)
3	12.44	0.66
4	115.25	1.92
5	1473.52	11.6
6	$>$ 24 hours	89.96
7	-	848.92
8	-	83430.93
9	-	$>$ 24 hours

Source: Author's ownership.

aim to compact the state space of the model by using techniques like local analysis, data independence, finding equivalence classes, partial order reduction among others. The overall idea is to reduce the possible values of data types using the only ones that are relevant for the analysis. Nevertheless, most promising approaches still need user expertise for a complete and adequate usage.

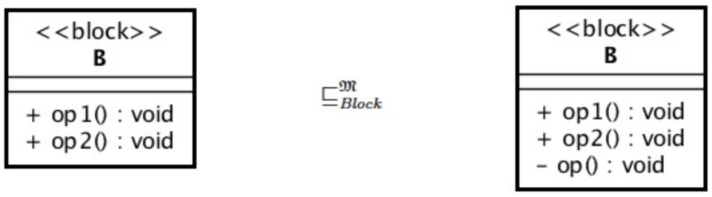
- Refinement calculus (MIYAZAWA; CAVALCANTI, 2014): Although our semantics cover several aspects of the SysML abstract syntax and be defined in a compositional manner, the corresponding formal models are too complex. This complexity can be minimised by the use of refinement rules that provides calculations to simplify the model. For instance, by eliminating flows that will never be executed or transforming parallelisms in sequence compositions when possible.
- Analysis based on communicating patterns (ROSCOE, 1998): these strategies aim to identify specific patterns in which the process interact because we can infer several simplifications to the analysis. Some of the verifications do not need to be performed because they are already proved given that the model respect the premise of the communicating pattern. Another approach is to lift the communicating pattern to the SysML level instead of verifying on the corresponding CSP specification avoiding the manipulation of the formal notation.

We believe that this strategies can play an important role for minimising the scalability problem and they are possible avenues of research for future work.

8.4 Validation of the Semantics

The translation from SysML to CML that we present in this work is itself a semantics we propose for SysML. A possible proof of correctness of this translation would require us to consider an independent semantics for SysML and CML, as well as a link between these semantics. The reasoning would entail to prove that, for an arbitrary SysML model, its translation

Figure 8.15: Example of refinement for inserting a private operation.



Source: Author's ownership.

generates a CML process whose semantics are linked to the source SysML model. As already mentioned, CML has a formal denotational semantics (WOODCOCK et al., 2013), and subsets of SysML also have formal semantics, including the foundational semantics for UML (fUML (OMG, 2013)). Nevertheless, a formal proof of our translation is out of the scope of the current work.

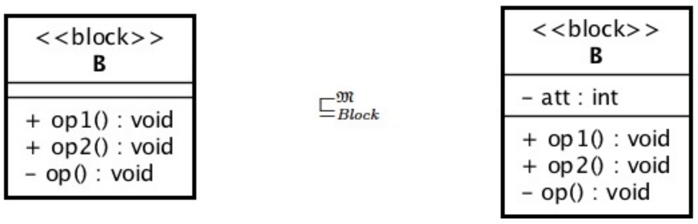
Nevertheless, in this section we present some initiatives performed to validate our semantics for SysML. These validations are presented in the form of refinement rules in the literature of UML/SysML that we verify using our SysML semantics. In this way, we could verify the consistency of some aspects of our semantics by performing semantic preserving transformations in SysML and verifying that their corresponding CML also preserve semantics. Basically, we take examples of SysML refinements available in the literature, then we build the two SysML models of the refinement, next we translate them to CML, translate them to CSP and finally we verify if the refinement is also valid using the FDR3 model checker. These validations are similar to sanity tests in order to provide evidences that our rules are sound. We recall that besides these validations, the rules were assessed by modelling experts that implemented them in the Artisan Studio tool. Several issues have already been identified and fixed along this implementation process; this also supports our confidence in the soundness of the rules.

Now we present some examples of refinement rules in the SysML level and then we describe how they are also valid according to our semantics. The refinements presented are the insertion of a private operation, the insertion of a private attribute and the decomposition of a block into a more structured one defined in terms of internal blocks.

8.4.1 Inserting a private operation

This refinement refers to the act of introducing local auxiliary behaviours in the model of a block as defined by Miyazawa and Cavalcanti (MIYAZAWA; CAVALCANTI, 2014). It takes a block and an operation, and introduces one operation in the block as a private operation. The resulting block has the same semantics as the original because the new operation is only available internally. Figure 8.15 illustrates an example of this refinement. Given a block B with two operations op1() and op2(), when we insert a new private operation op() the resulting block is a refinement of the original one. Whilst this refinement might seem useless, other operations can take advantage of the availability of the private operation to replace behaviours by calls to it.

Figure 8.16: Example of refinement for inserting a private attribute.



Source: Author's ownership.

This refinement holds according to our semantics. The newly introduced operation can only be used internally, that is, by the block internal elements like state machine and activities, and its use has to be justified by other rules, as previously discussed. Other blocks cannot invoke this operation because its operating events (`B_op`) are hidden by the block process according to Rule 4.16. Therefore the refinement is also valid in our semantics. We have translated this example to CML and CSP, and then we have verified the refinement using the FDR3 (GIBSON-ROBINSON et al., 2014) model checker. The corresponding CSP specification is available in Appendix C.

8.4.2 Inserting a private attribute

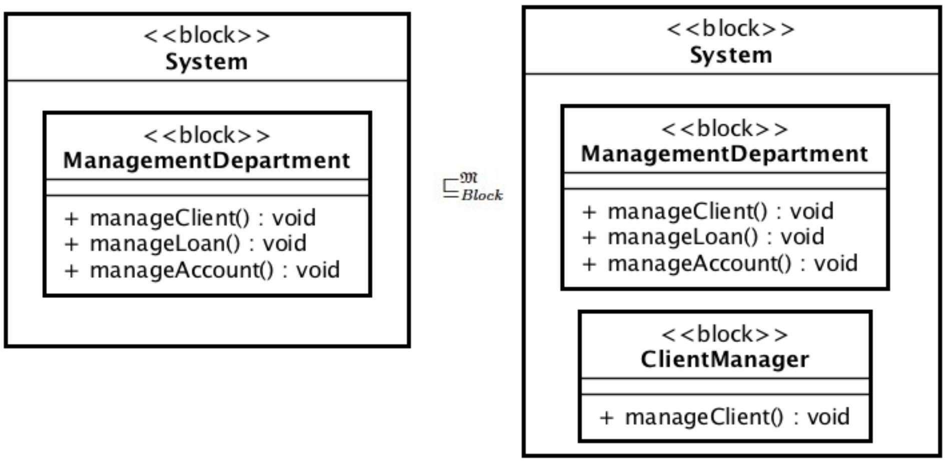
This second refinement is similar to the previous one because it deals with the introduction of another element with private visibility, however, this time it is an attribute of a block. To illustrate this scenario, in Figure 8.16 we take the resulting block of the previous refinement and add a private attribute att. The new block is a refinement of the previous one for the same reason the refinement of Section 8.4.1 works: the events that manipulate the attribute (`_get_` and `_set_`) are hidden from external elements as also detailed in Rule 4.16.

This type of refinement is considerably used because in general we need to encapsulate the internal structure of blocks to avoid unsafe accesses. We have also translated this example to CML and CSP in order to verify the refinement using the FDR3 model checker. The corresponding CSP specification is available in Appendix C.

8.4.3 Decomposition of a block

Another important task during system development is the decomposition of large units into smaller units. It involves breaking a block into other blocks in order to reduce the complexity of a system. This idea is similar to the extract class pattern (FOWLER, 1999) where classes are derived from another class that is too complex. It is a natural activity during the stepwise development of systems. We illustrate this decomposition performing a stepwise refinement of a ManagementDepartment block, which is a part of a System block. We assume that ManagementDepartment has three complex operations: manageClient(), manageLoan() and

Figure 8.17: Example of refinement for decomposing a block (part 1).



Source: Author's ownership.

manageAccount() and we want to restructure them in such a way that they are defined in terms of three other blocks, one for each operation. Firstly, we deal with the complexity related to clients by creating a block ClientManager that is a part of the System block as shown in Figure 8.17. The idea is that the ManagementDepartment block delegates any manageClient() calls to the ClientManager block, assuming that the latter has all the details to deal with clients.

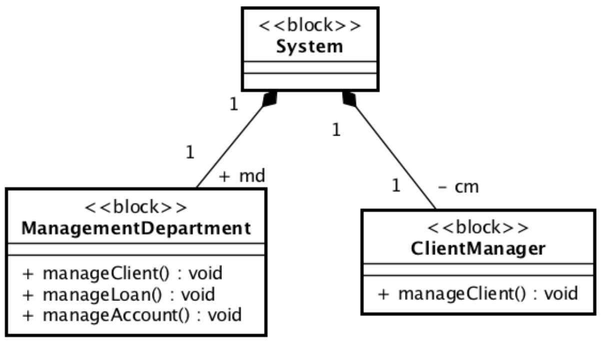
We could consider the ManagementDepartment block being restructured in terms of three internal parts instead of having a System block being the owner part of the composition. However, for simplicity, we present the model this way to cope with our guidelines. The ManagementDepartment block must have some behaviour to delegate calls and, according to our entity guideline number 4 in Section 3.1.1 on page 41, the owner block of a composition cannot have an associated behaviour.

Figure 8.18 shows the block diagrams of the right-hand side of the refinement. Figure 8.19a shows the block-definition diagram of the system where the blocks ManagementDepartment and ClientManager are parts of the System block. The internal-block diagram just connects the two parts of the composition as displayed in Figure 8.19b.

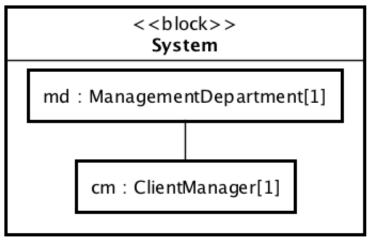
The state machine of the ManagementDepartment block is considerably simple. It has only one state with an implicit transition where when a manageClient() operation happens the action simply invokes the operation with the same name of the ClientManager block. Thus, the refinement shown in Figure 8.17 is also valid according to our semantics, because the System block accepts calls to manageClient() of the ManagementDepartment block due to its public visibility and it just passes the call to be treated by the ClientManager block, which has the operation call event hidden due to the private visibility of the part ClientManager.

The next step is to decompose the complexity of managing loans. This is performed similarly according to the refinement shown in Figure 8.20. We create a block LoanManager that is part of the System block to deal with the management of loans. Again, in order to cope with

Figure 8.18: Block diagrams of the ManagementDepartment system v1.



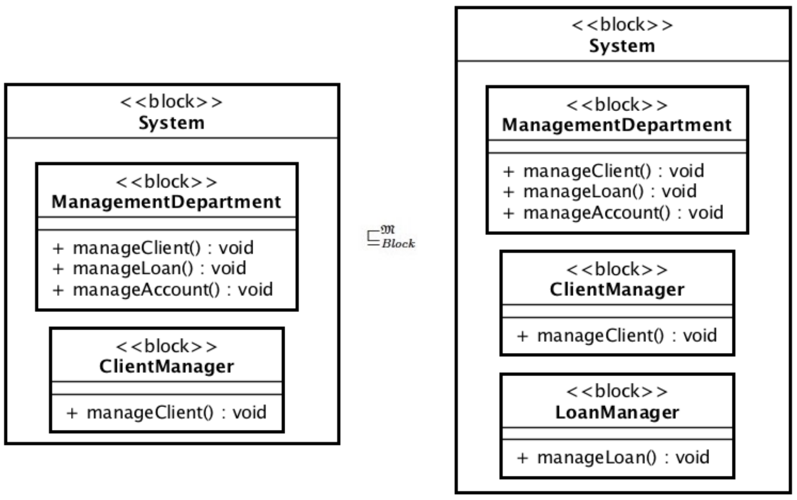
(a) BDD of the ManagementDepartment system.



(b) IBD of the ManagementDepartment system.

Source: Author's ownership.

Figure 8.20: Example of refinement for decomposing a block (part 2).



Source: Author's ownership.

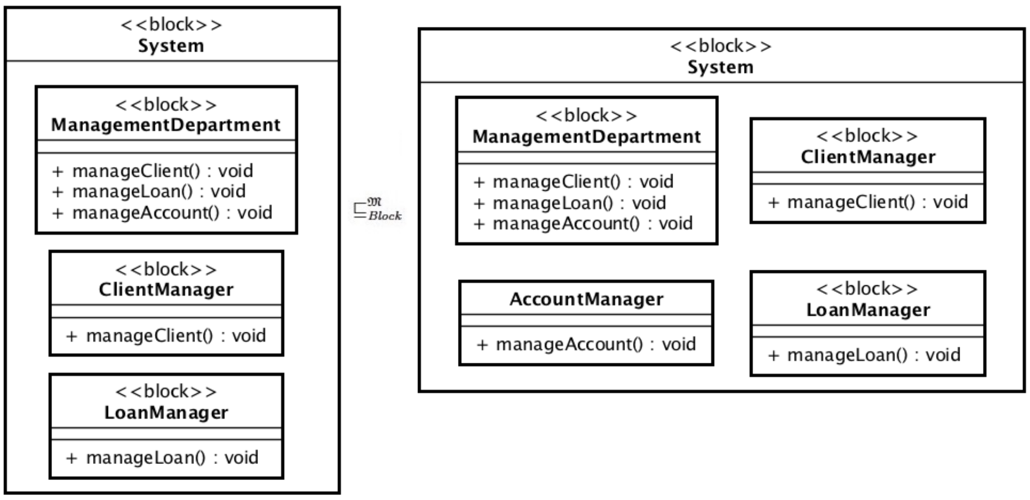
our guidelines, the new block is added as a part of the composition presented in Figure 8.19a and the ManagementDepartment part is connected to the LoanManager part in the internal-block diagram. Finally, the state machine also has to be updated in order to relay the calls.

The last step of the decomposition is to deal with the management of accounts. Similarly, we create a new block AccountManager to deal with this activity and the refinement is shown in Figure 8.21. The model has to be updated as already done for ClientManager and LoanManager. The final result is displayed in figures 8.22 and 8.23.

This final refinement also holds in our semantics for the same reason previously explained; the calls to manageClient(), manageLoan() and manageAccount of the block Management-Department are public, hence, visible in the traces. However, they are internally passed to the blocks ClientManager, LoanManager and AccountManager, respectively.

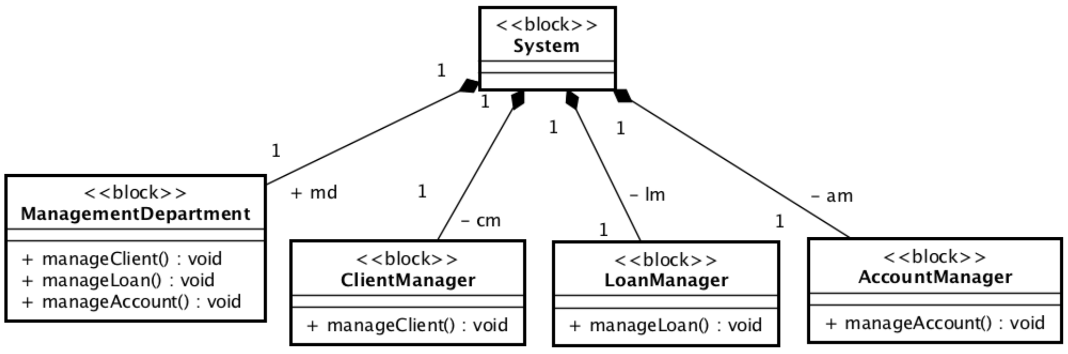
Again, we have translated the SysML models to CML and CSP, and performed the

Figure 8.21: Example of refinement for decomposing a block (part 3).



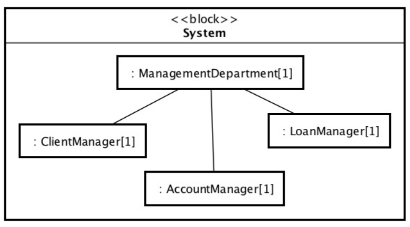
Source: Author's ownership.

Figure 8.22: Final BDD of the ManagementDepartment system.



Source: Author's ownership.

Figure 8.23: Final IBD of the ManagementDepartment system.



Source: Author's ownership.

refinement analysis using the FDR3 (GIBSON-ROBINSON et al., 2014) model checker in order to validate our semantics. The resulting CSP specifications from the translations are available in Appendix C.

8.5 Related Work

Lano and Evans have proposed a systematic development process using UML and a mix of syntactic checks and formal verification for consistency, enhancements and refinements among class diagrams, state machine, sequence, and collaboration diagrams (LANO; EVANS, 1999). Modelling and verification are carried out by hand, using first-order set theory. No general translation strategy like the one presented here has been developed.

Davies and Crichton have also proposed a formal semantics in CSP for UML class, object, state-chart, sequence and collaboration diagrams (DAVIES; CRICHTON, 2003). The semantics is used to verify both refinements and the consistency between sequence and the remaining diagrams. The translation from UML to CSP has been illustrated via examples. No general transformation rules are introduced.

Baresi et al. proposes a formal semantics based on a metric temporal logic for UML class and object diagrams, state machine diagram, sequence diagrams and interaction overview diagrams (BARESI et al., 2012). The formal models are used to analyse satisfiability of the model and to check properties of the system. Sequence diagrams are not used to describe valid our invalid scenarios and the focus of this work is in defining a framework for the cited subset of UML diagrams.

Makartetskiy and Sisto show efforts towards integrating embedded systems modelling with verification measures (MAKARTETSKIY; SISTO, 2011). They have proposed an approach to analyse the refinement of SysML requirements defined in terms of state machines. The underlying formalism is CSP and the tool used was the PAT model checker (SUN et al., 2008). However, the transformations are not completely automated and they work in higher level of abstraction than we do. Our strategy work at the design phase of the system development, while this work only tackles requirements.

Chouali and Hammad have proposed an approach that combines component SysML models and interface automata in order to verify formally their interoperability (CHOUALI; HAMMAD, 2011). They use block definition and internal block to define the components of the system, and sequence diagram to specify protocols between the components. They translate the SysML model to interface automata in order to verify interoperability in component-based systems.

Knorreck et al. present TEPE, a graphical TEmporal Property Expression language based on SysML parametric diagrams (KNORRECK et al., 2011). Properties are built upon logical and temporal relations between block attributes and signals. The approach translates the SysML model to automata to serve as input for UPPAAL (BENGTSSON et al., 1995) model checker for formal verification.

Abdelhalim et al. provide a strategy for a systems modelling approach based on UML and fUML (OMG, 2013) together (ABDELHALIM et al., 2011, 2013). It uses UML state diagrams for modelling system object behaviour abstractly, then refining each state diagram by adding the implementation decisions in a form of a fUML activity diagram. They have introduced a framework based on CSP that uses FDR (FDR: USER MANUAL AND TUTORIAL, VERSION 2.28, 1999) for checking behavioural consistency between each UML state diagram and its corresponding fUML activity diagram. The structural part of the system is not covered by this work.

8.6 Final Remarks

We have presented the validation and applications of our semantics to analyse SysML models. The validations were performed according to model refinements available in the literature and the experience of the implementers of the CML generation plugin in Artisan Studio. Also, the analyses of SysML models come in hand when system engineers have to verify if their models are consistent. Most of our semantics is implemented in the Artisan Studio tool, so that practitioners can create their models and generate the corresponding CML automatically. Once the CML specifications are available, the users can animate the model or manually translate the CML to CSP to use refinement checking in order to validate its traces.

Due to the compositional and general way our semantics is defined, we need to make the models more concrete to allow animation in the Symphony tool. As previously mentioned, the Appendix B describes how to derive a CSP model from a CML specification in order to apply the refinement checking approach we propose in Section 8.3. Some of the procedures require restricting the sizes of sets and sequences of the model. They are defined to reduce the state space, and thus, make the analysis feasible. However, models with considerable large state spaces may impact the feasibility of the approach. In situations like this, the model checker eventually aborts due to the lack of memory resources. To assess this concern, we have performed a scalability analysis, as described in Section 8.3.1, to provide an estimate about the size of the models we can handle. Another aspect regards restricting the size of one of these sets or sequences in a wrong way. For instance, procedure 5e on Appendix B defines that we should restrict the size of the parameter `enabled` according to the maximum size of concurrent request a block may receive. If this limit is not set correctly, then the result of the refinement analysis is not guaranteed to be right. For example, some scenarios can be returned as counterexample by the fact that this element could not store any more requests because it reached the wrong limit that was set.

The result of our analysis comes in terms of counterexamples in the format of the events defined in CSP. Therefore, the user must at least know the meaning of such events in order to identify in the SysML model what should be corrected. However, we plan to improve this outcome in the future by transforming the trace counterexample in a sequence diagram. This would help practitioners in understanding the erroneous trace by visualising a graphical language they are used to instead of reading a CSP trace.

Nevertheless, we have shown that our approaches to formal verification may help practitioners in finding errors in their design models. We have found problems in the two SysML models from our industry partners, the leadership election problem and the dwarf signal. Even simple SysML models may have flaws that are difficult to detect by reading each one of the diagrams. Therefore, we see this line of work as a promising technique to be used for the verification of system design models described in SysML.

9 Conclusion

We have presented a formal semantics for a comprehensive subset of SysML through a mapping into CML. Starting from the CML semantics for individual elements, we have adapted and evolved the semantics for an integrated view of a SysML model provided by the relationships between elements in a typical SysML design. Our work is extensive with respect to the coverage of both elements and constructions of diagrams, even when the diagrams are considered in isolation.

To allow a coherent formal interpretation of a SysML model, we have proposed guidelines that assign design roles to be played by each of the elements in an integrated model. The structural model and the behaviour of its internal components are captured by blocks, state machines and activities. Desired interactions that the model must (or must not) allow are specified by sequence diagrams. Although the semantics of state machines is not a contribution of this thesis, its integration with other model elements and the guidelines of usage that allow the integration are.

In addition to generality and integration, the following concerns have guided the design of our models.

- Abstraction. The models are given at a level of abstraction that makes them suitable for a variety of analysis techniques, including model checking and theorem proving. Not all generated models are executable, but we have shown how the limit of the number of blocks and their operations can produce an executable version of the model.
- Compositionality. Parts of the CML models can be analysed independently, and problems found can be traced back to elements of the SysML model. Besides being used in an integrated context, each model has its own characteristics, and they can be analysed independently. For example, a designer can use the FDR model checker to analyse the model of an activity, checking for deadlock freedom or if it is non-deterministic.
- Independence from particular tools. We provide a denotational semantics for the SysML models in terms of translation functions that receive as input elements of the

abstract syntax of SysML and generate the corresponding CML. The models define a theory of refinement and programming for SysML. We have explored this feature by considering refinement notions and laws for SysML models. One implementation of our semantics has been performed by Atego in its Artisan Studio modelling tool; however, other tools can also implement the same semantics in order to generate the corresponding CML models.

The main purpose of our formal semantics is to serve as a sound basis for a comprehensive reasoning strategy for SysML models based on refinement.

We take advantage of the wide range of CML constructions and operators, concerning both state and control behaviour, as well as its refinement theory (WOODCOCK et al., 2013). The SysML semantics is presented as a set of compositional translation rules whose automation contributes both for validation and applicability of our work.

With the mapping of SysML into CML, it is possible to check whether traces defined by sequence diagrams are valid or not in the obtained model. This can be automatically achieved, via refinement checking, using the CML model checker being developed, or by translating the CML model into an input notation for other tools. So far, we have translated CML into CSP and used the refinement checker FDR. Another form of validation that we have discussed is animation using the CML simulator in the Symphony tool, which exercises the SysML model by executing the communications of the CML model. The CML tool environment is considerably fresh but promising. With the development of the Symphony tool we expect that the model checking analysis and other kinds of analyses will be done directly in Symphony with no need to provide intermediary translations, like the one to CSP.

The entire approach has been illustrated using the industrial case studies leadership-election and dwarf signal. The semantic mapping has been exemplified for some constructions of the model, and both animation and scenario validation have been carried out.

Some limitations of our work are discussed as follows:

- **Loss of traceability between SysML and CML elements**: One of the challenges of our approach is to achieve complete traceability between CML elements, at the semantic level, and SysML constructions of the source diagrammatic model. This has been addressed by defining the translation rules in a compositional manner and documenting the generated CML to facilitate traceability. However, this limitation persists when we optimise the generated CML in order to perform model checking because the traceability information is lost.

- **Maintanability of our translation rules**: Again, compositionality is helpful. Our rules are organised into groups according to the SysML elements they address, so changes are contained to the groups concerned with the elements affected. A more sophisticated mechanism of maintenance is best considered in the context of Artisan

Studio because the dependencies between the rules can be explicitly viewed by the implementers. For instance, if some rule is removed or has its signature modified, the other rules that depend on the updated rule are marked with possible syntax errors.

- **Validation of our semantics**: The validation of our semantics was performed by the refinement scenarios we have exercised as described in Section 8.4 and by the Atego's modelling specialists who have implemented the translation rules in the Artisan Studio tool. We have used a version control system and a bugtracking tool to support the implementation of the rules. When these specialists found inconsistencies or errors in the rules, they would open an issue for us to resolve. This process has helped us to improve the quality of the translation rules. However, it does not guarantee the correctness of our semantics. Addressing semantic correctness is rather difficult because the official semantics of SysML is descriptive. Nevertheless, the OMG has spent some effort in defining a foundational semantics for UML (OMG, 2013), which is formally defined using first-order mathematical logic language called PSL (GRÜNINGER; MENZEL, 2003). Thus, we could relate our semantics in CML to PSL in order to prove the soundness of our semantics according to the OMG standards. However, by the time of the writing, OMG had only provided the semantics for classes, activities and composite structures. The semantics of state machines and interactions could still not be proved. Moreover, we perform transformations in the model to animate it (define finite sets and sequences) and to perform model checking (CML to CSP) and we do not prove that they preserve the properties of the previous model. Regarding the former, the intuition is that we just gather all possible values of the types internally defined for our semantic in finite sets and sequences by traversing the model. Hence, it would respect the previous model, however, we cannot guarantee any preservation for the user-defined types because their values are manually provided by the user. Regarding the CML to CSP, we have reused several results from the mappings provided by Oliveira et al. (OLIVEIRA et al., 2014) from Circus (WOODCOCK; CAVALCANTI, 2002) to CSP.

- **Expressiveness of our guidelines of usage**: Although one of our contributions are the guidelines of usage, which provide a manner for creating models, this may also limit the expressiveness of SysML. System engineers may use other modelling styles rather than the one we are proposing. However, we believe that some of these models can be transformed in other versions that comply with our guidelines. One kind of practitioners who have the greatest benefit of adhering to our guidelines are the novice ones because they usually need to learn or define a modelling style to start creating their models. Another advantage of adhering to our guidelines is to keep uniformity and standardisation of the different models that are constructed. For instance, our guidelines can be used for standardising the models of a company.

- **Analysis of complex models**: As we use model checking analysis we inherit the same limitation of this strategy regarding the state-space explosion problem. However, we note that models with more entities and relationships do not necessarily mean a larger model to be analysed. In fact, in some cases, having more models and relationships could restrict the state space even further, which reduces the computational effort in the analysis. For example, a block with several operations but with no relationships that invoke them makes the model checker to expand all possible invocations of these operations. When we define blocks that access these operations, we restrict the set of invocations according to those defined by the relationship. Indeed, our problem is with models that do not impose limits to its state space. Therefore, in order to deal with this drawback we plan to assess the use of data abstraction techniques (LAZIC; NOWAK, 2003; FARIAS et al., 2004). Although we already propose some data abstraction mechanisms in Chapter 8, we plan to investigate other approaches that could improve the performance of our analysis.

9.1 Future Research

There are some interesting opportunities for further research that will contribute to the proposed approach.

Regarding our guidelines of usage of SysML, to facilitate their communication to practitioners and their implementation in other tools, it can be beneficial to formalise them using OCL (WARMER; KLEPPE, 2003), for example. In addition, practitioners can also benefit from a method to construct SysML models that satisfy our guidelines; this can complement an automated support to check that an existing model follows the guidelines.

A more theoretical line for future work is the use of the precise semantics defined by fUML to establish the consistency between fUML and our CML semantics, for the constructs covered by fUML. Besides its denotational semantics (WOODCOCK et al., 2012b), CML has an operational semantics (BRYANS et al., 2014), used in its animator and model checker, and an algebraic semantics, used in its theorem prover and refinement editor. Exploring the relationship between PSL and CML, in the context of the Unifying Theories of Programming (HOARE; HE, 1998), which are used to give a denotational semantics for CML, is a promising way forward.

The generated CML models are not suitable for human readers. Whereas this is not relevant for our goal of reasoning purely at the SysML level, readability can be useful in other contexts. Further modularisation of the CML model to separate the semantics of the SysML protocols from the CML models of features of particular elements can improve readability. This, most likely, however, requires the use of higher-order actions, a feature not currently available in CML.

One of our major objectives is to develop a comprehensive framework to allow reasoning purely at the SysML diagrammatic level, with the CML models and analyses totally hidden from the developer. We plan to develop other case studies to explore the semantic mapping and the reasoning strategy described here. We also aim to investigate further data abstraction techniques that could be applied in our approach in order to allow the analysis of even more complex systems in terms of the number of states.

As discussed, we have implemented the transformations required for simulation using the CML animator and we have supported the implementation of our translation rules in the Artisan Studio tool. Other opportunities for automation include the support for refinement in general, especially at the SysML diagrammatic level, and for optimising model checking in FDR and other tools.

Moreover, we aim to expand the coverage of our approach. We plan to provide semantics for other SysML features, including new concepts, like the parametric diagram, and other exclusive characteristics of SysML. In addition, we plan to implement our translation functions in other modelling tools considering new model elements that we may provide semantics for (like the parametric diagram) and the sequence diagram semantics, which was not implemented in Artisan Studio.

Finally, extension of SysML to include CML concepts for refinement, and of CML to include SysML concepts, like asynchronous communication and shared variables, are interesting avenues for further work. What we have now, however, is a comprehensive and formal account for the refinement of SysML, as it is currently available and supported in commercial tools.

References

ABDELHALIM, I. et al. Formal verification of Tokeneer behaviours modelled in fUML using CSP‖. In: FORMAL ENGINEERING METHODS AND SOFTWARE ENGINEERING, 12., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2010. p.371–387. (ICFEM'10).

ABDELHALIM, I. et al. Towards a Practical Approach to Check UML/fUML Models Consistency Using CSP. In: ICFEM. **Anais...** Springer, 2011. p.33–48. (Lecture Notes in Computer Science, v.6991).

ABDELHALIM, I. et al. An Optimization Approach for Effective Formalized fUML Model Checking. In: SEFM. **Anais...** Springer, 2012. p.248–262. (Lecture Notes in Computer Science, v.7504).

ABDELHALIM, I. et al. An integrated framework for checking the behaviour of fUML models using CSP. **International Journal on Software Tools for Technology Transfer**, [S.l.], v.15, n.4, p.375–396, 2013.

ATEGO. **Artisan Studio**. 2013.

BACK, R. J.; WRIGHT, J. von. **Refinement Calculus**: a systematic introduction. [S.l.]: Springer-Verlag New York, Inc., 1998. (Texts in Computer Science).

BARESI, L. et al. A Logic-based Semantics for the Verification of Multi-diagram UML Models. **SIGSOFT Softw. Eng. Notes**, New York, NY, USA, v.37, n.4, p.1–8, July 2012.

BENGTSSON, J. et al. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In: WORKSHOP ON VERIFICATION AND CONTROL OF HYBRID SYSTEMS III. **Proceedings...** Springer–Verlag, 1995. n.1066, p.232–243. (Lecture Notes in Computer Science).

BERARDI, D. et al. Reasoning on UML Class Diagrams. **Artificial Intelligence**, [S.l.], v.168, n.1–2, p.70–118, 2005.

BISZTRAY, D. et al. **Case study**: uml to csp transformation. [S.l.]: University of Leicester, UK, 2007.

BOARD, I. T. **Systems Engineering Handbook**. INCOSE-TP-2003-002-03.

BOEHM, B. W. **Software Engineering Economics**. Englewood Cliffs, NJ: Prentice Hall, 1981.

BREU, R. et al. Systems, Views and Models of UML. In: UML WORKSHOP. **Anais...** [S.l.: s.n.], 1997. p.93–108.

BROY, M. et al. **Semantics of UML – Towards a System Model for UML**: the structural data model. [S.l.]: Institut für Informatik, Technische Universität München, 2006. (TUM-I0612).

BROY, M. et al. **Semantics of UML – Towards a System Model for UML**: the control model. [S.l.]: Institut für Informatik, Technische Universität München, 2007. (TUM-I0710).

BROY, M. et al. **Semantics of UML – Towards a System Model for UML**: the state machine model. [S.l.]: Institut für Informatik, Technische Universität München, 2007. (TUM-I0711).

BRYANS, J. et al. **CML Definition 4**. [S.l.]: COMPASS Deliverable, 2014. (D23.5).

CAFÉ, D. C. et al. Multi-Paradigm Semantics for Simulating SysML Models using SystemC-AMS. **Proceedings of the Forum on Specification & Design Languages**, [S.l.], Sept. 2013.

CAVARRA, A.; BOWLES, J. Combining Sequence Diagrams and OCL for Liveness. **Electron. Notes Theor. Comput. Sci.**, Amsterdam, The Netherlands, The Netherlands, v.115, p.19–38, Jan. 2005.

CENGARLE, M. V. et al. **A.**: operational semantics of uml 2.0 interactions. [S.l.]: Technische Universitat Munchen, 2005.

CHOUALI, S.; HAMMAD, A. Formal verification of components assembly based on SysML and interface automata. **ISSE**, [S.l.], v.7, n.4, p.265–274, 2011.

CLARKE, E. M. et al. **Model Checking**. Cambridge, Massachusetts: The MIT Press, 1999.

CLARKE, E. M. et al. Model Checking and the State Explosion Problem. In: MEYER, B.; NORDIO, M. (Ed.). **Tools for Practical Software Verification**. [S.l.]: Springer Berlin Heidelberg, 2012. p.1–30. (Lecture Notes in Computer Science, v.7682).

COLEMAN, J. et al. COMPASS Tool Vision for a System of Systems Collaborative Development Environment. In: INTERNATIONAL CONFERENCE ON SYSTEM OF SYSTEMS ENGINEERING, 7. **Anais...** [S.l.: s.n.], 2012. p.451–456.

DAN, L.; DANNING, L. Towards a Formal Behavioral Semantics for UML Interactions. In: THIRD INTERNATIONAL SYMPOSIUM ON INFORMATION SCIENCE AND ENGINEERING, 2010. **Proceedings...** IEEE, 2010. p.213–218. (ISISE '10).

DAVIES, J.; CRICHTON, C. Concurrency and Refinement in the Unified Modeling Language. **Formal Aspects of Computing**, [S.l.], v.15, n.2-3, p.118–145, 2003.

DEBBABI, M. et al. **Verification and Validation in Systems Engineering - Assessing UML / SysML Design Models**. [S.l.]: Springer, 2010. 1–248p.

DIJKSTRA, E. W. Guarded commands, nondeterminacy and formal derivation of programs. **Commun. ACM**, New York, NY, USA, v.18, n.8, p.453–457, 1975.

DING, S.; TANG, S.-Q. An approach for formal representation of SysML block diagram with description logic SHIOQ(D). In: INTERNATIONAL CONFERENCE ON INDUSTRIAL AND INFORMATION SYSTEMS, 2. **Proceedings...** [S.l.: s.n.], 2010. v.2, p.259–261.

EICHNER, C. et al. Compositional Semantics for UML 2.0 Sequence Diagrams Using Petri Nets. In: SDL FORUM. **Anais...** Springer, 2005. p.133–148. (LNCS, v.3530).

FARIAS, A. et al. Efficient CSP Z Data Abstraction. In: BOITEN, E. et al. (Ed.). **Integrated Formal Methods**. [S.l.]: Springer Berlin Heidelberg, 2004. p.108–127. (Lecture Notes in Computer Science, v.2999).

FDR: user manual and tutorial, version 2.28. [S.l.]: Formal Systems (Europe) Ltd, 1999.

FITZGERALD, J.; LARSEN, P. G. **Modelling Systems**: practical tools and techniques in software development. [S.l.]: Cambridge University Press, 2009.

FOSTER, S.; WOODCOCK, J. **A Dwarf Signal in CML**. [S.l.]: COMPASS, 2013.

FOWLER, M. **Refactoring**: improving the design of existing code. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

FRIEDENTHAL, S. et al. **A Practical Guide to SysML**: the systems modeling language. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. 2nd edition.

GIBSON-ROBINSON, T. et al. FDR3: a modern refinement checker for csp. In: ABRAHAM, E.; HAVELUND, K. (Ed.). **Tools and Algorithms for the Construction and Analysis of Systems**. [S.l.]: Springer Berlin Heidelberg, 2014. p.187–201. (Lecture Notes in Computer Science, v.8413).

GOUBAULT-LARRECQ, J.; MACKIE, I. **Proof Theory and Automated Deduction**. [S.l.]: Kluwer Academic Publishers, 1997. (Applied Logic Series, v.6).

GRAVES, H. Integrating SysML and OWL. In: OWL: EXPERIENCES AND DIRECTIONS. **Proceedings. . .** [S.l.: s.n.], 2009.

GRAVES, H. Modeling Structure in Description Logic. In: INTERNATIONAL WORKSHOP ON DESCRIPTION LOGICS (DL2011), Barcelona, Spain. **Proceedings. . .** [S.l.: s.n.], 2011.

GRAVES, H. Integrating Reasoning with SysML. In: INCOSE SYMPOSIUM, Rome, Italy. **Anais. . .** [S.l.: s.n.], 2012.

GRAVES, H.; BIJAN, Y. Using formal methods with SysML in aerospace design and engineering. **Ann. Math. Artif. Intel.**, [S.l.], p.1–50, 2011.

GRAVES, H.; BIJAN, Y. Using formal methods with SysML in aerospace design and engineering. **Annals of Mathematics and Artificial Intelligence**, [S.l.], v.63, n.1, p.53–102, 2011.

GRÜNINGER, M.; MENZEL, C. The Process Specification Language (PSL) Theory and Applications. **AI Magazine**, [S.l.], v.24, n.3, p.63–74, 2003.

HAMILTON, M. H. et al. **A Formal Universal Systems Semantics for SysML**. 2007.

HAUGEN, O. et al. STAIRS towards formal design with sequence diagrams. **Software and System Modeling**, [S.l.], v.4, n.4, p.355–367, 2005.

HOARE, C. A. R. **Communicating and Sequential Processes**. [S.l.]: Prentice Hall, 1985.

HOARE, T.; HE, J. **Unifying Theories of Programming**. [S.l.]: Prentice Hall, 1998.

HOLT, J.; PERRY, S. **SysML for Systems Engineering**. [S.l.]: IET, 2008.

IBM. **Rational Rhapsody Architect for Systems Engineers**. 2013.

ISO. **Information technology – Z formal specification notation – Syntax, type system and semantics**. [S.l.]: International Organization for Standardization, 2002. (ISO/IEC 13568).

JAMSHIDI, M.; JAMSHIDI, M. **Systems of Systems Engineering**: principles and applications. [S.l.]: CRC PressINC, 2009.

KNAPP, A.; WUTTKE, J. Model Checking of UML 2.0 Interactions. In: **Models in Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2007. p.42–51. (LNCS, v.4364).

KNORRECK, D. et al. TEPE: a sysml language for time-constrained property modeling and formal verification. **ACM SIGSOFT Software Engineering Notes**, [S.l.], v.36, n.1, p.1–8, 2011.

KUSKE, S. et al. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In: BUTLER, M. et al. (Ed.). **Integrated Formal Methods**. [S.l.]: Springer Berlin Heidelberg, 2002. p.11–28. (Lecture Notes in Computer Science, v.2335).

LANO, K.; EVANS, A. Rigorous Development in UML. In: FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING (FASE). **Proceedings. . .** [S.l.: s.n.], 1999. p.129–144.

LAURENT, Y. et al. Formalization of fUML: an application to process verification. In: JARKE, M. et al. (Ed.). **Advanced Information Systems Engineering**. [S.l.]: Springer International Publishing, 2014. p.347–363. (Lecture Notes in Computer Science, v.8484).

LAZIC, R.; NOWAK, D. On a Semantic Definition of Data Independence. In: HOFMANN, M. (Ed.). **Typed Lambda Calculi and Applications**. [S.l.]: Springer Berlin Heidelberg, 2003. p.226–240. (Lecture Notes in Computer Science, v.2701).

LI, X. et al. A Formal Semantics of UML Sequence Diagram. In: AUSTRALIAN SOFTWARE ENGINEERING CONFERENCE, 2004., Washington, DC, USA. **Proceedings. . .** IEEE Computer Society, 2004. p.168–. (ASWEC ’04).

LILIUS, J.; PALTOR, I. P. **The Semantics of UML State Machines**. [S.l.]: Turku Centre for Computer Science, 1999.

LIMA, L. **Analysis of CML Models of SysML Diagrams**. Available in: <http://www.cin.ufpe.br/~lal2/sysml2cml>: University of York, 2014. Accessed on: 30 Sep. 2015.

LIMA, L. et al. A Formal Semantics for SysML Activity Diagrams. In: IYODA, J.; MOURA, L. (Ed.). **Formal Methods**: foundations and applications. [S.l.]: Springer Berlin Heidelberg, 2013. p.179–194. (Lecture Notes in Computer Science, v.8195).

LIMA, L. et al. A Formal Semantics for Sequence Diagrams and a Strategy for System Analysis. In: INTERNATIONAL CONFERENCE ON MODEL-DRIVEN ENGINEERING AND SOFTWARE DEVELOPMENT (MODELSWARD). **Proceedings. . .** [S.l.: s.n.], 2014.

LIMA, L. et al. An integrated semantics for reasoning about SysML design models using refinement. **Software & Systems Modeling**, [S.l.], p.1–28, 2015.

LUND, M. S. **Operational analysis of sequence diagram specifications**. 2007. Ph.D. Thesis — University of Oslo.

MAKARTETSKIY, D.; SISTO, R. An approach to refinement checking of SysML requirements. In: EMERGING TECHNOLOGIES FACTORY AUTOMATION (ETFA), 2011 IEEE 16TH CONFERENCE ON. **Anais. . .** [S.l.: s.n.], 2011. p.1–4.

MENG, S. et al. On the semantics and refinement of UML statecharts: a coalgebraic view. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND FORMAL METHODS, 2. **Proceedings. . .** IEEE Computer Society, 2004.

MICSKEI, Z.; WAESELYNCK, H. The many meanings of UML 2 Sequence Diagrams: a survey. **Softw. Syst. Model.**, Secaucus, NJ, USA, v.10, n.4, p.489–514, Oct. 2011.

MILLER, S. P. et al. Software Model Checking Takes off. **Commun. ACM**, New York, NY, USA, v.53, n.2, p.58–64, Feb. 2010.

MIYAZAWA, A.; CAVALCANTI, A. Formal Refinement in SysML. In: ALBERT, E.; SEKERINSKI, E. (Ed.). **IFM**. [S.l.]: Springer, 2014. p.155–170. (LNCS, v.8739).

MIYAZAWA, A. et al. Formal Models of SysML Blocks. In: GROVES, L.; SUN, J. (Ed.). **Formal Methods and Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2013. p.249–264. (Lecture Notes in Computer Science, v.8144).

MIYAZAWA, A. et al. **Final Report on Combining SysML and CML**. [S.l.]: COMPASS, 2013.

MOTA, A. et al. Model checking CML: tool development and industrial applications. **Formal Aspects of Computing**, [S.l.], v.27, n.5, p.975–1001, 2015.

OLIVEIRA, M. et al. Model-checking *Circus* State-Rich Specifications. In: IFM 2014: INTEGRATED FORMAL METHODS. **Anais...** Springer International Publishing Switzerland, 2014. p.39 —- 54. (Lecture Notes in Computer Science, v.8739).

OMG. **MOF QVT Final Adopted Specification**. [S.l.]: OMG, 2005. n.OMG document formal/2005-11-01.

OMG. **OMG Systems Modeling Language (OMG SysML^{TM})**. [S.l.]: Object Management Group, 2010. OMG Document Number: formal/2010-06-02.

OMG. **OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.3**. [S.l.]: OMG, 2010.

OMG. **OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1**. [S.l.]: Object Management Group, 2011.

OMG. **OMG Systems Modeling Language (OMG SysML^{TM})**. [S.l.]: Object Management Group, 2012. OMG Document Number: formal/12-06-02.

OMG. **Semantics of a Foundational Subset for Executable UML Models (FUML)**. [S.l.]: Object Management Group, 2013. OMG Document Number: formal/2013-08-06.

OMG. **Precise Semantics Of UML Composite Structures (PSCS)**. [S.l.]: Object Management Group, 2014. OMG Document Number: 1.0 - Beta 1.

PANDA, P. R. SystemC. In: ISSS. **Anais...** [S.l.: s.n.], 2001. p.75–80.

RAMOS, R. et al. A Semantics for UML-RT Active Classes via Mapping into *Circus*. In: FMOODS. **Anais...** Springer, 2005. p.99–114. (Lecture Notes in Computer Science, v.3535).

RASCH, H.; WEHRHEIM, H. Checking Consistency in UML Diagramms: classes and state machines. In: NAJM, E. et al. (Ed.). **Formal Methods for Open Object-Based Distributed Systems (6th FMOODS'03)**. Paris, France: Springer-Verlag (Berlin/New York), 2003. p.229–243. (Lecture Notes in Computer Science (LNCS), v.2884).

RASCH, H.; WEHRHEIM, H. Checking the validity of scenarios in UML models. In: INTERNATIONAL CONFERENCE ON FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS, 7. **Proceedings...** [S.l.: s.n.], 2005. p.67–82.

ROSCOE, A. **Understanding Concurrent Systems**. 1st.ed. New York, NY, USA: Springer-Verlag New York, Inc., 2010.

ROSCOE, A. W. **The Theory and Practice of Concurrency**. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

ROSCOE, A. W. **The Theory and Practice of Concurrency**. New York: Prentice-Hall, 1998. (Prentice-Hall Series in Computer Science). Oxford.

SCATTERGOOD, B.; ARMSTRONG, P. **CSPM**: a reference manual. [S.l.: s.n.], 2011.

SHEN, H. et al. Formalize UML 2 Sequence Diagrams. In: IEEE HIGH ASSURANCE SYSTEMS ENGINEERING SYMPOSIUM, 2008., Washington, DC, USA. **Proceedings...** [S.l.: s.n.], 2008. p.437–440. (HASE '08).

SMITH, G. **The Object-Z Specification Language**. [S.l.]: Kluwer Academic Publishers, 2000.

STORRLE, H. **Trace Semantics of Interactions in UML 2.0 Abstract**. 2004.

STORRLE, H. **Trace Semantics of Interactions in UML 2.0 Abstract**. Pre-print.

SUN, J. et al. Model checking csp revisited: introducing a process analysis toolkit. In: IN ISOLA 2008. **Anais...** Springer, 2008. p.307–322.

SYSTEM, S. **Sparx Systems' Enterprise Architect supports the Systems Modeling Language**. 2013.

VACHOUX, A. et al. Analog and mixed signal modelling with SystemC-AMS. In: ISCAS (3). **Anais...** [S.l.: s.n.], 2003. p.914–917.

WARMER, J.; KLEPPE, A. **The Object Constraint Language**: getting your models ready for mda. 2.ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

WOODCOCK, J. C. P.; CAVALCANTI, A. L. C. The Semantics of *Circus*. In: ZB 2002: FORMAL SPECIFICATION AND DEVELOPMENT IN Z AND B. **Anais...** Springer-Verlag, 2002. p.184—203. (Lecture Notes in Computer Science, v.2272).

WOODCOCK, J. et al. Features of CML: a formal modelling language for systems of systems. In: INTERNATIONAL CONFERENCE ON SYSTEM OF SYSTEM ENGINEERING, 7. **Proceedings...** [S.l.: s.n.], 2012a. (IEEE Systems Journal, v.6).

WOODCOCK, J. et al. **CML Definition 0**. [S.l.]: COMPASS Deliverable, 2012b. (D23.1).

WOODCOCK, J. et al. **CML Definition 3 - Denotational Semantics**. [S.l.]: COMPASS Deliverable, 2013. (D23.4a).

XU, D. et al. Towards Formalizing UML Activity Diagrams in CSP. In: INTERNATIONAL SYMPOSIUM ON COMPUTER SCIENCE AND COMPUTATIONAL TECHNOLOGY - VOLUME 02, 2008., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2008. p.450–453. (ISCSCT '08).

XU, D. et al. Model Checking UML Activity Diagrams in FDR. **ACIS International Conference on Computer and Information Science**, Los Alamitos, CA, USA, p.1035–1040, 2009.

Appendix

A Translation rules

Rule A.1 declares the type `Bag`, which is used to store the operation calls that have been received by a block. This type is used by block processes as described in Chapter 4 (rules 4.20 and 4.21). The type `Bag` models a bag (i.e, an unordered collection of objects where repetition is allowed) of tokens, which is used in our semantics to store names of operations. Additionally, three functions are declared: `in_bag`, `bunion` and `bdiff`. These functions are the bag equivalent of set membership, union and difference respectively.

Rule A.1: declare_bag

```
declare_bag(): seq of class paragraph =
   "types"
   "public Bag = map token to nat"
   "values"
      "public empty_bag = {>}"
   "functions"
      "public in_bag: token * Bag -> bool"
      "in_bag(o,b) == "
         "exists n in set dom(b) @ b(n) > 0 and o = n"

      "public bunion: Bag * Bag -> Bag"
      "bunion(m1,m2) == "
         "{x |-> y | x in set (dom m1 inter dom m2),"
            "y: nat @ y = m1(x)+m2(x)}"
         "munion"
         "{x |-> y | x in set dom m1, y: nat @"
            "x not in set dom m2 and y = m1(x)}"
         "munion"
         "{x |-> y | x in set dom m2, y: nat @"
            "x not in set dom m1 and y = m2(x)}"

      "public bdiff: Bag * Bag -> Bag"
      "bdiff(m1,m2) == "
         "{x |-> y | x in set (dom m1 inter dom m2),
            y: nat
            @ y = if m1(x)-m2(x) > 0
                 then m1(x)-m2(x)
                 else 0}"
         "munion"
         "{x |-> y | x in set dom m1, y: nat @"
            "x not in set dom m2 and y = m1(x)}"
```

Rule A.2 defines some of the channels used in the model. For each connector used in the model, a channel named after it is defined as communicating a natural number, to identifiers and a message. The following defined channels are used for internal control in the state machine semantics. All these channels are internal to the model.

Rule A.2: define_stm_channels

```
define_stm_channels(m: SysML model): program =
   "channels"

   for each c in m.AllConnectors do
      name(c)"_op: nat*ID*ID*OPS"
      name(c)"_sig: nat*ID*ID*S"
   end for

      "instate, setstate: ID*ID*bool"
      "enter,entered,exit,exited,cancel,final_state: ID*ID"
      "enabled: ID*bool"
      "completion: ID"
      "fire,fired"
```

Rule A.3 defines auxiliary functions used along the model. The `DL_or` function implements a logic operation OR for the type DL considering it has a third value `<defer>`. Function `prefix` verifies if a given element identifier is a prefix of the second identifier. It is used to check if a given element is in the hierarchy of another (e.g., a block is part of another block, a transition is part of a state machine, etc). The function `drop_two` removes the last two tokens in the sequence that identifies an element.

Rule A.3: declare_aux_functions

```
declare_aux_functions(): program =
      "functions
      DL_or(a: DL, b: DL) c: DL
      post (is_bool(a) and is_bool(b) => c = a or b)
         and (not is_bool(a) or not is_bool(b) => (
            ((a = true) or (b = true) => (c = true))
            and ((not (a = true) and not (b = true))
               => c = <defer>)
         ))
```

```
    prefix: ID*ID->bool
    prefix(p, q) == len p <= len q and (forall i in set inds p @ p(i) =
        q(i))

    drop_two(x: ID) c: ID
    pre len x >= 2
    post c = reverse(tl(tl(reverse(x))))"
```

Rule A.4 defines a channel set with the alphabet of communication between the activities of a block and their owner block.

Rule A.4: t_activities_chanset

```
t_activities_chanset(ads: seq of Activity, block: Block): chanset
    expression =
   "{|" name(block)"_hasevent, "name(block)"_getevent |}"
   for ad in seq ads do
      " union {| startActivity_"name(ad)", endActivity_"name(ad)",
          interruptActivity_"name(ad) "|}"
      for action in seq ad.Nodes(Action.Type) do
         if action.Type == CallOperation.Type then
            " union {|
               "name(action.target.type)"_op.m.($id^[mk_token(\"acts\")]^
                         [mk_token(\""name(ad)"\")]^
                         [mk_token(\""name(action)"\")]) | m: nat |}"
         else if action.Type == SendSignal.Type then
            " union {|
               "name(action.target.Type)"_sig.m.($id^[mk_token(\"acts\")]^
                         [mk_token(\""name(ad)"\")]^
                         [mk_token(\""name(action)"\")]) | m:nat |}"
         end if
      end for
   end for

   for a in set block.AllAttributes do
     " union {|"name(b)"_get_"a.name".x.$id | x: ID @
         prefix($id^[mk_token(\"acts\")],x)|}"
     " union {|"name(b)"_set_"a.name".x.$id | x: ID @
         prefix($id^[mk_token(\"acts\")],x)|}"
   end for
```

Rule A.5 translates a statement written in the action language (described in Chapter 3) to CML.

Rule A.5: t_action

```
t_action(a: Action, b: Block, n: Name, trs: seq of Trigger): action =
   if a = sep "?" as then sep "?"{t_action(a,b,n,trs) | a in set as}
   else t_statecopy(b,n) ";"
      rename_parameters(t_simple_action(a,b,n,trs),trs) ";"
      t_stateupdate(b,n) ")"
   end if


where
   1. ? stands for any constructor of a CML statement (e.g., ;)
```

Rule A.6 is an auxiliary function used to retrieve the current values of the attributes of a block.

Rule A.6: t_statecopy

```
t_statecopy(b:Block, n: Name): action =
   if b.Attributes.size() = 0 then "(Skip"
   else "(dcl " sep "," {a.name":" t_types(a.type) | a in set
      b.Attributes} "@"
         t_readstate(b.Attributes,b,n)
   end if
```

Rule A.7 is used by Rule A.6 to read the current values of the attributes of a block.

Rule A.7: t_readstate

```
t_readstate(as: set of Attribute, b:Block, n: Name): action =
    if as.size() = 0 then "Skip"
    else if as = {a} then
        "("name(b)"_get_"a.name".($id^[mk_token(\""n"\")]).
            (reverse(tl(tl(reverse($id)))))?x -> "a.name" := x)"
    else let a in set as in
        "(("name(b)"_get_"a.name".($id^[mk_token(\""n"\")]).
            (reverse(tl(tl(reverse($id)))))?x -> "a.name" := x)"
        "[||{"a.name"} | {"sep "," {x.name | x in set as\{a}}"} ||]"
        t_readstate(as\{a},b,n) ")"
    end if
```

Rule A.8 applies a rename in the parameters used in an action statement in order to use the CML event notation for recovering the message of the event. The notation `e.#4` describes an access to the fourth argument of an event, which corresponds to the message (operation call or signal) received by the block. This event follows has the type `E` as defined in Rule 3.3 (on page 55) in Chapter 3.

Rule A.8: rename_parameters

```
rename_parameters(a: Action, trs: seq of Trigger): action =
   let old_names = sep "," dunion{t:elements trs @ parameters(t.event)}
      new_names = sep "," {n: dunion{t:elements trs @
         parameters(t.event)} @ "(e.#4."n")"}
   in
      a[new_names/old_names]
```

Rule A.9 defines the translations to CML of the possible action language statements defined by our guidelines in Chapter 3.

Rule A.9: t_simple_action

```
t_simple_action(a: Action,b: Block, n: Name, trs: seq of Trigger): action
   =
   if a = (Op(ps)) then
      if Op is an operation of the SysML model then
         name(b)"_op?w.($id^[mk_token(\""n"\")]).
             (reverse(tl(tl(reverse($id))))."
            "mk_"Op.name"_I(mk_token(\""Op.name"_I\")" if ps <> "" then
               "," ps end if " ) -> "
         name(b)"_op.w.($id^[mk_token(\""n"\")]).
             (reverse(tl(tl(reverse($id)))))?"
            "x:(x.$id = mk_token(\""Op.name"_O\")) -> Skip"
      else -- Op is an activity of the SysML model
         "startActivity_"name(Op)".($id^[mk_token(\""n"\")])"for i in set
            inds(ps) do "."ps(i) end for" -> "
         "endActivity_"name(Op)".($id^[mk_token(\""n"\")])"for p in set
            Op.OutputParameters do "?"p end for " -> Skip"
      end if
   else if a = (v := Op(ps)) then
      if Op is an operation of the SysML model then
         name(b)"_op?w.($id^[mk_token(\""n"\")]).
             (reverse(tl(tl(reverse($id)))))."
            "mk_"Op.name"_I(mk_token(\""Op.name"_I\")" if ps <> "" then
               "," ps end if " ) -> "
         name(b)"_op.w.($id^[mk_token(\""n"\")]).
             (reverse(tl(tl(reverse($id)))))?"
            "x:(x.$id = mk_token(\""Op.name"_O\"))-> "v":=(x._ret)"
      else -- Op is an activity of the SysML model (there is an
          assumption that Op has a single output
         "startActivity_"name(Op)".($id^[mk_token(\""n"\")])"for i in set
            inds(ps) do "."ps(i) end for" -> "
         "endActivity_"name(Op)".($id^[mk_token(\""n"\")])?x -> "v":=x"
      end if
   else if a = (return v) and len(trs) = 1 and trs(1) is Operation then
```

```
      name(b)"_op.(e.#1).(e.#2).(e.#3).mk_"trs(1).name"_O("
         "mk_token(\""trs(1).name"_O\"),"v")-> Skip"
   else if a = (return) and len(trs) = 1 and trs(1) is Operation then
      name(b)"_op.(e.#1).(e.#2).(e.#3).mk_"trs(1).name"_O("
         "mk_token(\""trs(1).name"_O\"))-> Skip"
   else if a = ([e]&S) and e is an Expression and S is a Statement then
      "[" e "] &" t_simple_action(S,b,n,trs)
   else a
   end if


where
   1. Op is the name of an operation
   2. ps is a comma separated sequence of expressions
   3. v is variable name
   4. e is an expression
   5. S is an action
   6. Op.OutputParameters is the sequence of output parameters of an
       activity Op
```

Rule A.10 updates the values of attributes of a block.

Rule A.10: t_stateupdate

```
t_stateupdate(as: set of Attributes; b: Block; n: Name): action =
   if as.size() = 0
   then "Skip"
   else "(" sep "|||" {name(b)"_set_"a.name".($id^[mk_token(\""n"\")]).
        (reverse(tl(tl(reverse($id))))))!"a.name" -> Skip" | a in set as}
           ")"
   end if
```

Rule A.11 defines the function that communicates events from a set of events received as argument indefinitely.

Rule A.11: t_RUN

```
RUN(S: set of channel): action paragraph =
    "mu X @ " sep "[]" {ev "-> X" | ev in set S}
```

A.1 Rules for Activity

In this this section we present the remaining rules used for the definition of a semantics for SysML activities.

A.1.1 Call Behaviour Action Parallelism

We use the function `t_ad_cba_parallel` (described in Rule A.12), if there is at least one Call Behaviour action in a diagram, to introduce channels for synchronisation between the diagram model and call behaviour actions models. They run in parallel without synchronising. The function `t_ad_cba_parallel` has an activity identification and a block instance as arguments.

The channels whose names begin with `startActivity_CBA_` are used for synchronisation between a diagram model and a Call Behaviour action (CBA) model, so that the CBA model can start its execution. The channel name is composed by the activity name given by `name(ad)`, the block id `$id`, and the Call Behaviour action index. The channels whose names begin with `endActivity_CBA_` are used for synchronisation between a CBA action model and a diagram model, indicating the end of a CBA execution. Every CBA action model has these two channels.

Rule A.12: t_ad_cba_parallel

```
t_ad_cba_parallel(ad: Activity, block: Block): program paragraph =

   if ad.Nodes(CallBehaviour.Type).size > 0
      "[|{|" sep "," { "startActivity_CBA_"name(ad)".$id."cba.index",
         endActivity_CBA_"name(ad)".$id."cba.index | cba in seq
         ad.Nodes(CallBehaviour.Type) } "|}|]"
      "(" sep "|||" { "(ad_"name(cba_ad)"($id))
         [[ startActivity_"name(cba_ad)".$id <-
            startActivity_CBA_"name(ad)".$id."cba.index","
            "endActivity_"name(cba_ad)".$id <-
               endActivity_CBA_"name(ad)".$id."cba.index" ]]"
                  | cba in seq ad.Nodes(CallBehaviour.Type), cba_ad ==
                     cba.behaviour.activity}")"
   end if
```

The CML actions for the Call Behaviour actions are executed in interleaving. The channel name that begins with `startActivity_` of each action is renamed so that it gets the same name as used in the set of synchronising events. A similar renaming is done with respect to channels with names starting with `endActivity_`.

Next, we illustrate in the following extract the application of rules 5.5 and A.12 by showing the generated CML code for the **TreatEmergencyCall** and **BroadcastCall** activity diagrams (figures 5.5 and 5.6 on pages 104 and 105, respectively). Note that as the latter does not have call behaviour actions it is reduced to their internal CML process.

```
channels
...
process ad_internal_BroadcastCall = ...

process ad_BroadcastCall = val $id: ID @ ad_internal_BroadcastCall($id)

process ad_internal_TreatEmergencyCall = ...

process ad_TreatEmergencyCall = val $id: ID @
ad_internal_TreatEmergencyCall($id)
[|{|startActivity_CBA_TreatEmergencyCall.$id.2,
endActivity_CBA_TreatEmergencyCall.$id.2|}|]
((ad_BroadcastCall($id))[[startActivity_BroadcastCall.$id <-
startActivity_CBA_TreatEmergencyCall.$id.2,
endActivity_BroadcastCall.$id <-
endActivity_CBA_TreatEmergencyCall.$id.2]]
)
```

In this extract, first, we have the definition of the process for the internal representation of activity **BroadcastCall** (`ad_internal_BroadcastCall`). This process is used in the definition of the main process for this activity (`ad_BroadcastCall`). The activity **BroadcastCall** does not have any Call Behaviour action, then, its main process is the same as the internal process. On the other hand, the definition of the main process for the activity **TreatEmergencyCall** (`ad_TreatEmergencyCall`) is the parallel composition of its internal representation (`ad_internal_TreatEmergencyCall`) and the process of the Call Behaviour action used in this activity, which refers to activity **BroadcastCall** process previously defined. We rename the events to start and end the activity **BroadcastCall** according to the events of the Call Behaviour action. In this case, it is parametrised by the ID of the enclosing activity (`$id`) and the index of the Call Behaviour action, which in this case is 2.

A.1.2 Start Activity

The CML action `START_ACTIVITY` is defined by Rule A.13. It offers to the environment an event for synchronisation. To start the execution of an activity diagram, it is necessary to synchronise on a channel named `startActivity_` with a suffix composed by the activity name `name(ad)`. This channel synchronises with an entity that invokes the activity, which can be for example the block instance or the block state machine. That is why it declares a value-result parameter (**`vres`**) to store the source that invoked the diagram in order to respond to this same entity once the diagram finishes. The environment must provide values for each activity parameter. The channel parameter names are the same as the activity parameter names. Notice that only activity parameters with no incoming edges are used; this ensures that they are used for input. For instance, the activity of Figure 5.5 (on page 104) has an input parameter (**call**) because such a parameter has only outgoing edges. On the other hand, the parameter of the activity displayed in Figure 8.11 (on page 161) has only incoming edges, hence, it is not used in the channel `startActivity_` After communicating the input values, such values are assigned to local variables whose purpose is to store the input data. We use the CML construct **`atomic`** in order to make all assignments one single command. This data is used by the object nodes defined in Rule A.37. The definition of these variables is detailed in Rule 5.9. The names of these variables are the same names as the input activity parameter nodes. This happens to allow a sequential composition, which is detailed in Rule 5.9 where the `START_ACTIVITY` CML action is used.

Rule A.13: t_start_activity

```
t_start_activity(ad: Activity, block: Block): seq of action =
    "START_ACTIVITY = "
          "vres $source: ID" {", vres "name(param)": "t_types(param) |
             param in seq ad.Nodes(ActivityParameterNodes.Type) and
             param.IncomingEdges.size() == 0 } " @ "
       "startActivity_" name(ad)"?$s"
       if (card {param | param in seq
          ad.Nodes(ActivityParameterNodes.Type) and
          param.IncomingEdges.size() == 0 } > 0) then
          "?"
       end if
       sep "?" {"x_"name(param) | param in seq
          ad.Nodes(ActivityParameterNodes.Type) and
          param.IncomingEdges.size() == 0 }
    " -> " if card {param | param in seq
       ad.Nodes(ActivityParameterNodes.Type) and
       param.IncomingEdges.size() == 0 } > 0 then
             "atomic($source := $s;" sep ";" {name(param)" :=
                x_"name(param) | param in seq
                ad.Nodes(ActivityParameterNodes.Type) and
                param.IncomingEdges.size() == 0 }");"
          else
             "($source := $s);"
          end if
    " Skip "
```

The next extract shows the application of Rule A.13 to activity **Add** shown in Figure 5.3 on page 103. When we have only one CML action in the interleaving parallelism (`|||`), it can be reduced to this action, that is why we only have `(item := x_item -> Skip)` instead of the parallelism.

```
process ad_internal_Add = val Buffer_id: ID @ begin
```

```
chansets
...
actions
START_ACTIVITY = vres $source: ID, vres item: Item @
startActivity_Add?$s?x_item -> atomic($source := $s;item := x_item);Skip
...
end
```

A.1.3 Interruptible Regions

An interruptible region allows a subset of the actions to be interrupted by the destruction of their tokens; the other actions outside the interruptible region remain executing. A mechanism called interrupting edge indicates when an interruptible region is to be interrupted: a token is accepted by the interrupting edge. We use the function `t_interruptible_regions` (Rule A.14) to translate interruptible regions. A CML action whose name begins with `InterruptibleRegion_` and whose suffix is the interruptible region index is introduced for every interruptible region of an activity diagram of a block instance. The CML action is recursive (defined using the μ construct). When an interrupting edge is to be traversed, the actions inside an interruptible region must finish execution. This is modelled by the external choice of all interrupting edges that leave the region. The synchronisation should be accomplished via one of the events defined by `control.[edge.index]` or by those channels named `obj_`. Then, an event over a channel `interrupted` is available to the environment, defining an interrupting edge of an interruptible region. This channel is specific to the model of a diagram, a block instance (`$id`), and an interruptible region (`intRegion.index`). After synchronisation, the CML action behaves as **`Skip`**. The external choice is sequentially composed with the recursive call `X`, indicating that the region that was interrupted is now available to start execution again.

The CML action `InterruptibleRegions` is defined by the parallelism of all CML actions introduced for interruptible regions. As these regions execute asynchronously, they are interleaved.

Rule A.14: t_interruptible_regions

```
t_interruptible_regions(ad: Activity, block: Block): seq of action =
   for intRegion in seq ad.group and
      intRegion.isInterruptibleActivityRegion() do
      "InterruptibleRegion_"intRegion.index" = mu X @ (("
         for edge in seq intRegion.interruptingEdge sep "[]" do
            if edge.isControl() then
               "control."edge.index
            else
               "obj_"name(ad)"_"edge.index"?x"
            end if
```

```
              "-> interrupted."id(ad)".$id."intRegion.index" -> Skip"
           end for
        ");X)"
    end for

    "InterruptibleRegions = Skip"

    for intRegion in seq ad.group and
        intRegion.isInterruptibleActivityRegion() do
        " ||| InterruptibleRegion_"intRegion.index
    end for
```

A.1.4 Interrupt Activity Manager

Rule A.15 is responsible for controlling when an external entity (e.g., a state machine) interrupts an activity. This behaviour is not defined by the OMG, but is provided by our semantics in safe situations. When a call operation or a call behaviour is invoked, we must wait the termination of the action in order to allow the interruption of the activity, otherwise this interruption would lead to an erroneous state because the entity waiting for a return event of an operation would never be synchronised. We use a variable `i` to control when such interruption is possible. Every time one of the cited actions starts to execute its main behaviour, the variable is incremented. Once the execution finishes, the variable is decremented (see rules A.21 and A.26). Hence, while the value of the variable is zero, the activity may be interrupted. The `INT_ACT_MANAGER` action is used inside the main action (Rule 5.9).

Rule A.15: t_interrupt_activity_manager

```
t_interrupt_activity_manager(ad: Activity, block: Block): seq of action =

    "INT_ACT_MANAGER = (dcl i: nat := 0 @ (mu X @ (
                      inc?o -> i := i + 1;X
                      []
                      dec?o -> i := i - 1;X
                      []
                      [i = 0] & interruptActivity_"name(ad)"?x:
                          (prefix($id,x)) -> Skip)))"
```

A.1.5 Token Manager

The function `t_token_manager` (Rule A.16) introduces the CML action `TOKEN_MANAGER`, which models the protocol that controls the termination of the diagram according to its active tokens. `TOKEN_MANAGER` is a recursive action that has an initialisation (first `update` event), and then performs three behaviours. One of them changes the number of active tokens by receiving a communication through the `update` channel. Some nodes can increase or decrease the number of active tokens. Initial nodes, fork nodes and action nodes that have more outgoing edges than incoming edges are examples of nodes that increase the number of active nodes. On the other hand, flow final nodes, join nodes and action nodes are examples of nodes that decrease the number of active nodes. They communicate with `TOKEN_MANAGER` through the `update` channel. When an active flow reaches an activity final node all remaining flows must terminate. This is performed by the `clear` event. When the number of active tokens is zero, then the activity must terminate. Firing the `endDiagram` event interrupts all elements of the diagram as can be seen further in the translations of the nodes.

Rule A.16: t_token_manager

```
t_token_manager(ad: Activity, block: Block): seq of action =

   "TOKEN_MANAGER = update?o?x -> nTokens := x; mu X @ (
      update?o?x -> nTokens := nTokens + x;X
      []
      clear?o -> nTokens := 0; endDiagram."id(ad)" -> Skip
      []
      [nTokens = 0] & endDiagram."id(ad)" -> Skip)"
```

The next extract shows the application of Rule A.16 for activity **Add**.

```
...
process ad_internal_Add = val Buffer_id: ID @ begin
chansets
...
actions
...
TOKEN_MANAGER =  update?o?x -> nTokens := x; mu X @ (
update?o?x -> nTokens := nTokens + x;X
[]
clear?o -> nTokens := 0; endDiagram.Add -> Skip
[]
[nTokens = 0] & endDiagram.Add -> Skip)
...
end

...
```

The action `TOKEN_MANAGER` models the token semantics of activity diagrams. However, its relationship with the nodes is what determines the termination mechanism as described in the next sections where we describe action, control and object nodes.

A.1.6 Action Nodes

The function `t_action_node` (Rule A.17) translates an action node. It receives as arguments the action node to be translated, the activity diagram to which the action belongs, a block instance from which we translate the diagram, and the interruptible regions that enclose the node. This function introduces a CML action with a name composed by the SysML action name appended with the action node index.

In SysML, every action execution finishes when the executing diagram finishes execution. If the action is part of an interruptible region, its execution halts when the interrupting edge accepts a token, but the node is restarted and waits for tokens on its inputs once again. Notice that it is possible to have nested interruptible regions, so that when an interruptible region finishes its execution, actions of any nested interruptible region must also halt execution. The boolean variable `end_guard` indicates when an action can have its execution finished or halted. This variable is used as the guard of an action: if it is true, then the action after the guard is enabled. So, for every action, the function `t_action_node` declares `end_guard`, a boolean variable, with **`true`** as initial value. In order to avoid undesirable situations, this variable is falsified if the action is a Call Operation or Call Behaviour action. Therefore, we do not allow the interruption of Call Operation and Call Behaviour actions during their executions. Once the behaviour is completed, then the variable is set true, allowing the interruption once again.

The CML action for a SysML action is recursive. If there are any input or output pins from an action, we declare variables to hold the values communicated by input pins or values that are going to be used in output pins. The variable identifiers are defined by the names of the input and the output pins. The types of these variables are defined by the types of the input and the output pins.

If there are incoming edges to an action (control and object), we translate them by using the function `t_action_incoming_edges` (see Rule A.19) applied to the action and the diagram. When there is no incoming edge and the number of outgoing edges is greater than zero, then the action generates active tokens, which is performed by the `update` event. Then, the function `t_action_types(action,ad,block,regions)` (see Rule A.18) is called to translate the action according to its type. Next, we check if the number of arrows leaving the action is different from the number of the incoming ones, in case the latter is grater than zero. If it is true, then we must update the number of active tokens (*#outgoingEdges* minus *#incomingEdges*). If the number of outgoing edges is greater than zero, all the previous CML actions are sequentially composed with another CML action related to the outgoing edges. This is defined in function `t_action_outgoing_edges` (see Rule A.20. Notice that the variables initially declared are visible through the whole action.

A SysML action can stop its execution due to two situations: (1) an interrupting edge has accepted a token (and the action is part of the interrupting region) or (2) execution of the activity has finished. If the action belongs to at least one interruptible region, then we insert an interruption in a guarded action. The guard is the boolean variable `end_guard` that, if true, enables an external choice over the indices of interruptible regions of an activity, using the channel `interrupted`. If an interrupting edge accepts a token, an event synchronises with one of the channels available for external choice, then we have a recursive call.

Independently of the existence of interruptible regions, SysML actions finish when an executing activity to which the actions belong finishes. The action termination mechanism is defined by `/_\ end_guard & END_DIAGRAM` that takes control only when the guard `end_guard` value is **`true`**, then it behaves as `END_DIAGRAM` (defined in Rule 5.8 on page 112 of Chapter 5). The `END_DIAGRAM` behaviour synchronises on the `endDiagram` event, which can only be fired according to the `TOKEN_MANAGER` action described in Section A.1.5.

Rule A.17: t_action_node

```
t_action_node(action: ActivityNode, ad: Activity, block: Block, regions:
    seq of InterruptibleActivityRegion): action =
   name(action)"_"action.index" = (dcl end_guard: bool := true @ ( mu X @
       ("
   "("
   if action.input.size() > 0 or action.output.size() > 0
         "(dcl " sep "," {name(obj)": "t_types(obj) | obj in seq
            concat(action.input, action.output)}
         "@ ("
   end if
   if action.IncomingEdges.size > 0 then
      t_action_incoming_edges(action,ad)
   else if action.OutgoingEdges.size > 0 then
      "(update."id(action)"!"action.OutgoingEdges.size" -> Skip);"
   end if
   "("t_action_types(action,ad,block,regions)");"
   if action.IncomingEdges.size > 0 and action.OutgoingEdges.size !=
      action.IncomingEdges.size then
      "update."id(action)"!("action.OutgoingEdges.size -
         action.IncomingEdges.size") -> "
   end if
   if action.OutgoingEdges.size > 0 then
      t_action_outgoing_edges(action,ad)
   end if
   if action.IncomingEdges.size == 0 then
      "wait -> X "
   else
      "X "
   end if
   if action.input.size() > 0 or action.output.size() > 0
      ")))"
   else
      ")"
   end if

   if regions.size() > 0 then
      "/_\ [end_guard] & ([]i in set {"{intRegion.index | intRegion in
         seq regions}"} @ (interrupted."id(ad)".$id.i -> X))"
   end if
   ")) /_\ [end_guard] & END_DIAGRAM)"
```

A.1.7 Action Types

The function `t_action_types` (Rule A.18) receives as arguments an action node, the activity diagram to which the node belongs, a block instance, and the diagram interruptible regions. Based on the type of the activity node, this function selects an adequate function for translation. The following types of actions can be translated: call operation, opaque, accept event, send signal, value specification, call behaviour, read self and read structural feature.

Rule A.18: t_action_types

```
t_action_types(action: ActivityNode, ad: Activity, block: Block, regions:
    seq of InterruptibleActivityRegion): action =
   switch(action.Type)
      case CallOperation.Type: t_call_operation_action(action,ad,block)
      case Opaque.Type: t_opaque_action(action,ad,block)
      case AcceptEvent.Type:
         t_accept_event_action(action,ad,block,regions)
      case SendSignal.Type: t_send_signal_action(action,ad,block)
      case ValueSpecification.Type:
         t_value_specification_action(action,ad,block)
      case CallBehaviour.Type: t_call_behaviour_action(action,ad,block)
      case ReadSelf.Type: t_read_self_action(action,ad,block)
      case ReadStructuralFeature.Type:
         t_read_structural_feature_action(action,ad,block)
   end switch
```

A.1.8 Action Incoming Edges

An action only starts its execution if all data it needs and the control tokens are available. This is specified as an interleaving of control edges and data input as displayed in Rule A.19. This function is used as part of the function `t_action_node` (Rule A.17). Concerning control, there is an interleaving involving the `control` channel, each one synchronising on the incoming control edge index. After synchronisation, the action behaves as **`Skip`**. The interleaving of the input data is accomplished on input pin channels (starting with **`in`**). The input parameter names (which are preceded by a question mark) of these channels begin with **in** appended with the edge index. After receiving a value over a channel whose name starts with **in**, such value is assigned to the variable that has already been introduced by the function `t_action_node`. After both control and object inputs finishes, the CML action for the activity node continues.

Rule A.19: t_action_incoming_edges

```
t_action_incoming_edges(action: ActivityNode, ad: Activity): action =
   "( " sep "|||"  {"control."inEdge.index" -> Skip" | inEdge in seq
      action.IncomingEdges(Control.Type)}
   if action.input.size > 0 then
      " ||| " sep "|||"
         {"in_"name(ad)"_"name(action)"_"inObj.index"?in"name(inObj)" ->
         "name(inObj)  " := in"name(inObj) | inObj in seq action.input }
   end if
   ");"
```

A.1.9 Action Outgoing Edges

The function `t_action_outgoing_edges` (Rule A.20) defines how a SysML action that has not been interrupted passes the control to other activity nodes with data output. The resultant action from this function is used as part of function `t_action_node` (Rule A.17). The function `t_action_outgoing_edges` receives as arguments an action node and an activity. This function introduces an interleaving of control edges and another for dealing with output. First, it attempts to synchronize over the channel `control` with the outgoing edge indices as parameters. After synchronization, the CML action behaves as **Skip**. The output of values is accomplished by using output pin channels whose names start with `out`. This channel communicates the value of variable `name(outObj)`, the outgoing edge name.

Rule A.20: t_action_outgoing_edges

```
t_action_outgoing_edges(action: ActivityNode, ad: Activity): action =
   "( " sep "|||" {"control."outEdge.index" -> Skip "| outEdge in seq
      action.OutgoingEdges(Control.Type)}
   if action.output.size > 0 then
      " ||| " sep "|||"
         {"out_"name(ad)"_"name(action)"_"outObj.index"!"name(outObj)"
         -> Skip"  | outObj in seq action.output }
   end if
   ");"
```

A.1.10 Call Operation action

A Call Operation action is an operation call to a target (typically a block). The function `t_call_operation_action` (Rule A.21) receives as arguments an activity node (`action`) of an activity diagram (`ad`) of a block instance (`block`). Depending on the target block of the call, the operation can be an operation call to a block associated with the owner block (target is not null), or be an internal call to the block that owns the activity (the target is null). For instance, the activity **Rem** displayed in Figure 5.4 (page 104) has a call operation action called **rem**. As there is no input pin named **target**, which would be the target of the call, we assume that such an operation call is internal to the block that owns this activity. In case of having a target to the call, this action attempts to synchronize with a block operation over a channel with a name that begins with the name of the target block, given by `name(action.target.type)`, followed by an operation identifier. The channel expects an operation call identifier (`m_id`); it outputs the sender block name `$id` (the owner block) appended with the action name, and the receiver block name.

The last parameter is related to the operation record type as presented in Section 4.2 of Chapter 4. We use `mk_`, which is the value constructor operator of CML, to create a value according to the input record type of the operation (suffixed with `_I`). Notice that the data source is given by input pin names. After calling an operation of another block, we assign `end_guard` the value **`false`** and we communicate `inc`, indicating that the executing action cannot be interrupted. We wait the return from the operation call on a channel with a name that begins with the type of the target block name (`name(action.target.type)`). The channel event has an operation identifier, the sender block identifier, and the receiver block identifier. The sender and the receiver block identifiers are always the same. The input parameter `oper` receives a value of the output record type of the operation (`oper.$id` is the name of the operation and we check if it is the set of operations of the target block and if it has the name of the operation we are interested in). The output parameters and the return of the operation are assigned to output pins. Only after calling and returning from an operation, we establish that the executing action can be interrupted, and communicate `dec` and assign **`true`** to `end_guard`. Finally, the action behaves as **`Skip`**.

Similarly, when the operation call is internal, the target block identifier is replaced by the identifier of the owner block itself. The function `drop_two($id)` removes the two last names of the ID sequence because at this point the attribute `$id` is the identifier of the block appended with two names: the activity flag `mk_token("acts")` (included by Rule 4.13), and the name of the current activity `mk_token([name(ad)])` (included by Rule 5.3). At this point we are only interested in the identification of the block to send the internal call, therefore we remove these two names using function `drop_two($id)`

Rule A.21: t_call_operation_action

```
t_call_operation_action(action: ActivityNode, ad: Activity, block:
   Block):action =
    if (action.target != null) then
       name(action.target.type)"_op?m_id!
       ($id^[mk_token(\""name(action)"\")])!"name(action.target)"!
       mk_"action.operation.name"_I(mk_token(\""action.operation.name"_I\")"
       if size(action.input - action.target) > 0 then
          ","
       end if
       sep "," {name(inPin) | inPin in seq action.input and inPin !=
          action.target}
       ") -> inc."id(action)" -> end_guard := false;"
       name(action.target.type)"_op?m_id!
       ($id^[mk_token(\""name(action)"\")])!"name(action.target)"?oper:
       (oper.$id in set "name(action.target.type)"_O and
       oper.$id = mk_token(\""action.operation.name"_O\")) -> "
       if action.operation.return != null then
          "("   sep "|||" {"("name(outPin)" := oper.ret)" | outPin in seq
             action.output and
           ret in seq action.operation.OutputParameters}");"
       end if
       " dec."id(action)" -> (end_guard := true)"
    else
       name(block)"_op?m_id!($id^[mk_token(\""name(action)"\")])!
       (drop_two($id))!mk_"action.operation.name"_I(
             mk_token(\""action.operation.name"_I\")"
       if size(action.input - action.target) > 0 then
          ","
       end if
       sep "," {name(inPin) | inPin in seq action.input and inPin !=
          action.target}
       ") -> inc."id(action)" -> end_guard := false;"
       name(block)"_op?m_id!($id^[mk_token(\""name(action)"\")])!
       (drop_two($id))?oper: (oper.$id in set "name(block)"_O and
       oper.$id = mk_token(\""action.operation.name"_O\")) -> "
       if action.operation.return != null then
          "("   sep "|||" {"("name(outPin)" := oper.ret)" | outPin in seq
             action.output and
                         ret in seq action.operation.OutputParameters}");"
       end if
       " dec."id(action)" -> (end_guard := true)"
    end if
```

The next extract shows the application of Rules A.17- A.21 for the Call Operation action of the **Rem** activity (Figure 5.4 on page 104). We assume that the id of the action is `rem1`.

```
...
process ad_internal_Rem = val $id: ID @ begin
```

```
actions
...
Rem_1  = dcl end_guard: bool := true @ ( mu X @ ((
dcl rem:Item @ (
(control.3 -> Skip);
(Buffer_op?m_id!($id^[mk_token("Rem")])!(drop_two($id))!
mk_rem_I(mk_token("rem_I")) -> inc.[mk_token("Rem")] ->
end_guard := false;
Buffer_op?mi_id!($id^[mk_token("Rem")])!(drop_two($id))?oper:
(oper.$id in set Buffer_O and oper.$id = mk_token("rem_O")) ->
(rem := oper.ret); dec.[mk_token("Rem")] -> (end_guard := true));
(out_Rem_rem_1!rem -> Skip));X)) /_\ end_guard & END_DIAGRAM)
...
end
...
```

In this extract, we have an example of an internal operation call because the target pin of the call operation action is not used. The only pin of the action is **rem**, which stores the output of the operation call. We establish that after data is available for calling an operation and the operation is called, it is not possible to interrupt the action `rem_1`. This is established by the assignment of **`false`** to `end_guard`. After the operation returns a value, the action `rem_1` can be interrupted. After this, the action attempts to output the result of the operation on the channel that begins with `out_`. For simplification, we assume that the result of operations are only accessible by output parameters, hence, `ret` is an output parameter of the operation called and its value is assigned to the corresponding output pin (**`rem`**).

A.1.11 Opaque action

The function `t_opaque_action` (Rule A.22) defines the translation for SysML opaque actions. The CML action for a SysML opaque action is defined by the translation of SysML action body (`action.body`), which is written using the CML action language. Therefore, we simply translate the action language code according to function `t_action`, which is shown in Rule A.5.

Rule A.22: t_opaque_action

```
t_opaque_action(action: ActivityNode, ad: Activity, block: Block):action =
   t_action(action.body,block, name(action), [])
```

The next extract shows the application of Rules A.17- A.22 for the Opaque action of the **Add** activity (Figure 5.3 on page 103). The translation of an assignment is defined in several steps. First, we define an auxiliary variable for the attributes used in the expression (in this case `b`), then we get the current value of the attribute (`_get_`) and assign it to the variable created, next we execute the assignment to the auxiliary variable, and finally, we update the attribute with its new value (`_set_`). We assume that the id of this action is `OpaqueAction`.

```
...
process ad_internal_Add = val $id: ID @ begin
actions
...
OPAQUEACTION_1  = dcl end_guard: bool := true @ ( mu X @ ((
dcl x: Item; @ (
(control.2 -> Skip ||| in_Add_OpaqueAction_1?inx -> x := inx);
(dcl b: seq of Item @
Buffer_get_b.([mk_token("OpaqueAction")]).(drop_two($id))?$b ->
b := $b; b := b ^ [x];
Buffer_set_b.([mk_token("OpaqueAction")]).(drop_two($id))!b -> Skip);
(control.4 -> Skip)); update.opaque1!-1 -> X)
) /_\ end_guard & END_DIAGRAM)
...
end
...
```

A.1.12 Send Signal action

By using the function `t_send_signal_action` (Rule A.23) we can translate an activity node (`action`) that is a SysML Send Signal action of an activity diagram (`ad`) from a block instance. The CML action is defined by a communication and then the action behaves as **`Skip`**. The communication occurs over a channel with a name that begins with the name of the target block, followed by the indication that we are dealing with a signal (`_sig`). We expect a signal call identifier as input and we output the sender block identifier `$id` and the target identifier. Finally, we construct the signal record type, passing any parameters needed.

Rule A.23: t_send_signal_action

```
t_send_signal_action(action: ActivityNode, ad: Activity, block:
    Block):action =
   name(action.target.Type)"_sig?m_id!($id^[mk_token(\""name(action)"\")])
      !"name(action.target)"!mk_"action.signal.name"(
         mk_token(\""action.signal.name"\")"
   if size(action.input - action.target) > 0 then
      ","
   end if
   sep "," {name(inPin) | inPin in seq action.input and inPin !=
      action.target}") -> Skip"
```

In Figure 5.6 (on page 5.6), there are three send signal actions. The following extract shows the CML generated from application of Rule A.23 to the **sendPolice** Send Signal action. As explained before, the diagram omits the existence of a target pin that defines to whom the signal is sent, hence, we assume that there is such a pin and it is named `target`. Note that as the sending a signal is asynchronous, we do not need to worry about avoiding the interruption of its behaviour, hence, the `end_guard` variable does not change.

```
...
process ad_internal_BroadcastCall = val $id: ID @ begin
actions
...
BroadcastCall_1  = dcl end_guard: bool := true @ ( mu X @ ((
dcl c1: EmergencyCall, target: ID @ (
(control.2 ||| in_BroadcastCall_sendPolice_1?inc1 -> (c1 := inc1) |||
in_BroadcastCall_sendPolice_2?intarget -> (target := intarget) );
(Police_sig?m_id!($id^[mk_token("BroadcastCall")])!target!
mk_sendPolice(mk_token("sendPolice"),c1) -> Skip);
(control.3 -> Skip)); update.opaque1!-1 -> X)
) /_\ end_guard & END_DIAGRAM)
...
end
...
```

A.1.13 Value Specification action

A Value Specification action is an action that evaluates a value specification (OMG, 2011). For instance, the activity shown in Figure 2.6 on page 29 has a Value Specification action that evaluates the expression **(++currentState) mod nDevices** and outputs its value in the **s** output pin. Function `t_value_specification_action` translates an activity node (`action`) of an activity diagram (`ad`) that is related to a block instance (`block`). The activity node, a value specification action, is defined as the evaluation of the value expression and its assignment to a variable with name given by `name(action.result)`, an output pin.

Rule A.24: t_value_specification_action

```
t_value_specification_action(action: ActivityNode, ad: Activity, block:
    Block):action =
   name(action.result)" := "action.value
```

A.1.14 Accept Event action

AcceptEventAction is an action that waits for the occurrence of an event meeting specified condition (OMG, 2011). The function `t_accept_event_action` (Rule A.25) receives the following as arguments: an activity node (action), which is an accept event action, an activity diagram (ad), a block instance (block), and a sequence of interruptible regions (regions). This function returns a CML action. An Accept Event action with no incoming edges located in an interruptible region only starts its execution when the interruptible region starts. If there are interruptible regions and an accept event actionhas no incoming edge, we introduce an external choice of the incoming edges to the interruptible regions. These edges have source outside the region. These edges can be related to a control flow or an object flow. These are defined by channels control or a channel with name beginning with obj_. After synchronisation, the action behaves as **Skip**. Only after termination of the external choice, the Accept Event action is enabled to treat an event of the block. This is established by checking if the corresponding event is available in the event pool (hasevent channel) and removing it from the pool (getevent). In case there is no event, the action recurses, otherwise, the parameters of the event are put in the output pin variables.

Rule A.25: t_accept_event_action

```
t_accept_event_action(action: ActivityNode, ad: Activity, block: Block,
    regions: seq of InterruptibleActivityRegion): action =
   if regions.size() > 0 and action.IncomingEdges.size == 0 then
      "("
      for region in seq regions sep "[]" do
         for edge in seq region.edges do
            if notContains(edge.source,region) then
               if edge.isControl() then
                  "control."edge.index" -> Skip"
               else
                  "obj_"name(ad)"_"edge.index"?x_"edge.index" -> Skip"
               end if
            end if
         end for
      end for
      ");"
   end if
   "mu X @ ("name(block)"_hasevent!($id^[mk_token(\""name(action)"\")])!
                  mk_token(\""action.trigger.event.name"\") ->"
      name(block)"_getevent!($id^[mk_token(\""name(action)"\")])?e -> ("
   "if (e = <NOEVENT>) then
      X
   else"
      if action.result != null then
         "("
         sep "|||" {"("name(outPin)" := e.#4."ret")" | outPin in seq
             action.output and ret in seq action.trigger.event.Parameters}
         ")"
      else
```

```
        "Skip"
      end if
  "))"
```

A.1.15 Call Behaviour action

A Call Behaviour action invokes a behaviour of the model (OMG, 2011). In our semantics, we only consider this behaviour to be described as another activity, hence, this action represents the invocation of another activity. For instance, see the Call Behaviour action **BroadcastCall** in the activity **TreatEmergencyCall** shown in Figure 5.5 on page 104. The function stated in Rule A.26 receives as arguments an activity node (`cba`), which is a call behaviour action, an activity (`ad`), and a block instance (`block`). The CML action synchronizes over a channel whose name begins with
`startActivity_CBA_`, appended with the diagram name. This channel synchronizes with an event of a specific block identifier (given by `$id`, which is the owner of the activity) and the node index under translation. This channel may communicate values from the input pins of the action. Then, we assign **`false`** to the variable `end_guard` and communicate the `inc` event, stating that action cannot be interrupted. Afterwards, the action expects an event to synchronize over the channel with a name that begins with `endActivity_CBA_`, appended with the diagram name. This channel synchronizes with an event of a specific block identifier (given by `$id`, which is the owner of the activity) and Call Behaviour action index (in the diagram). This may receive value from the output parameters of the activity invoked by the Call Behaviour Action. These values are assigned to the output pin variables. Finally, we assign **`true`** to variable `end_guard` and communicate `dec` event, stating that the action can be interrupted. Note that the `startActivity_CBA` and `endActivity_CBA` events are renamed by Rule A.12 in order to fire the execution of the activity referred by the Call Behaviour action.

Rule A.26: t_call_behaviour_action

```
t_call_behaviour_action(cba: ActivityNode, ad: Activity, block: Block):
    action =
    "startActivity_CBA_"name(ad)".$id."cba.index if cba.input.size > 0 then
                                                "!" sep "!" { name(obj) |
                                                    obj in seq cba.input }
                                    end if

    "-> inc."id(cba)" -> (end_guard := false);"
    "endActivity_CBA_"name(ad)".$id."cba.index if cba.output.size > 0 then
                                            "?" sep "?" { "x_"name(obj)
                                                | obj in seq cba.output }
                                  end if
    " -> "
    if cba.output.size > 0 then
        "( " sep "|||" {"("name(outPin)" := x_"name(outPin)")" | outPin in
            seq cba.output}"); "
    end if
    "dec."id(cba)" -> (end_guard := true)"
```

The following extract shows the CML action for the call behaviour action depicted in Figure 5.5 on page 104.

```
...
process ad_internal_TreatEmergencyCall = val $id: ID @ begin
actions
...
TreatEmergencyCall_2  = dcl end_guard: bool := true @ ( mu X @ ((
dcl c3: EmergencyCall, c4: EmergencyCall @ (
(in_TreatEmergencyCall_BroadcastCall_3?inc3 -> (c3 := inc3));
(startActiity_CBA_TreatEmergencyCall.$id.3!c3 ->
inc.([mk_token("BroadcastCall")]) -> (end_guard := false);
endActivity_CBA_TreatEmergencyCall.$id.3?x_c4 ->
(c4 := x_c4); dec.([mk_token("BroadcastCall")]) ->
(end_guard := true));
(out_TreatEmergencyCall_BroadcastCall_3!c4 -> Skip));X))
/_\ end_guard & END_DIAGRAM)
...
end
...
```

A.1.16 Read Self action

Read Self action is an action that outputs the host block of an action (OMG, 2011). The function for Read Self action stated in Rule A.27 is very simple as it only puts in the output pin variable the identifier of the block, and which is passed as parameter of the CML process.

Rule A.27: t_read_self_action

```
t_read_self_action(action: ActivityNode, ad: Activity, block: Block):
    action =
   name(action.result)" := (drop_two($id))"
```

A.1.17 Read Structural Feature action

Read Structural Feature action is a structural feature action that outputs the values of a structural feature (e.g., values and properties) of a block (OMG, 2011). For instance, the activity **ActShine** shown in Figure 8.11 on page 161 has three Read Structural Feature actions that output the values for properties **l1**, **l2** and **l3**, respectively. The function stated in Rule A.28 recover the value of the structural feature of a given block according to the structural feature desired. Once the communication returns such value, it is assigned to the variable of the output pin.

Rule A.28: t_read_structural_feature_action

```
t_read_structural_feature_action(action: ActivityNode, ad: Activity,
    block: Block): action =
   name(block)"_get_"action.structuralFeature.name".($id^"id(action)").
      (drop_two($id))?x -> "name(action.result)" := x"
```

The next extract details the CML action for one of the read structural feature actions shown in Figure 8.11 on page 161 for reading the property **l1** of the block **DwarfSignal**.

```
...
process ad_internal_ActShine = val $id: ID @ begin
actions
...
Readl1_1  = dcl end_guard: bool := true @ ( mu X @ ((
dcl lamp1: LampType @ (
(control.1);
(DwarfSignal_get_l1.($id^[mk_token("Readl1")]).drop_two($id)?x ->
lamp1 := x);
(out_ActShine_Readl1_1!lamp1 -> Skip));X)
) /_\ end_guard & END_DIAGRAM)
...
end
...
```

A.1.18 Control Nodes

The function `t_control_node` (Rule A.29) defines the translation of control nodes. The translation of each control node introduces a CML action with a name that begins with `CNode_` followed by the node index (`ctr.index`). Based on the type of the control node, we call the appropriate function. A control node can have one of the following types: initial, flow final, activity final, decision, merge, fork, and join. A control action is interrupted if it is inside an interruptible region and the interrupting edge accepts a token. The environment can choose one of the events over a channel with name `interrupted` appended with the diagram identifier, the block identifier, and an interruptible region index `i`. If an interrupting edge accepts a token, the environment synchronises over one of these channels, finishing the control action execution. Then, the action behaves as the control node action again. When a diagram finishes, all executing actions finish by an interruption on the action `END_DIAGRAM`.

Rule A.29: t_control_node

```
t_control_node(ctr: ActivityNode, ad: Activity, block: Block, regions:
    seq of InterruptibleActivityRegion): action =
   "CNode_"ctr.index" = (("
   switch(ctr.Type)
      case Initial.Type: t_initial_node(ctr,ad,block)
      case FlowFinal.Type: t_flow_final_node(ctr,ad,block)
      case ActivityFinal.Type: t_activity_final_node(ctr,ad,block)
      case Decision.Type: t_decision_node(ctr,ad,block)
      case Merge.Type: t_merge_node(ctr,ad,block)
      case Fork.Type: t_fork_node(ctr,ad,block)
      case Join.Type: t_join_node(ctr,ad,block)
   end switch
   ")"
   if regions.size() > 0 then
      "/_\ ([]i in set {"{intRegion.index | intRegion in seq regions}"} @
         (interrupted."id(ad)".$id.i -> CNode_"ctr.index"))"
   end if
   ") /_\ END_DIAGRAM"
```

A.1.19 Initial Node

The function `t_initial_node` (Rule A.30) defines how to translate initial nodes (`ctr`) of an activity diagram (`ad`) of a block instance (`block`). Before firing all outgoing edges, the number of active tokens is updated to the number of edges leaving the node (`ctr.OutgoingEdges.size`). The function introduces interleaved actions that communicate `control!x`, where `x` is the index of an outgoing edge of an initial node. The `wait` event blocks the node to avoid it firing the outgoing edges again. The node can only be released by the termination of the diagram.

Rule A.30: t_initial_node

```
t_initial_node(ctr: ActivityNode, ad: Activity, block: Block): action =
   "update."id(ctr)"!"ctr.OutgoingEdges.size" -> (||| x in set {" sep ","
      {x.index | x in seq ctr.OutgoingEdges}"} @ [{}] control!x ->
      Skip);wait -> Skip"
```

The next extract shows the application of Rules A.29 and A.30 to the InitialNode of the **Add** activity (Figure 5.3 on page 103). Since it only has one outgoing edge, the set that defines the interleaving in Rule A.30 only has one element, which is the index of the control edge (the number 1 in this case). We assume that the id of this node is `cnode1`.

```
...
process ad_internal_Add = val $id: ID @ begin
actions
...
CNode_1 = (update.([mk_token("cnode1")])!1 ->
(||| x in set {1} @ control!x -> Skip);
wait -> Skip) /_\ END_DIAGRAM
...
end
...
```

A.1.20 Flow Final Node

The function `t_flow_final_node` (Rule A.31) receives as argument a control node (`ctr`), an activity (`ad`), and a block instance defined by the parameter `block`. This function introduces an external choice of actions related to every incoming edge that arrives in the flow final node. In the case of a control flow, the channel `control` is used for synchronisation, next it decreases the number of active tokens and then it behaves as the action that models the control node introduced by the Rule A.29 (`CNode_` appended with the control node index). For object flows, a channel with name starting with `obj_`, which is appended with the diagram unique identifier given by `id(ad)` and the edge index, expects a value on parameter `_[edge.index]`. Next, it decreases the number of active tokens and then it behaves as the control node the same way as previously described.

Rule A.31: t_flow_final_node

```
t_flow_final_node(ctr: ActivityNode, ad: Activity, block: Block): action =
    for edge in seq ctr.IncomingEdges sep "[]" do
        if edge.Type == Control.Type then
            "control."edge.index" -> update."id(ctr)"!(-1) ->
                CNode_"ctr.index
        else
            "obj_"name(ad)"_"edge.index"?x_"edge.index" ->
                update."id(ctr)"!(-1) -> CNode_"ctr.index
        end if
    end for
```

A.1.21 Activity Final Node

The function `t_activity_final_node` (Rule A.32) receives as argument a control node (`ctr`), an activity (`ad`), and a block instance defined by the parameter `block`. This function follows the same idea of Rule A.31 for FlowFinal nodes; however, instead of decreasing the number of active tokens, it sets this number to zero through the communication of the `clear` event. This enables the termination of the diagram.

Rule A.32: t_activity_final_node

```
t_activity_final_node(ctr: ActivityNode, ad: Activity, block: Block):
    action =
   for edge in seq ctr.IncomingEdges sep "[]" do
      if edge.Type == Control.Type then
         "control."edge.index" -> clear."id(ctr)" -> wait -> Skip"
      else
         "obj_"name(ad)"_"edge.index"?x_"edge.index" -> clear."id(ctr)"
             -> wait -> Skip"
      end if
   end for
```

The **Add** activity has two activity final nodes (Figure 5.3 on page 103). The next extract shows the corresponding CML to these nodes.

```
...
process ad_internal_Add = val $id: ID @ begin
actions
...
CNode_3 = (control!3 -> clear.([mk_token("cnode3")]) -> wait ->
        Skip) /_\ END_DIAGRAM
CNode_4 = (control!4 -> clear.([mk_token("cnode4")]) -> wait ->
        Skip) /_\ END_DIAGRAM
...
end
...
```

A.1.22 Decision Node

The function `t_decision_node` (Rule A.33) translates a decision node. If the incoming edge is concerned with control, then the action expects synchronisation over the channel `control` with the control incoming edge index as parameter. Notice that this rule uses the auxiliary rule `t\_statecopy(block, ctr)` (Rule A.6) that creates a local copy of the block instance state. This is needed because the constraint of the Decision node may be defined in terms of attributes of the block, thus, we create a local copy of their current values. A non-deterministic if statement is introduced with the guards of the outgoing edges from the decision node. This is followed by an event over the channel `control` with the decision node outgoing edge index. In the case of an object flow, the same non-deterministic if statement is used, however, the functions declares a variable `$var` to store the content flowing through the edge. When a guard is satisfied, the object received via the incoming edge is communicated using the variable `$var` through the object flow channel (starting with `obj_`).

Rule A.33: t_decision_node

```
t_decision_node(ctr: ActivityNode, ad: Activity, block: Block): action =
   if ctr.IncomingEdge.Type == Control.Type then
      "control."ctr.IncomingEdge.index" -> "
      t_statecopy(block, name(ctr))
      "; if " for outEdge in ctr.OutgoingEdges sep "|" do
            if outEdge.guard == "else" then
               "not (" sep "or" {x.guard | x in ctr.OutgoingEdges and x
                  != outEdge}") -> control."outEdge.index" ->
                  CNode_"ctr.index
            else
               outEdge.guard" -> control."outEdge.index" ->
                  CNode_"ctr.index
            end if
            end for
      "end"
   else
      "dcl $var: "t_types(ctrIncomingEdge.source.Type)" @ "
      "(obj_"name(ad)"_"ctr.IncomingEdge.index"?x_"ctr.IncomingEdge.index
      " -> ($var := x_"ctr.IncomingEdge.index")"
      t_statecopy(block, ctr)
      "; if " for outEdge in ctr.OutgoingEdges sep "|" do
            if outEdge.guard == "else" then
               "not (" sep "or" {x.guard | x in ctr.OutgoingEdges and x
                  != outEdge}") ->
               obj_"name(ad)"_"outEdge.index"!x_"ctr.IncomingEdge.index)"
                  -> CNode_"ctr.index
            else
               outEdge.guard" -> obj_"name(ad)"_"outEdge.index"!$var ->
                  CNode_"ctr.index
            end if
            end for
      "end)"
   end if
   ")"
```

The next extract shows the application of Rules A.29 and A.33 to the DecisionNode of the **Add** activity diagram displayed in Figure 5.3.

```
...
process ad_internal_Add = val Buffer_id: ID @ begin
actions
...
CNode_2 = (control.1 -> dcl b: seq of Item @ (
Buffer.get_b.CNode2.Buffer_id?$b -> (b := $b);
if  #b >= 5 -> control.3 -> CNode_2
| #b < 5 -> control.2 -> CNode_2
end
)) /_\ END_DIAGRAM
...
end
...
```

A.1.23 Merge Node

A merge node is translated by the function `t_merge_node`, which is defined by Rule A.34. It receives as arguments an activity node (`ctr`), which is a merge node, an activity (`ad`), and block instance (`block`). The incoming edges of a merge node are either all made of object flows or all made of control flows. If the outgoing edge of a merge node is a control flow (`Control.Type`), then the merge node synchronises with one of the incoming control edges from the set of indexes given by `{x.index | x in ctr.IncomingEdges}`. Then the action communicates over the `control` channel the outgoing edge index. Finally, the action behaves again as the (merge) control node. In the case of an object flow, the action offers to the environment the possibility of communication over channels with name starting with `obj_` for each incoming edge. The whole channel name is defined by the concatenation of `obj_` with `id(ad)_`, and the incoming edge index (`inEdge.index`). The input parameter is called `x`. After communication of a value recorded in `x`, such value is given as output over a channel whose name starts with `obj_` appended with the with `id(ad)_`, and the outgoing edge index (`ctr.OutgoingEdge.index`). After the output, the action behaves again as the merge control node action just introduced.

Rule A.34: t_merge_node

```
t_merge_node(ctr: ActivityNode, ad: Activity, block: Block): action =
   if ctr.OutgoingEdge.Type == Control.Type then
      "control?i: (i in set {"{x.index | x in ctr.IncomingEdges}"})->
         control."ctr.OutgoingEdge.index" -> CNode_"ctr.index
   else
      for inEdge in seq ctr.IncomingEdges sep "[]" do
         "obj_"name(ad)"_"inEdge.index"?x ->
            obj_"name(ad)"_"ctr.OutgoingEdge.index"!x -> CNode_"ctr.index
      end for
```

```
    end if
```

A.1.24 Fork Node

A fork node has just one incoming edge that is an object flow or a control flow. The function `t_fork_node` (Rule A.35) introduces an action for a fork node. In the case of an incoming control flow, the channel `control` is used for synchronization with the channel that represents the incoming edge index. Next, we increase the number of active tokens using channel `update`, and then the control token is available for all target actions of the outgoing edges from the fork node. At the end, the CML action behaves like **Skip** . In the case of an object flow, the (fork) action synchronises with the object incoming edge through channel `obj_`, appended with the incoming edge index, receiving an object in the parameter `x`. After this, the actions behave like an interleaving to communicate the object `x` through each channel `obj_` (appended with the outgoing edge index `outEdge.index`) . After the object is consumed by each outgoing edge, the (fork) action starts again. The sequential composition guarantees that only after finishing the execution of the interleaving related to control or object flow, the whole action behaves again as itself .

Rule A.35: t_fork_node

```
t_fork_node(ctr: ActivityNode, ad: Activity, block: Block): action =
   if ctr.IncomingEdge.Type == Control.Type then
      "control."ctr.IncomingEdge.index" ->
         update."id(ctr)"!"ctr.OutgoingEdges.size-1" ->"
      "(|||i in set {"{x.index | x in ctr.OutgoingEdges}"} @ [{}]
         control!i -> Skip);CNode_"ctr.index
   else
      "obj_"name(ad)"_"ctr.IncomingEdge.index"?x ->
         update."id(ctr)"!"ctr.OutgoingEdges.size-1" -> ("
      for outEdge in seq ctr.OutgoingEdges sep "|||" do
         "obj_"name(ad)"_"outEdge.index"!x -> Skip"
      end for
      "); CNode_"ctr.index
   end if
```

A.1.25 Join Node

The function `t_join_node` (Rule A.36) receives as arguments an activity node (`ctr`), which is the join node, an activity diagram (`ad`), and a block instance (`block`). For each object incoming edge of a join node, we declare a variable with name beginning with `edge_` and appended with the edge index. The type of each variable is the same as the type of the edge source in SysML. For each incoming edge related to control flow, we introduce an action prefixed with the `control` channel synchronising on the edge index. Then the action behaves as **Skip**. If the incoming edge is related to an object flow, we introduce an action that begins with an event over a channel whose name begins with `obj_` appended with the incoming edge index and that expects a value on a parameter with name `x_` followed by the edge index. Then the object is assigned to one of the variables declared in the beginning of the action. We distinguish variables according to the edge index.

After all CML events related to (control and object) incoming edges, we check if there is any object flow into the join node; notice that the communication is sequentially composed with the remaining of the join action. If there is an object flow, we decrease the number of active tokens (channel `update`) and we use channel `obj_` to communicate an object. If there are only control incoming nodes to the join node, we decrease the number of active tokens (channel `update`) and we synchronise with the control outgoing edge index over the channel `control`. After this communication happens in both cases, the action behaves as itself. In the case of the outgoing edge being a control flow, then all incoming edges are control flow as well. This is translated to a simpler version of what was previously described, as the function synchronises all incoming flows in interleaving, decreases the number of active tokens and communicates the control outgoing edge event before reinitialising the action.

Rule A.36: t_join_node

```
t_join_node(ctr: ActivityNode, ad: Activity, block: Block): action =
   if ctr.OutgoingEdge.Type == Object.Type then
      "(dcl" for edge in seq ctr.IncomingEdge(Object.Type) sep "," do
            "edge_"edge.index": "t_types(edge.source)
            end for
      "@ (("
      let objIndex = -1 in
         for inEdge in seq ctr.IncomingEdges sep "|||" do
            if inEdge.Type == Control.Type then
               "control."inEdge.index" -> Skip"
            else
               objIndex = inEdge.index
               "obj_"name(ad)"_"inEdge.index"?x_"inEdge.index" ->
                  edge_"inEdge.index" := x_"inEdge.index";Skip"
            end if
         end for
         ");"
         if objIndex != -1 then
            "update."id(ctr)"!("1-ctr.IncomingEdges.size") ->
               obj_"name(ad)"_"ctr.OutgoingEdge.index"!edge_"objIndex"))
               -> CNode_"ctr.index
         else
            "update."id(ctr)"!("1-ctr.IncomingEdges.size") ->
```

```
                control!"ctr.OutgoingEdge.index")) -> CNode_"ctr.index
          end if
       end let
    else
       "( "
       for inEdge in seq ctr.IncomingEdges sep "|||" do
          "control."inEdge.index" -> Skip"
       end for
       ");update."id(ctr)"!("1-ctr.IncomingEdges.size") ->
          control!"ctr.OutgoingEdge.index" -> CNode_"ctr.index
    end if
```

A.1.26 Object Nodes

The function `t_object_node` (Rule A.37) translates an object node into CML. It receives an object node, an activity, and a block as arguments. The function simply calls the appropriate function according to the type of the object node, which can be an input pin, an output pin, a parameter or a data store. Each of the related functions are described next.

Rule A.37: t_object_node

```
t_object_node(obj: ActivityNode, ad: Activity, block: Block, regions: seq
    of InterruptibleActivityRegion): action =

    switch(obj)
       case InputPin:
          t_input_pin(obj, ad, block, regions)
       case OutputPin:
          t_output_pin(obj, ad, block, regions)
       case ActivityParameterNode:
          t_parameter_node(obj, ad, block, regions)
       case DataStore:
          t_data_store(obj,ad,block, regions)
    end switch
```

A.1.27 Input Pin

Rule A.38 defines the semantics for input pins. It declares a CML action `ObjNode_` suffixed by the object node index. Input pins may receive data from incoming edges and each of these data is appended to the `elements` variable. If there are data inside `elements`, we send them to the action to which the pin is connected. After providing a data to the action we remove it from the pin. The CML action is a recursion consisting of an external choice. The first choice deals with the flow arriving in the pin. It checks if the pin has an infinite upper bound because in this case it can infinitely add elements to the pin. Otherwise, it can only receive a data if the pin has not reached its limit. It receives a data through an object channel and add it to the `elements` variable. The second choice represents the transfer of data from the pin to the action. Hence, it can only be performed if `elements` is not empty. The input data channel (starting with **`in`**, which is used by SysML actions in Rule A.19) is communicated removing the head of `element`. Similarly to other elements already presented, this action can be interrupted if it is inside an interruptible region, and it is terminated by `END_DIAGRAM`.

Rule A.38: t_input_pin

```
t_input_pin(obj: ActivityNode, ad: Activity, block: Block, regions: seq
    of InterruptibleActivityRegion): action =
    "ObjNode_"obj.index" = (dcl elements: seq of "t_types(obj.Type)" @"
    "(mu X @ (("
    "( "for edge in seq obj.IncomingEdges sep "[]" do
          if obj.upperBound != '*' then
             "[len elements < "obj.upperBound"] & "
          end if
          "obj_"name(ad)"_"edge.index"?x_"edge.index" -> elements :=
             elements^[x_"edge.index"]; X"
       end for
    ")"
    "[]"
    "([len elements > 0] &
       in_"name(ad)"_"name(obj.OwnerNode)"_"obj.index"!(hd elements) ->
       elements := (tl elements); X)"
    ")"
    if regions.size() > 0 then
       "/_\ ([]i in set {"{intRegion.index | intRegion in seq regions}"} @
          interrupted."id(ad)".$id.i -> X)"
    end if
    "))) /_\ END_DIAGRAM"
```

A.1.28 Activity Parameter

Rule A.39 shows the function `t_parameter_node` that translates parameter nodes of activities. It has a value-result parameter, which is passed by Rule 5.8 (page 5.8). This parameter represents the data stored in the node. If the node has no outgoing edge and at least one incoming edge, then it is an output parameter node. This node receives some data from object flow edges (`obj` channels), which is assigned to the parameter value, so it can be used outside of the scope of this action in Rule 5.9 (page 5.9) as the output of the activity. Finally, the number of active tokens is decreased by one. On the other hand, if it is an input parameter node, it means that it has no incoming edges and at least one outgoing edge. For this type of parameter, the number of active tokens is increased by the number of object edges flowing out of it. Next, each object edge transmits the data inside the node, which is stored in the parameter. The `wait` event is used to block the node because, after sending the data through the object flow edges, this node cannot do anything else besides waiting for termination.

Rule A.39: t_parameter_node

```
t_parameter_node(obj: ActivityNode, ad: Activity, block: Block, regions:
    seq of InterruptibleActivityRegion): action =
    "ObjNode_"obj.index" = vres " name(obj) ": " t_types(obj) " @"
    "(mu X @ (("
          if obj.OutgoingEdges.size == 0 and obj.IncomingEdges > 0 then
             for edge in seq obj.IncomingEdges sep "[]" do
                "obj_"name(ad)"_"edge.index"?x_"obj.index" -> "name(obj)"
                   := x_"obj.index"; update."id(obj)"!(-1) -> X)"
             end for
          else if obj.OutgoingEdges.size > 0 and obj.IncomingEdges.size ==
             0 then
                "update."id(obj)"!"obj.OutgoingEdges.size" -> ("
                for edge in seq obj.OutgoingEdges sep "|||" do
                   "obj_"name(ad)"_"edge.index"!"name(obj)" -> wait ->
                      Skip))"
                end for
          end if
       if regions.size() > 0 then
       "/_\ ([]i in set {"{intRegion.index | intRegion in seq regions}"} @
           interrupted."id(ad)".$id.i -> X)"
    end if
    ")) /_\ END_DIAGRAM"
```

The next extract shows the application of object node rules (rules A.37-A.39) to the input pin of the OpaqueAction (`ObjNode_1`) and the input activity parameter node of the **Add** activity (`ObjNode_2`) (Figure 5.3 on page 103).

```
...
process ad_internal_Add = val $id: ID @ begin
actions
...
ObjNode_1 = (dcl elements: seq of Item @( mu X @ ((
([len elements < 1] & obj_Add_1?x_1 ->
elements := elements^[x_1]; X)
[]
([len elements > 0] &
in_Add_OpaqueAction_1!(hd elements) ->
elements := (tl elements); X))
))) /_\ END_DIAGRAM
```

```
        ObjNode_2 = vres item: Item @( mu X @ ((
        update.([mk_token("ObjNode2")])!1 -> (obj_Add_1!item ->
        wait -> Skip))
        )) /_\ END_DIAGRAM
        ...
        end
        ...
```

The CML action `ObjNode_1` corresponds to the input pin **x** of the Opaque action shown in the **Add** activity. As there is no explicit upper bound, we assume that it can only hold one element of type **Item** at a time. When it is empty, it synchronises on channel `obj_Add_1` to receive an object data and adds such data to the sequence `elements`. When there is one element in the sequence, it synchronises on channel `in_Add_OpaqueAction_1`, which passes the data to the action node. The CML action `ObjNode_2` is related to the input parameter node **item** of type **Item**. It receives as argument the input data received by channel `startActivity_` shown in Rule 5.8 (page 112). As an input parameter node, it increases the number of current tokens (`update`) according to the number of outgoing edges. Then, it synchronises on channel `obj_Add_1` communicating the data received as argument. Finally, it stays blocked on channel `wait` until the activity terminates.

A.1.29 Output Pin

Rule A.40 defines the semantics for output pins. Output pins receive some data from the action that they are connected, and then they add such data to the variable that keep the state of the pin (`elements`). While there are some data in `elements`, the output pin communicates each of them through one of the outgoing edges of the pin. Its CML representation is similar to input pins, however, it first receives data from the output data channel (starting with `out`, which is used by SysML actions in Rule A.20) to store some data in the pin. Once there is at least one data in `elements`, it can transmit it through the object edge that flows out of the pin (channel `obj_`) before removing it from the pin.

Rule A.40: t_output_pin

```
t_output_pin(obj: ActivityNode, ad: Activity, block: Block, regions: seq
   of InterruptibleActivityRegion): action =
   "ObjNode_"obj.index" = (dcl elements: seq of "t_types(obj.Type)" @ "
   "(mu X @ (("
   "("
   if obj.upperBound != '*' then
      "[len elements < "obj.upperBound"] &"
   end if
   "out_"name(ad)"_"name(obj.OwnerNode)"_"obj.index"?x_"obj.index" ->
      elements := elements^[x_"obj.index"]; X"
```

```
    ")"
    "[]"
    "("
    for edge in seq obj.OutgoingEdges sep "[]" do
       "([len elements > 0] &  obj_"name(ad)"_"edge.index"!(hd elements)
           -> elements := (tl elements); X)"
    end for
    ")"
    ")"
    if regions.size() > 0 then
       "/_\ ([]i in set {"{intRegion.index | intRegion in seq regions}"} @
           interrupted."id(ad)".$id.i -> X)"
    end if
    "))) /_\ END_DIAGRAM"
```

A.1.30 Data Store

The function `t_data_store` (Rule A.41) shows how a data store node is translated. It has three types of behaviour. It may receive some data from incoming edges and put it in the head of the sequence that keeps the state of the node. If there is some data already in the head of the sequence, it is replaced by the new data. The other two cases reflect the edges leaving the node. There are two situations in this case. If the node was empty and a data just arrived, the next communication leaving the node does not generate a new token. However, after this first data leaves the node, the next communications through the outgoing edges generate a new token, because the data store keeps sending, in fact, a copy of it, which represents a new active token in the diagram.

Rule A.41: t_data_store

```
t_data_store(obj: ActivityNode, ad: Activity, block: Block, regions: seq
    of InterruptibleActivityRegion): action =
    "ObjNode_"obj.index" = (dcl elements: seq of "t_types(obj.Type)" @"
    "(mu X @ (("
    "(" for edge in seq obj.IncomingEdges sep "[]" do
          "(obj_"name(ad)"_"edge.index"?x_"obj.index" -> elements :=
              [x_"obj.index"];X)"
       end for
    ")"
    "[]"
    "(" for edge in seq obj.OutgoingEdges sep "[]" do
       "([len elements = 1] & obj_"name(ad)"_"edge.index"!(hd elements) ->
           elements := elements^[(hd elements)]; X)"
       end for
    ")"
    "[]"
    "( "for edge in seq obj.OutgoingEdges sep "[]" do
       "([len elements = 2] & obj_"name(ad)"_"edge.index"!(hd elements) ->
          update."id(obj)"!1 -> X)"
       end for
    ")"
    ")"
    if regions.size() > 0 then
       "/_\ ([]i in set {"{intRegion.index | intRegion in seq regions}"} @
          interrupted."id(ad)".$id.i -> X)"
    end if
    "))) /_\ END_DIAGRAM"
```

A.1.31 Channel sets

The following rules depict how to define the channel sets used for communication between the different elements. Rule A.42 shows how to build a channel set according to the type of the activity action node.

Rule A.42: t_channels_action_node

```
t_channels_action_node(action: ActivityNode, ad: Activity, block: Block,
    regions: seq of InterruptibleActivityRegion): chanset =
    "{|"
    for edge in (seq ctr.IncomingEdges or seq ctr.OutgoingEdges) sep "," do
       if edge.type == Control.Type then
          "control."edge.index
       else
          "obj_"name(ad)"_"edge.index
       end if
    end for
    for inPin in seq action.input do
       ",in_"name(ad)"_"name(action)"_"inPin.index
    end for
    for outPin in seq action.output do
       ",out_"name(ad)"_"name(action)"_"outPin.index
    end for
    ",update."id(action)

    switch(action)
       case CallOperation.Type:
          "|} union {| "name(action.target.type)"_op.m.($id^[mk_token(\""
          name(action)"\")]).target.op | m: nat, target: ID,
          op: OPS @ op.$id in set ("name(action.target.type)"_I union "
          name(action.target.type)"_O) and op.$id in set
          {mk_token(\""action.operation.name"_I\"),
        mk_token(\""action.operation.name"_O\")} |} union
        {| inc."id(action)",dec."id(action)",wait"
      case AcceptEvent.Type:
        t_accept_event_action(action,ad,block,regions)
        if regions.size() > 0 and action.IncomingEdges.size == 0 then
          for region in seq regions do
            for edge in seq region.edges do
              if notContains(edge.source,region) then
                if edge.isControl() then
                  ",control."edge.index
                else
                  ",obj_"name(ad)"_"edge.index
                end if
              end if
            end for
          end for
          ", "name(block)"_hasevent.($id^[mk_token(
          \""name(action)"\")]).mk_token(\""action.trigger.event.name
          "\"), "name(block)"_getevent.($id^[mk_token(
          \""name(action)"\")]),wait"
        end if
      case SendSignal.Type: t_send_signal_action(action,ad,block)
        "|} union {| "name(action.target.Type)"_sig.m.($id^[mk_token(
        \""name(action)"\")]).target.signal | m:nat, target: ID,
        signal: S @ signal.$id in set "name(action.target.type)"_S and
        signal.$id = mk_token("\"action.signal.name\"") |} union {|wait"
      case CallBehaviour.Type: t_call_behaviour_action(action,ad,block)
        ",startActivity_CBA_"name(ad)".$id."action.index",
        endActivity_CBA_"name(ad)".$id."action.index",inc."id(action)",
        dec."id(action)",wait"
      case ReadStructuralFeature.Type:
        t_read_structural_feature_action(action,ad,block)
        ","name(block)"_get_"action.structuralFeature.name".
        ($id^"id(action)").(drop_two($id)), wait"
    end switch
    if regions.size() > 0 then
      for intRegion in seq regions do
        ",interrupted."id(ad)".$id."intRegion.index
      end for
    end if
    ", endDiagram."id(ad)" |}"
```

Rule A.43 shows how to build a channel set according to the type of the activity control node.

Rule A.43: t_channels_control_node

```
t_channels_control_node(ctr: ActivityNode, ad: Activity, block: Block,
    regions: seq of InterruptibleActivityRegion): chanset =
   "{|"
   switch(ctr)
      case InitialNode:
         if (ctr.OutgoingEdges.size > 0) then
            "wait,update."id(ctr)
            for edge in seq ctr.OutgoingEdges.size do
               ",control."edge.index
            end for

      case FlowFinalNode:
         "update."id(ctr)","
         for edge in seq ctr.IncomingEdges sep "," do
            if edge.Type == Control.Type then
               "control."edge.index
            else
               "obj_"name(ad)"_"edge.index
            end if
         end for
      case ActivityFinalNode:
         "wait,clear."id(ctr)","
         for edge in seq ctr.IncomingEdges sep "," do
            if edge.Type == Control.Type then
               "control."edge.index
            else
               "obj_"name(ad)"_"edge.index
            end if
         end for
      case DecisionNode:
         for edge in (seq ctr.IncomingEdges or seq ctr.OutgoingEdges) sep
             "," do
            if edge.type == Control.Type then
               "control."edge.index
            else
               "obj_"name(ad)"_"edge.index
            end if
         end for
         t_readstate_chanset_constraints(block,name(ctr),ctr.Constraints)
      case MergeNode:
         for edge in (seq ctr.IncomingEdges or seq ctr.OutgoingEdges) sep
             "," do
            if edge.type == Control.Type then
               "control."edge.index
            else
               "obj_"name(ad)"_"edge.index
            end if
         end for
      case ForkNode:
         "update."id(ctr)","
         for edge in (seq ctr.IncomingEdges or seq ctr.OutgoingEdges) sep
            "," do
            if edge.Type == Control.Type then
               "control."edge.index
            else
               "obj_"name(ad)"_"edge.index
            end if
         end for
      case JoinNode:
         "update."id(ctr)","
         for edge in (seq ctr.IncomingEdges or seq ctr.OutgoingEdges) sep
            "," do
            if edge.Type == Control.Type then
               "control."edge.index
            else
               "obj_"name(ad)"_"edge.index
            end if
         end for
   end switch
   if regions.size() > 0 then
      for intRegion in seq regions do
         ",interrupted."id(ad)".$id."intRegion.index
      end for
   end if
   ",endDiagram."id(ad)"|}"
```

Rule A.44 shows how to build a channel set according to the type of the activity object node.

Rule A.44: t_channels_object_node

```
t_channels_object_node(obj: ActivityNode, ad: Activity, block: Block,
    regions: seq of InterruptibleActivityRegion): chanset =
   "{|"
   switch(obj)
      case InputPin:
         if obj.IncomingEdges.size > 0 then
            for edge in seq obj.IncomingEdges sep "," do
               "obj_"name(ad)"_"edge.index
            end for
            ",in_"name(ad)"_"name(obj.OwnerNode)"_"obj.index
         end if
      case OutputPin:
         if obj.OutgoingEdges.size > 0 then
```

```
            for edge in seq obj.OutgoingEdges sep "," do
               "obj_"name(ad)"_"edge.index
            end for
            ",out_"name(ad)"_"name(obj.OwnerNode)"_"obj.index
         end if
      case ActivityParameterNode:
         if (obj.IncomingEdges.size > 0 or obj.OutgoingEdges.size > 0)
            then
            for edge in (seq obj.IncomingEdges or seq obj.OutgoingEdges)
               sep "," do
               "obj_"name(ad)"_"edge.index
            end for
            ",update."id(obj)
         end if
      case DataStore:
         if (obj.IncomingEdges.size > 0 or obj.OutgoingEdges.size > 0)
            then
            for edge in (seq obj.IncomingEdges or seq obj.OutgoingEdges)
               sep "," do
               "obj_"name(ad)"_"edge.index
            end for
            ",update."id(obj)
         end if
   end switch
   if regions.size() > 0 then
      for intRegion in seq regions do
         ",interrupted."id(ad)".$id."intRegion.index
      end for
   end if
   ",endDiagram."id(ad)"|}"
```

Rule A.45 shows how to build a channel set used by interruptible regions.

Rule A.45: t_chanset_int_regions

```
t_chanset_int_regions(ad: Activity, block: Block): action =
   "{|"
   for intRegion in seq ad.group and
       intRegion.isInterruptibleActivityRegion() sep "," do
      for edge in seq intRegion.interruptingEdge sep "," do
         if edge.isControl() then
            "control."edge.index
         else
            "obj_"name(ad)"_"edge.index
         end if
      end for
      ",interrupted."id(ad)".$id."intRegion.index
   end for
   ",endDiagram."id(ad)"|}"
```

A.2 Rules for Sequence diagram

Rule A.46 creates the CML processes for sequence diagrams. Firstly, it checks if the diagram refers to any other diagram (interactionUse elements) in order to translate the referred diagrams first. Otherwise, it generates the process for the sequence diagram by invoking the function `t_simple_sd`.

Rule A.46: t_create_sd_process

```
t_create_sd_process(sd: Interaction, generated: set of names): seq of
    program paragraph =
   for iu in seq sd.InteractionUse do
      if iu.Interaction.hasInteractionUse() then
         t_create_sd_process(iu.Interaction,generated)
      else
         if not member(iu.Interaction.name,generated)
            t_simple_sd(iu.Interaction)
            generated = generated union iu.Interaction.name
         end if
      end if
   end for
   if not member(sd.name,generated) then
      t_simple_sd(sd)
      generated = generated union sd.name
   end if
```

Rule A.47 defines the internal process and the main process of sequence diagram. The internal process (`sd_internal_`) defines the the flow of events of the sequence digram. The main process (`sd_`) composes the internal process in parallel with any other processes of sequence diagrams referred inside the current diagram.

Rule A.47: t_simple_sd

```
t_simple_sd(sd: Interaction): seq of program paragraph =
   "process sd_internal_"name(sd)" = val sd_id: ID,"
   sep "," {t_lifeline_name(lf)"_id: ID" | lf in seq sd.Lifelines}
   let paramList = {} in
      for m in seq sd.Messages do
         for p in seq m.Arguments do
            if p.isInteractionParameter() and not exists(p.name,
               paramList) then
```

```
                 ", " p.name": " t_types(p.type)
                 paramList = paramList union p.name
              end if
           end for
        end for
     end let
     " @ begin"
     t_sd_actions(sd)
     "end"

     "process sd_"name(sd)" = val sd_id: ID,"
     sep "," {t_lifeline_name(lf)"_id: ID" | lf in lf in seq sd.Lifelines}
     let paramList = {} in
        for m in seq sd.Messages do
           for p in seq m.Arguments do
              if p.isInteractionParameter() and not exists(p.name,
                 paramList) then
                 ", " p.name": " t_types(p.type)
                 paramList = paramList union p.name
              end if
           end for
        end for
     end let
     "@ sd_internal_"name(sd)"(sd_id,"
     sep "," {t_lifeline_name(lf)"_id" | lf in lf in seq sd.Lifelines}
     let paramList = {} in
        for m in seq sd.Messages do
           for p in seq m.Arguments do
              if p.isInteractionParameter() and not exists(p.name,
                 paramList) then
                 ", " p.name
                 paramList = paramList union p.name
              end if
           end for
        end for
     end let
     ")"
     t_sd_iu_parallel(sd)
```

Rule A.48 completes the parallelism defined for the main process of a sequence diagram shown in the end of Rule A.47. It defines the right-hand side of the parallelism where the processes of the sequence diagrams referred inside the main diagram are interleaved.

Rule A.48: t_sd_iu_parallel

```
t_sd_iu_parallel(sd: Interaction): seq of process paragraph =
   if sd.hasInteractionUse then
```

```
      "[|{|" sep "," { "startRef."iu.index", endRef"iu.index
                  | iu in seq sd.InteractionUse }
         sep ","
            {"gate_ev."t_idMessageEnd(x,sd)"."t_idMessageEnd(y,iu_sd) |
               iu in seq sd.InteractionUse, x in seq iu.ActualGates,
               y in iu.formalGates, iu_sd == iu.interaction, x.matches(y)
                  }
         |}||"
      "(" sep "|||" {
         "(sd_"name(iu_sd)"(sd_id^[mk_token(\""name(iu_sd)"\")],
         sep "," {t_lifeline_name(lf)"_id" | lf in seq sd.Lifelines,
         lf_iu in seq iu_sd.Lifelines, lf == lf_iu}
         ))
         [[ beginInteraction.sd_id^[mk_token(\""name(iu_sd)"\")] <-
               startRef."iu.index","
            "endInteraction.sd_id^[mk_token(\""name(iu_sd)"\")] <-
                  endRef."iu.index"]]"
                  | iu in seq sd.InteractionUse, iu_sd ==
                     iu.interaction}")"
   end if
```

Rule A.49 generates an identifier according to the type of the message end, which can be a gate or lifeline.

Rule A.49: t_idMessageEnd

```
t_idMessageEnd(me: MessageEnd,sd: Interaction): action =
   if me.Type == Gate.Type then
      "([mk_token(\""name(sd)"\"), mk_token(\""gate_"me.index"\")])"
   else
      t_lifeline_name(me.Lifeline)"_id"
```

Rule A.50 generates an name for the lifeline whether the block of the lifeline has an instance identifier or not.

Rule A.50: t_lifeline_name

```
t_lifeline_name(lf: Lifeline): action =
   if lf.represents.selector == null then
      name(lf.represents)
   else
      name(lf.represents.selector)"_"name(lf.represents)
   end if
```

Rule A.51 defines the action for the messages buffer, which simulates the environments where the messages are exchanged.

Rule A.51: t_messages_buffer

```
t_messages_buffer(sd: Interaction): action =
   for m in seq sd.Messages do
      if m is an Operation then
         m.name" = mu X @ (
             "name(m.receiver.represents)"_mOP.s."m.index"."
            t_idMessageEnd(m.sender,sd)"."t_idMessageEnd(m.receiver,sd)"
               -> "
            name(m.receiver.represents)"_mOP.r."m.index"."
            t_idMessageEnd(sender,sd)"."t_idMessageEnd(receiver,sd)  " ->
               X)"
      else
         m.name" = mu X @ (
             "name(m.receiver.represents)"_mSIG.s."m.index"."
            t_idMessageEnd(m.sender,sd)"."t_idMessageEnd(m.receiver,sd)"
               -> "
            name(m.receiver.represents)"_mOP.r."m.index"."
            t_idMessageEnd(sender,sd)"."t_idMessageEnd(receiver,sd)  " ->
               X)"

   end for

   "MessagesBuffer = (" sep " ||| " {m.name | m in seq sd.Messages}  ")
      /_\ endInteraction.sd_id -> Skip"
```

A.2.1 Combined fragments

Rule A.52 is invoked by Rule 6.3 (on page 124) when a combined fragment is translated. It simply calls the appropriate translation function according to the type of the combined fragment.

Rule A.52: t_combined_frament

```
t_combined_fragment(cf: CombinedFragment, lf: Lifeline): action =
   switch(cf.cfType)
      case PAR.Type: t_par_combined_fragment(cf, lf)
      case STRICT.Type: t_strict_combined_fragment(cf, lf)
      case ALT.Type: t_alt_combined_fragment(cf, lf)
      case OPT.Type: t_opt_combined_fragment(cf, lf)
      case BREAK.Type: t_break_combined_fragment(cf, lf)
      case LOOP.Type: t_loop_combined_fragment(cf, lf)
      case CRITICAL.Type: t_critical_combined_fragment(cf, lf)
   end switch
```

A.2.1.1 PAR

Rule A.53 defines the translation function for the parallelism operator. It is represented by the interleaving of the content of the operands. For example, Figure 2.8 (on page 31) depicts the parallel combined fragment with three operands. Each operand has two asynchronous messages exchanged between two blocks. The only lifeline present in more than one operand is the lifeline **bus**. Thus, its corresponding CML action has the interleaving of three flows, each one having two events related to the reception of messages (**transmitPack**) from lifelines **Dev1**, **Dev2** and **Dev3**. The other lifelines only participate in one operand each. Hence, they only have the two sequential sending message events.

Rule A.53: t_par_combined_fragment

```
t_par_combined_fragment(par: CombinedFragment, lf: Lifeline): action =
"(" sep "|||"
    {t_lf_interaction_fragments(op.InteractionFragmentsFromLifeline(lf),
    lf) | op in seq par.Operands}");join."par.index" -> Skip"
```

A.2.1.2 STRICT

The strict order operator requires that the operands should be executed in the specific top-down order depicted in the fragment (i.e., if event e1 is above event e2 in different lifelines, then e1 happens before e2). Rule A.54 defines the translation function for this operator. Its semantics is provided by adding an internal control channel (`strict`) between the operands of a lifeline action. This channel communicate two natural numbers. The first (`strict.index`) represents the identifier of the this combined fragment and the second controls the order of execution of the operands (`op.index`). All lifeline actions involved in this combined fragment synchronise on this channel, which guarantees that the operands are executed in the specified order.

Rule A.54: t_strict_combined_fragment

```
t_strict_combined_fragment(strict: CombinedFragment, lf: Lifeline):
    action =
   sep ";"
       {t_lf_interaction_fragments(op.InteractionFragmentsFromLifeline(lf),
       lf)";strict."strict.index"."op.index" -> Skip" | op in seq
       strict.Operands}
```

A.2.1.3 OPT

Rule A.55 shows the translation function for the option fragment. Likewise the ALT fragment (shown on page 130), it is also translated as if-then-else statement, where the operand is executed if the constraint is true, otherwise, nothing happens.

Rule A.55: t_opt_combined_fragment

```
t_opt_combined_fragment(opt: CombinedFragment, lf: Lifeline): action =
   t_statecopyConstraint(opt.Operand.Constraint, opt)
   "if "opt.Operand.InteractionConstraint.specification" then"
      t_lf_interaction_fragments(
         opt.Operand.InteractionFragmentsFromLifeline(lf), lf)
   "else
      Skip)"
```

A.2.1.4 BREAK

Rule A.56 shows the translation function for the BREAK combined fragment. This fragment determines that, if the guard is evaluated to true, then the operand is executed and all remaining messages outside the break are ignored and are never executed. This semantic is defined in terms of an interruption that will halt the flow after the break when the constraint is evaluated to true.

Rule A.56: t_break_combined_fragment

```
t_break_combined_fragment(brk: CombinedFragment, lf: Lifeline): action =
   t_statecopyConstraint(brk.Operand.Constraint, brk)
   "if "brk.Operand.InteractionConstraint.specification" then"
      t_lf_interaction_fragments(
         brk.Operand.InteractionFragmentsFromLifeline(lf), lf)
      "; break."brk.index" -> block -> Skip"
   "else
      Skip
   )"
```

A.2.1.5 CRITICAL

Rule A.57 shows the translation function for the critical combined fragment. It is translated by the addition of two control events, one representing the initialisation of the critical region (`beginCR`) and another representing the termination of the critical region(`endCR`). These events synchronise with an auxiliary action (`CRITICAL`), which is defined by Rule A.58 that controls all events of the sequence diagram. It simply interleaves all CML actions for critical regions and it is invoked by Rule 6.2 (page 122). A CML action for a critical region executes the `RUN` function (Rule A.11) for all events of the diagram minus the events of the critical region. When any the lifelines enter the critical regtion, this action is interrupted and it starts to behave as `RUN` function again, however, only communicating the events of the critical region. When the critical region finishes, then this action is interrupted again and it recurses.

Therefore, once the critical region is reached, this auxiliary action only synchronises the events inside the critical region, thus any other flow outside of the region halts. When the events of the critical region terminates, the auxiliary action returns to synchronise on all events of the diagram.

Rule A.57: t_critical_combined_fragment

```
t_critical_combined_fragment(crt: CombinedFragment, lf: Lifeline): action
    =
   "beginCR."crt.index" -> Skip;"
   t_lf_interaction_fragments(
      crt.Operand.InteractionFragmentsFromLifeline(lf), lf)
   ";endCR."crt.index" -> Skip"
```

Rule A.58: t_critical_actions

```
t_critical_actions(crts: seq of CombinedFragment, sd: Interaction):
    action =
   for crt in seq crts do
      "CR_"crt.index" = "RUN(ev(sd)\ev(crt))" /_\ beginCR."crt.index" ->
          "RUN(ev(crt)) "/_\ endCR."crt.index" -> CR_"crt.index
   end for
   "CRITICAL = " sep "|||" {"CR_"crt.index | ctr in seq ctrs}
```

A.2.1.6 LOOP

Finally, Rule A.59 defines the translation function for the loop operator. It simply invokes the correspondent loop CML action defined by Rule A.60, which is called inside the Rule 6.2 (page 122). It also depends on the number of parameters of the loop. In case of two parameters, then it has a minimum and a maximum number of iterations and a constraint, in case of one parameter, then it has a maximum number of iterations a constraint, otherwise it only has a constraint. The loop action is defined as a recursive CML action controlled by maximum and minimum number of iterations and the constraint of the loop, which is evaluated at the beginning of each iteration, similar to a while statement.

Rule A.59: t_loop_combined_fragment

```
t_loop_combined_fragment(loop: CombinedFragment, lf: Lifeline): action =
   switch (loop.numberOfParameters)
      case 2:
         lf.Name"_LOOP_"loop.index"(1,"loop.firstParameter",
             "loop.secondParameter")"
      case 1:
         lf.Name"_LOOP_"loop.index"(1,"loop.firstParameter")"
      case 0:
         lf.Name"_LOOP_"loop.index
   end switch
   for intf in seq loop.InteractionFragments do
      if intf.Type == CombinedFragmentType and intf.cfType == BREAK.Type
         then
         "/_\ interrupt."intf.index" -> Skip"
      end if
   end for
```

Rule A.60: t_loop_actions

```
t_loop_actions(loops: seq of CombinedFragment, lf: Lifeline): action =
   for loop in seq loops do
      switch (loop.numberOfParameters)
         case 2:
            lf.name"_LOOP_"loop.index" = dcl counter, min, max: nat @ ("
            if loop.hasInteractionConstraint() then
               t_statecopyConstraint(loop.InteractionConstraint, loop)
            end if
            "if (counter < min) or (counter >= min and counter <= max"
            if loop.InteractionConstraint != null then
               "and " loop.InteractionConstraint.specification") then"
            else
               ") then"
            end if
               t_lf_interaction_fragments(
                  loop.Operand.InteractionFragments, lf)
                  ";"lf.name"_LOOP_"loop.index"(counter+1, min, max))"
            "else
               Skip)"
         case 1:
            lf.name"_LOOP_"loop.index" = dcl counter, m: nat @ ("
            if loop.hasInteractionConstraint() then
               t_statecopyConstraint(loop.InteractionConstraint, loop)
            end if
            "if (counter <= m"
            if loop.InteractionConstraint != null then
               "and " loop.InteractionConstraint.specification") then"
            else
               ") then"
            end if
               t_lf_interaction_fragments(
                  loop.Operand.InteractionFragments, lf)
                  ";"lf.name"_LOOP_"loop.index"(counter+1, m))"
```

```
             "else
                Skip)"
          case 0:
             lf.name"_LOOP_"loop.index" = "
             if loop.InteractionConstraint != null then
                t_statecopyConstraint(loop.InteractionConstraint, loop)
                "if ("loop.InteractionConstraint.specification") then"
             end if
                   t_lf_interaction_fragments(
                      loop.Operand.InteractionFragments, lf)
                      ";"lf.name"_LOOP_"loop.index
                "else
                   Skip"
       end switch
       if loop.hasInteractionConstraint() then
          ")"
       end if
    end for
```

A.2.2 InteractionUse

A sequence diagram also can call another sequence diagram that is already defined. The interactionUse constructor allows the modularisation of sequence diagrams. Figure 2.9 (on page 32) details this situation where the diagram depicted refers to another sequence diagram that is illustrated in Figure 2.10 (on page 32). When this constructor is used, the sequence diagram CML process is the parallel composition of the internal representation of the sequence diagram and the process of the diagram that is referred (rules A.47 and A.48). These two processes synchronise on two events, one for starting the referenced diagram and another for communicating its termination as detailed by Rule A.61. However, these events are renamed to others that signalise the beginning and the termination of a sequence diagram, as shown in Rule A.48. This renaming together with the parallelism between the process of the sequence diagrams allow a sequence diagram to invoke another.

Rule A.61: t_interaction_use

```
t_interaction_use(iu: InteractionFragment): action =
   "startRef."iu.index" -> endRef."iu.index" -> Skip"
```

B

Pre-processing of the CML models

The CML models generated by the Artisan Studio tool need to be made finite in order to be animated or to be analysed by a model checker. Animation is performed using the Symphony tool (COLEMAN et al., 2012), which is a integrated environment for specifying and analysing CML specifications. These adjustments are mainly due to the impossibility of these techniques to handle types with infinite values, like sets and sequences. The necessary changes for animation are simple and partially automated. For model checking, in addition to the changes required for animating the model, some other adjustments to use the corresponding syntax of CSP, and other performance optimisations are needed. Translation to CSP is needed because the CML model checker was not completely operational by the time of writing, hence, we use the FDR tool (GIBSON-ROBINSON et al., 2014), which is a CSP refinement checker, to perform our analyses. In what follows, refinement checker and model checker are used interchangeably. Optimisations are used to reduce the size of state space to be traversed, and to make the analysis feasible, otherwise, for some cases, the amount of time and computational resources required would be impracticable.

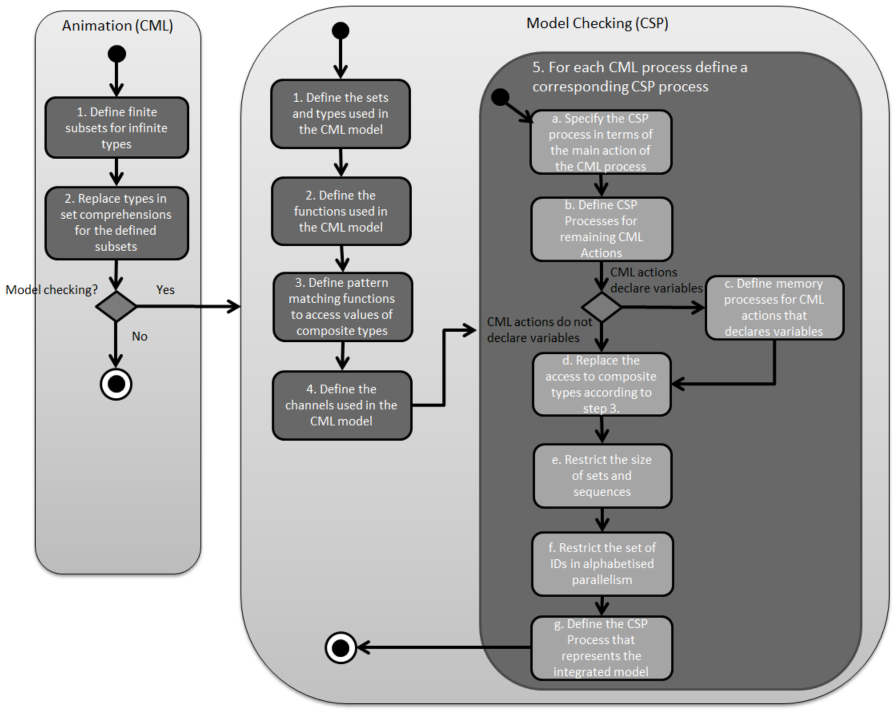
B.1 Procedures

Figure B.1 illustrates the needed steps to allow animation or model checking on the specification derived from the SysML model. We developed a tool to support the pre-processing steps for animation. The remaining steps related to model checking are not mechanised yet. The steps of Figure B.1 are detailed as follows.

B.1.1 Procedures for Animation

1. *Define finite subsets for the infinite types.* Both Animator and Model Checker cannot reason on types that have infinite values. Regarding animation, channel communication of values related to infinite types can be resolved because it waits for the user input and it only performs validation if the input provided is of that specific type.

Figure B.1: Procedures for pre-processing the CML model.


Animation (CML)
1. Define finite subsets for infinite types
2. Replace types in set comprehensions for the defined subsets
Model checking?
Yes
No
Model Checking (CSP)
1. Define the sets and types used in the CML model
2. Define the functions used in the CML model
3. Define pattern matching functions to access values of composite types
4. Define the channels used in the CML model
5. For each CML process define a corresponding CSP process
a. Specify the CSP process in terms of the main action of the CML process
b. Define CSP Processes for remaining CML Actions
CML actions declare variables
CML actions do not declare variables
c. Define memory processes for CML actions that declares variables
d. Replace the access to composite types according to step 3.
e. Restrict the size of sets and sequences
f. Restrict the set of IDs in alphabetised parallelism
g. Define the CSP Process that represents the integrated model

Source: Author's ownership.

However, if there are communications of sequences or sets restricted by comprehensions, the animator cannot evaluate the predicates of comprehensions of infinite values. For instance, the communication `sum?x`, where`sum` is a channel that waits for a sequence of natural numbers `x`, can be animated because the user must provide the sequence. However, suppose that this communication is restricted by a synchronous parallelism where the alphabet of synchronisation is defined by the following set comprehension `{|sum.x | x: seq of nat @ len x < 6|}`. In this case the animator cannot reason on all sequences of natural numbers to evaluate the constraint `len x < 6`. The finite subsets must be defined according to the following procedures:

1a	Define finite subsets for infinite user-defined types
1b	Define finite subsets for the internal types:
1(b)i	Define a finite subset for the type `ID`
1(b)ii	Define a finite subset for the type `OPS`
1(b)iii	Define a finite subset for the type `S`

(a) **Define finite subsets for infinite user-defined types**. In step 1 any user defined type with infinite values, including sets and sequences of types, must have its finite set of values defined. For instance, consider the following type definition,

```
types
LampId = <L1>|<L2>|<L3>

public light_I ::
$id: token
l: set of nat
```

the type `LampId` is already finite. However, the type `set of nat` is infinite. Therefore, a finite set for these type must be defined. For instance, the following set `{{1,2,3},{1,2},{2,3},{1,3},{1},{2},{3},{}}` is an example of a finite set for `set of nat`.

(b) **Define finite subsets for the internal types**.

i. **Define a finite subset for the type ID**. The type `ID` corresponds to each sequence of tokens used to identify diagrams and their constructs. Each one of the diagrams must be traversed and each ID used to identify any entity must be inserted in the subset of type `ID`. This subset must be defined as follows:

A. search the root process of the model

B. identify any value of `ID` used in this process and add it to the finite subset of type `ID`

C. traverse any process of the hierarchy and execute (b).

ii. **Define a finite subset for the type OPS**. This subset is required for animation because it is a composite type used to group operations. The procedure for the operations is the following:

A. Define a set for each record type of operation as a set comprehension where the field `$id` corresponds to a token to record the exact name of the operation record type and any attribute must have a finite type. For example,

```
public light_I ::
$id: token
l: set of nat
public light_O ::
$id: token
```

becomes the sets

```
fLight_I = {mk_light_I(mk_token(
"light_I"),att1)
| att1 in set fsetofnat}
```

```
fLight_O = {mk_light_O(mk_token(
"light_O"))}
```

where `fsetofnat` is a finite set provided by the user for a respective infinite type `set of nat`.

B. Define a finite set for all operations as the union of all finite sets previously defined for each operation record type.

```
fOPS = fLight_I union fLight_O ...
```

iii. **Define a finite subset for the type S**. Similarly to what is done to operations, this subset is required for animation because it is a composite type used to group signals. The same procedure used for operations in the previous step can be applied to the type `S`.

2. *Replace any occurrence of these types used in set comprehensions*. Now that the sets are defined, all occurrences of these types in set comprehensions must be replaced by the corresponding finite set. As explained earlier, as set comprehensions define restriction on possible values that can be used, when sets or sequences with infinite values are used in this constructs the Animator cannot reason on all possibilities. Considering the previous example `{|sum.x | x: seq of nat @ len x < 6|}`, we replace the infinite type `seq of nat` by its corresponding defined set `fseqofnat`, thus, the updated set comprehension is `{|sum.x | x in fseqofnat @ len x < 6|}`.

Both steps 1 and 2 are mechanised by a tool that was developed to automatically define the internal sets for types `ID`, `OPS` and `S` and replace any occurrence of these types in set comprehensions. Moreover, the sets for user-defined types can be informed by the user in order to make it replace their occurrences by finite sets in the CML model.

B.1.2 Procedures for Model Checking

0. *Do steps 1 and 2 as shown above*

1. *Define the sets and types used in the CML model*. In this step, the sets defined in the step 2 of the Animation must be defined in CSP as except for the sets of steps 1(b)ii and 1(b)iii regarding the `OPS` and `S` types. These types are defined as subtypes in CSP that use sets already defined. Other sets are defined as well in order to improve the performance of the model checking analysis. The procedure for the definition of these sets is detailed as follows:

1a	Define in CSP the same sets defined in the step 2 of the animation phase.
1b	Define the set `sysnat`
1c	Define the set `actint`
1d	Define the set `actnat`
1e	Define the set `sdnat`
1f	Define a CSP datatype `token`
1g	Define a subtype `OPS`
1h	Define a subtype `S`
1i	Define a datatype `MSG`
1j	Define the types `Bag` and `DL`.
1k	Define tuples for CML record types and datatypes for CML enumerations.
1l	Define the type `E`.
1m	Declare operation and signal sets of blocks.

(a) Define in CSP the same sets defined in the step 2 of the animation phase;

(b) Define the set `sysnat`. The `sysnat` subset is used in operation call and signal to uniquely identify concurrent events with the same source and target. In order to avoid that the response from one call to be returned to the other, this natural number identifies each one of these concurrent events. For instance, if two operations or signals with the same target instances are fired from two concurrent regions of a state machine, the only way to differentiate the calls is uniquely identifying them. Although such a scenario is considerably rare, the user must provide enough natural numbers to identify the maximum number of concurrent events. If such a scenario does not exist in the CML model only one natural number is enough. The standard procedure for the definition of this set is detailed below:

   i. for each state machine, search if any event (operation or signal) occurs concurrently to another event with same signature, same source and same target and count the number of concurrent events.

   ii. for each activity diagram, search if any event (operation or signal) occurs concurrently to another event with same signature, same source and same target and count the number of concurrent events.

   iii. define a set `sysnat` that must have a range of natural numbers from 1 to the highest number found in (i) and (ii).

(c) Define the set `actint`. This subset must provide a finite range of integer numbers to increment and decrement the number of active tokens inside an activity diagram as described previously in Chapter 5. This set must contain the numbers communicated by the `update` channel in the activity diagram process.

(d) Define the set `actnat`. This subset must provide a finite range of natural numbers to indexes, where each one is a control edge used in activity diagrams. This set must contain the numbers communicated by the `control` channel in the activity diagram process that has the maximum number of control edges.

(e) Define the set `sdnat`. This subset must provide natural numbers to identify each one of the messages in a sequence diagram. Therefore, the amount of natural numbers needed in this set is equal to the quantity of messages depicted in the sequence diagram. For example, if the sequence diagram has 8 messages, the subset of `sdnat` is `{1..8}`

(f) Define a CSP datatype `token` that has all names used for operations and signals. The values of this datatype are used in the definitions detailed in 1g and 1h below.
Example:
<u>CML</u>

```
public light_I ::
$id: token
l: set of LampId
public light_O ::
$id: token
public tick ::
$id: token
```

<u>CSP</u>

```
datatype token = light_I | light_O | tick
```

(g) Define a subtype `OPS`. For each operation record type, define a value in `OPS` following the standard `[oper_name](.[param_type])*`.
Example:
<u>CML</u>

```
public light_I ::
$id: token
l: set of LampId
public light_O ::
$id: token
```

CSP

```
subtype OPS = light_I.isetoflampid | light_O
```

(h) Define a subtype `S`. For each data type for signals define a value in `S` following the standard `[signal_name](.[parameter_type])*`. Example:

CML

```
public tick ::
$id: token
```

CSP

```
subtype S = tick
```

(i) Define a datatype `MSG`. This datatype is used to group all events related to operations and signals. It must have all values of `S` and `OPS`.

(j) Define the types `Bag` and `DL`. The type `Bag` is used to store the calls that arrive in a block. We use bags in our implementation because we need to differentiate the calls but their order is not relevant. The `DL` type is an extension to the boolean type in order to consider the `DEFER` value, which is used for events that are deferred by the state machine.

```
datatype DL = TRUE | FALSE | DEFER
nametype Bag = Seq(Token)
empty_bag = <>
```

(k) Define nametype tuples for CML record types and datatypes for CML enumerations. Any CML record type must be specified as nametype tuples and any enumeration types as datatypes in CSP.

(l) Define the type `E`. This type is used to represent events inside a block. It must be defined as a nametype tuple following the syntax `nametype E = (sysnat,ID,ID,MSG)`, where `sysnat` is the index of the event, the first `ID` is the sender, the second `ID` is the receiver and `MSG` is the event to be treated.

(m) Declare operation and signal sets of blocks. For each set of input operation, output operation and signal values of a block define its corresponding set in CSP using the same values declared in the CML model. These sets are used to verify that a certain operation or signal is in the block's interface. Example:

CML

```
Dwarf_I = {mk_token("light_I"),
mk_token("foo_I"),...}
Dwarf_O = {mk_token("light_O"),
mk_token("foo_O"),...}
Dwarf_S = {mk_token("tick"),...}
```

CSP

```
Dwarf_I = {light_I,foo_I...}
Dwarf_O = {light_O,foo_O...}
Dwarf_S = {tick, ...}
```

2. *Define the functions used in the CML model.* We define auxiliary functions to manipulate the data structures we use in our semantics. These functions also need to be defined in the CSP model.

   (a) Define the functions `prefix` and `drop_two`, which are used to manipulate the `ID` type, as detailed below.

```
prefix(x,y) = if #x > #y then false
else checkPrefix(x,y)

checkPrefix(x,y) = (if head(x) != head(y) then
  false
else if tail(x) == <>
then true
else checkPrefix(tail(x), tail(y)))

reverse(<>) = <>
reverse(<x>^s) = reverse(s)^<x>

drop_two(x) = reverse(tail(tail(reverse(x))))
```

   (b) Define the functions to manipulate the types Bag and DL as detailed below.

```
in_bag(t,<>) = false
in_bag(t,<x>^l) = if t == x then true else
                      in_bag(t,l)

bunion(b1,b2) = b1^b2
```

```
bdiff(b1,<>) = b1
bdiff(b1,<x>^l) = bdiff(bdiff_aux(b1,x),l)

bdiff_aux(<>,x) = <>
bdiff_aux(<y>^l,x) = if x == y then l
else <y>^bdiff_aux(l,x)

OR(a,b) = if (a == TRUE) or (b == TRUE) then
            TRUE
                  else FALSE
AND(a,b) = if (a == TRUE) and (b == TRUE) then
             TRUE
           else FALSE
DL_OR(a,b) = if (a == TRUE or a == FALSE) and
(b == TRUE or b == FALSE)
then OR(a,b)
else (
if (a == TRUE or b == TRUE)
then TRUE else DEFER
)
```

3. *Define pattern matching functions to access values of composite types.* Unlike CML, there is not a straightforward way in CSP of accessing fields of types following the syntax `nametype.field`. Hence, some pattern matching functions must be defined to access these fields for the types declared. The following procedure details what pattern matching functions must be created.

3a	Define functions for recovering the names of operations and signals.
3b	Define functions for recovering the parameters of operations and signals.
3c	Define functions for recovering the output types.
3d	Define functions for manipulating the values of the type `E`.

   (a) Define functions for recovering the names of operations and signals. For each value of OPS and S define functions `get_name` that returns the token representing the name of the message.
   Example:

```
get_name(light_I._) = light_I
get_name(light_O) = light_O
get_name(tick) = tick
```

(b) Define functions for recovering the parameters of operations and signals. For each value of OPS and S that has parameters, define functions of the type `get_paramX` that returns the Xth parameter of the type.
Example:

```
get_param1(light_I.u) = u
get_param2(foo_I._.u) = u
```

(c) Define functions for recovering the output types. For each value of OPS of type input (_I), define functions of the type `get_out` that return the corresponding output type.
Example:

```
get_out(light_I._) = light_O
get_out(foo_I._._) = foo_O
```

(d) Define functions for manipulating values of the type `E`. Define the following functions to access fields of a value of type `E`:

```
get_event((_,_,_,e))=e
get_n((n,_,_,_))=n
get_id1((_,id,_,_))=id
get_id2((_,_,id,_))=id
```

4. *Define the channels used in the CML model.* The channels of the CML model must be defined in CSP respecting the correspondence between the types previously defined. In addition, some channels must use the types defined in 1b-1e. The procedures for those types are:

   (a) Replace type `nat` by `sysnat` in the definition of the channels `X_addevent`, `X_op` and `X_sig`, where `X` is the name of a block.

   (b) Replace type `int` by `actint` in the definition of the channel `update`, which is used in activity diagram processes.

   (c) Replace type `nat` by `actnat` in the definition of the channel `control`, which is used in activity diagram processes.

   (d) Replace type `nat` by `sdnat` in the definition of the channels `X_mOP` and `X_mSIG`, which is used in sequence diagram processes, where `X` is the name of a block.

5. *For each CML process define a corresponding CSP process.* For each block, state machine, activities and sequence diagram, translate their related processes according to the following guidelines:

   (a) *Specify the CSP process in terms of the main action of the CML process.* The specification of the CSP process must be equal to the definition of the main CML action. When the CML process is not defined in terms of actions, then the CSP process must reflect the same specification of the CML process. Regarding the parameters and state variables of the CML process, if they are not modified along the process, they can be defined as parameters of the corresponding CSP process. In case any state variable or parameter of the process is modified along the CML process, the auxiliary memory introduced in 5c must be used.
   Example:
   CML

   ```
   process simple_Dwarf = $id: ID @ begin
   state
   l1: LIGHT := ON
   l2: LIGHT := ON
   l3: LIGHT := OFF
   ps: set of LampId := {}
   Dwarf_enabled: Bag := empty_bag
   actions
   ...
   @
   Dwarf_state
   [||{l1,l2,l3,ps}|{Dwarf_enabled}||]
   Dwarf_requests
   ```

   CSP

   ```
   simple_Dwarf(p_id) =
   Dwarf_state(p_id,ON,ON,OFF,{})
   |||
   Dwarf_requests(p_id,empty_bag)
   ```

   (b) *Define CSP processes for the remaining CML actions.* Each CML action of a CML process must be translated into a CSP process. The processes of these actions may be used by the process defined in the previous step or by the processes of other actions. If a parameter of the CML process is used inside the action, the CSP process must have it declared as a parameter of itself. The parameter should be passed where the action is invoked. For instance, see the parameter `$id` in CML that corresponds to the parameter `p_id` in CSP for the extract below. Moreover, some state variables are only used inside a specific action, hence, instead of defining it as a parameter of the process, it can be defined as a parameter of that specific action. See the `Dwarf_enabled` parameter in the example below.

CML

```
process simple_Dwarf = $id: ID @ begin...
state
Dwarf_enabled: Bag := empty_bag
actions
Dwarf_requests = mu X @ (
Dwarf_op?n?o!$id?x:(in_bag(x.$id,Dwarf_enabled))
->( ...
@
Dwarf_requests ...
```

CSP

```
Dwarf_requests(p_id,Dwarf_enabled) =
Dwarf_op?n?o!p_id?x:{x|x<-O,in_bag(get_id(x),
Dwarf_enabled)} -> (
...

simple_Dwarf(p_id) = Dwarf_requests(p_id,
                        empty_bag) ...
```

(c) *Define memory processes for CML actions that declare variables.* Some CML actions may have a declaration of state, which is not possible in CSP. To overcome this, we define a memory process that has the variables as parameters and provides channels to access the values of the parameters and to change their contents. This memory process must be composed in parallel with the respective processes that access and modify the declared variables.
Example:
CML

```
RSFA_2 = (...(dcl lamp: LIGHT @ (
(control.1 -> Skip); (DW_get_l1!id^<ad>!id?lamp1
-> (lamp := lamp1));
(out_ActShine_RSFA_5!lamp -> Skip)
/_\ end_guard & END_DIAGRAM
```

CSP

```
MEM_RSFA_2(lamp) = get_lamp!lamp ->
MEM_RSFA_2(lamp) [] set_lamp?x -> MEM_RSFA_2(x)

RSFA_2(id) =(... (control.1 -> Skip);
(DW_get_l1!id^<ad>!id?lamp1 ->
(lamp := lamp1));(out_ActShine_RSFA_5!lamp ->
Skip)...)
[|{| get_lamp,set_lamp |}|]
MEM_RSFA_2) /\ end_guard & END_DIAGRAM
```

(d) *Replace the access to composite types according to step 3.* As detailed earlier in step 3, the mechanisms to access values of composite data in CSP is different from the ones in CML. Hence, this step consists of updating the places where these accesses happen in the CML model for the syntax of CSP. Moreover, the native functions used to manipulate sets and sequences must be updated as well. Table B.1 describes some of the mappings between CSP and CML to manipulate sequences and sets. For more details see (SCATTERGOOD; ARMSTRONG, 2011).

Table B.1: Mapping between CML and CSP functions for manipulating sequences and sets.

CML	CSP	Description
x in set S	member(x,S)	Set membership
S union T	union(S,T)	Set union
S \T	diff(S,T)	Set difference
card S	card(S)	Set cardinality
[s]^[q]	<s>^<q>	Sequence concatenation
len s	#s or length(s)	length of a sequence
hd s	head(s)	head of a sequence
tl s	tail(s)	tail of a sequence

Source: Author's ownership.

Below we have an example of the updated CSP specification regarding the access to composite values and the mapping of a set membership function from CML to CSP.

Example:

<u>CML</u>

```
Dwarf_sig?n?o!b_id?x:(x.$id in set Dwarf_S)
```

<u>CSP</u>

```
Dwarf_sig?n?o!b_id?x:{x|x<-S,member(
get_name(x),Dwarf_S)}
```

(e) *Restrict the size of sets and sequences.* This is one of the optimisations that should be made in order to allow the model checker traverse a finite state space. Some of the processes have sequences and sets parameters that can increase infinitely. To avoid that, any sequence or set that can increase indefinitely must have their limits set in order to allow model checking. The way we describe how to reason about these limits does not follow a formal approach. However, in the future we aim to propose a strategy that uses data abstraction techniques (FARIAS et al., 2004; LAZIC; NOWAK, 2003). Below we describe some of treatments that should be applied in variables defined and used internally by our semantics.

- Restrict the size of the `enabled` sequence parameter of block's requests CSP process (`X_requests`, where `X` is the name of the block) according to the maximum size of concurrent requests that a block may receive. The `enabled` variable stores the operation calls that should be treated by the blocks. Once an operation call is treated, it is removed from this sequence. In order to define the maximum size of concurrent requests to operations of a block, the designer must investigate the scenarios where a block can concurrently receive operation calls and verify the scenario with the highest number of concurrent operation calls. This number must be the size limit of this sequence. If such a concurrent scenario does not exist, the size of the sequence must be limited to at most one element.
- Restrict the size of the set parameter `events` of the block's controller CSP process (`controller_X`, where `X` is the name of the block) according to the maximum size of the block's event pool. The `events` set is responsible to store events to be treated by the block's controller. Differently from the `enabled`, `events` stores values of the type `E`, which can be related to operation calls and signals.

- Restrict the size of the `deferred` sequence parameter of the block's controller CSP process (`controller_X`, where `X` is the name of the block) according to the maximum size of the deferred events pool of the block's state machine.

(f) *Restrict the set of IDs in alphabetised parallel compositions.* Replace the set of IDs used in channel sets of the alphabetised parallelism between processes in order to make them reflect the exact possible communications between blocks, state machines, activities and sequence diagrams.

(g) *Define a CSP process that represents an integrated model.* The system model that corresponds to blocks, state machines and activities must be defined in terms of the CSP process of the top-level block in the hierarchy of the system renaming all channels of operations and signals to their correspondent ones without the first `sysnat` value.
Example:

```
Model = Dwarf(<id_Dwarf>)[[Dwarf_op.n <-
Dwarf_OP, Dwarf_sig.n <- Dwarf_OP | n: sysnat]]
```

Following the steps described so far, we believe that model checking is possible for models whose sizes are similar to the ones described in Section 8.3.1 of Chapter 8 using the resultant CSP specification derived from the application of these procedures.

C

Semantic Validation Models

Here we present the CSP specifications used to validate our semantics as described in Section 8.4 of Chapter 8.

C.1 CSP model of the refinement for inserting a private operation

The first listing corresponds to the CSP specification for the private operation introduction refinement described in Section 8.4.1.

```
ID = {<bb>}
datatype token = bb
datatype DL = TRUE | FALSE | DEFER
OR(a,b) = if (a == TRUE) or (b == TRUE) then TRUE else FALSE
AND(a,b) = if (a == TRUE) and (b == TRUE) then TRUE else FALSE
DL_OR(a,b) = if (a == TRUE or a == FALSE) and (b == TRUE or b ==
    FALSE)
                then OR(a,b)
                else (
                    if (a == TRUE or b == TRUE)
                    then TRUE
                    else DEFER
                )
MyInt = {0,1,2}
MyNat = {0}
SetInt = Set(MyInt) --{{0},{1},{2},{0,1},{1,2},{0,2},{0,1,2},{}}--
    Set(MyInt)

datatype MSG = op1_I | op1_O | op2_I | op2_O | op_I | op_O |
    NOEVENT
```

```
nametype E = (MyNat,ID,ID,MSG)

datatype Token = oper1_I | oper1_O | oper2_I | oper2_O | oper_I |
    oper_O | noevent

get_id(op1_I) = oper1_I
get_id(op1_O) = oper1_O
get_id(op2_I) = oper2_I
get_id(op2_O) = oper2_O
get_id(op_I) = oper_I
get_id(op_O) = oper_O
get_id(NOEVENT) = noevent

get_out(op1_I) = oper1_O
get_out(op2_I) = oper2_O
get_out(op_I) = oper_O

get_event((_,_,_,e))=e
get_n((n,_,_,_))=n
get_id1((_,id,_,_))=id
get_id2((_,_,id,_))=id

subtype S = NOEVENT
subtype OPS = op1_I | op1_O | op2_I | op2_O | op_I | op_O
subtype I = op1_I | op2_I | op_I
subtype O = op1_O | op2_O | op_O
nametype Bag = Seq(Token)
empty_bag = <>
in_bag(t,<>) = false
in_bag(t,<x>^l) = if t == x then true else in_bag(t,l)
bunion(b1,b2) = b1^b2
bdiff(b1,<>) = b1
bdiff(b1,<x>^l) = bdiff(bdiff_aux(b1,x),l)
bdiff_aux(<>,x) = <>
bdiff_aux(<y>^l,x) = if x == y then l else <y>^bdiff_aux(l,x)
prefix(x,y) = if #x > #y then false
              else checkPrefix(x,y)

checkPrefix(x,y) = (if head(x) != head(y) then false
          else if tail(x) == <>
          then true
```

```
        else checkPrefix(tail(x), tail(y)))

reverse(<>) = <>
reverse(<x>^s) = reverse(s)^<x>

drop_two(x) = reverse(tail(tail(reverse(x))))

B_I = {oper1_I,oper2_I}
B_O = {oper1_O,oper2_O}

Bref_S = {}
Bref_I = {oper1_I,oper2_I,oper_I}
Bref_O = {oper1_O,oper2_O,oper_O}

channel B_op: MyNat.ID.ID.OPS
channel B_OP: ID.ID.OPS
channel B_addevent: MyNat.ID.ID.MSG
channel Bref_addevent: MyNat.ID.ID.MSG
B_state(id) = SKIP -- no state
Bref_state(id) = SKIP -- no state
B_requests(p_id,enabled) =
        B_op?n?o!p_id?x:{x|x<-I,member(get_id(x),B_I)} -> (
                B_addevent!n!o!p_id!x -> if #enabled > 2 then
                SKIP else B_requests(p_id,bunion(enabled, <get_out(
                   x)>))
        )
        []
        B_op?n?o!p_id?x:{x|x<-O,in_bag(get_id(x),enabled)} -> (
                B_requests(p_id,bdiff(enabled, <get_id(x)>))
        )
Bref_requests(p_id,enabled) =
        B_op?n?o!p_id?x:{x|x<-I,member(get_id(x),Bref_I)} -> (
                Bref_addevent!n!o!p_id!x -> if #enabled > 2 then
                   SKIP
                else Bref_requests(p_id,bunion(enabled, <get_out(x)
                   >))
        )
        []
        B_op?n?o!p_id?x:{x|x<-O,in_bag(get_id(x),enabled)} -> (
                Bref_requests(p_id,bdiff(enabled, <get_id(x)>))
        )
```

```

simple_B(p_id) = B_state(p_id) ||| B_requests(p_id,<>)

simple_Bref(p_id) = Bref_state(p_id) ||| Bref_requests(p_id,<>)

-- Controller and State Machine
--datatype NEWE = E | NOEVENT
channel B_inevent:ID.E
channel B_consumed: ID.DL
channel B_hasevent: ID.Token
channel B_getevent: ID.E


channel Bref_inevent:ID.E
channel Bref_consumed: ID.DL
channel Bref_hasevent: ID.Token
channel Bref_getevent: ID.E

remove(b,y) = (if (null(b)) then
           <>
             else if (head(b) == y) then
           tail(b)
             else
           <head(b)>^remove(tail(b),y)
           )



controller_B(id,events,deffered) = if card(events) > 1 or #deffered
     > 0 then
                              SKIP
                                 else


           ((B_addevent?n?o!id?e -> controller_B(id,union(events,{(n,o
                ,id,e)}),deffered) -- Event Arrival

                  []-- State Machine without Deferred Events
                    -- treatment because there is no event of this
                       type
           (card(events) > 0) & (
            |~| ev: events @ (
                B_inevent?o!ev-> (
                    B_consumed.o.TRUE -> controller_B(id,diff(
                        events,{ev}),deffered)
                    []
                    B_consumed.o.FALSE -> controller_B(id,diff(
                        events,{ev}),deffered)
                    []
                    B_consumed.o.DEFER -> controller_B(id,diff(
                        events,{ev}),deffered^<ev>)
                    )
                )
            )
        )

        )

controller_Bref(id,events,deffered) = if card(events) > 1 or #
    deffered > 0 then
                            SKIP
                                else


        ((Bref_addevent?n?o!id?e -> controller_Bref(id,union(events
            ,{(n,o,id,e)}),deffered) -- Event Arrival

                []-- State Machine
        (card(events) > 0) & (
            |~| ev: events @ (
                Bref_inevent?o!ev-> (
                    Bref_consumed.o.TRUE -> controller_Bref(id,diff
                        (events,{ev}),deffered)--Add
                        TreatDeferredEvents later
                    []
                    Bref_consumed.o.FALSE -> controller_Bref(id,
                        diff(events,{ev}),deffered)
                    []
                    Bref_consumed.o.DEFER -> controller_Bref(id,
                        diff(events,{ev}),deffered^<ev>)
                    )
                )
                )
            )

            )

bare_B(id) = (simple_B(id) [|{|B_addevent|}|] controller_B(id
    ,{},<>)) \ {|B_addevent|}

bare_Bref(id) = (simple_Bref(id) [|{|Bref_addevent|}|]
    controller_Bref(id,{},<>)) \ {|Bref_addevent|}




block_B(id) = (bare_B(id)) \ {|B_inevent, B_consumed|}

block_Bref(id) = (bare_Bref(id)) \  {| Bref_inevent, Bref_consumed,
     B_op.n.x.y.z | n: MyNat, x:ID, y:ID, z: {op_I, op_O}|}

channel loop

Loop = loop -> Loop

assert block_B(<bb>);Loop \ {|loop|} [T= block_Bref(<bb>);Loop \{|
    loop|}
```

C.2 CSP model of the refinement for inserting a private attribute

The second listing corresponds to the CSP specification for the private attribute introduction refinement described in Section 8.4.2.

```
ID = {<bb>}
datatype token = bb
datatype DL = TRUE | FALSE | DEFER
OR(a,b) = if (a == TRUE) or (b == TRUE) then TRUE else FALSE
AND(a,b) = if (a == TRUE) and (b == TRUE) then TRUE else FALSE
DL_OR(a,b) = if (a == TRUE or a == FALSE) and (b == TRUE or b ==
    FALSE)
                then OR(a,b)
                else (
                   if (a == TRUE or b == TRUE)
                   then TRUE
                   else DEFER
                )
MyInt = {0,1,2}
MyNat = {0}
SetInt = Set(MyInt) --{{0},{1},{2},{0,1},{1,2},{0,2},{0,1,2},{}}--
   Set(MyInt)

datatype MSG = op1_I | op1_O | op2_I | op2_O | NOEVENT
nametype E = (MyNat,ID,ID,MSG)
datatype Token = oper1_I | oper1_O | oper2_I | oper2_O | noevent

get_id(op1_I) = oper1_I
get_id(op1_O) = oper1_O
get_id(op2_I) = oper2_I
get_id(op2_O) = oper2_O
get_id(NOEVENT) = noevent
get_out(op1_I) = oper1_O
get_out(op2_I) = oper2_O
get_event((_,_,_,e))=e
get_n((n,_,_,_))=n
get_id1((_,id,_,_))=id
get_id2((_,_,id,_))=id

subtype S = NOEVENT
subtype OPS = op1_I | op1_O | op2_I | op2_O
subtype I = op1_I | op2_I
subtype O = op1_O | op2_O
nametype Bag = Seq(Token)
empty_bag = <>
in_bag(t,<>) = false
in_bag(t,<x>^l) = if t == x then true else in_bag(t,l)
bunion(b1,b2) = b1^b2
bdiff(b1,<>) = b1
bdiff(b1,<x>^l) = bdiff(bdiff_aux(b1,x),l)
bdiff_aux(<>,x) = <>
bdiff_aux(<y>^l,x) = if x == y then l else <y>^bdiff_aux(l,x)
prefix(x,y) = if #x > #y then false
              else checkPrefix(x,y)
```

```

checkPrefix(x,y) = (if head(x) != head(y) then false
        else if tail(x) == <>
        then true
        else checkPrefix(tail(x), tail(y)))

reverse(<>) = <>
reverse(<x>^s) = reverse(s)^<x>

drop_two(x) = reverse(tail(tail(reverse(x))))

B_I = {oper1_I,oper2_I}
B_O = {oper1_O,oper2_O}
Bref_S = {}
Bref_I = {oper1_I,oper2_I}
Bref_O = {oper1_O,oper2_O}

channel B_op: MyNat.ID.ID.OPS
channel B_OP: ID.ID.OPS
channel B_addevent: MyNat.ID.ID.MSG
channel Bref_addevent: MyNat.ID.ID.MSG
channel Bref_get_att: ID.ID.MyInt
channel Bref_set_att: ID.ID.MyInt

B_state(id) = SKIP
Bref_state(id,att) = Bref_get_att?o: {x | x <-ID, prefix(id,x)}!id!
    att -> Bref_state(id,att)
                            []
                        Bref_set_att?o: {x | x <-ID, prefix(id,x)}!id?
                            x -> Bref_state(id,x)

B_requests(p_id,enabled) =
        B_op?n?o!p_id?x:{x|x<-I,member(get_id(x),B_I)} -> (
                B_addevent!n!o!p_id!x -> if #enabled > 2 then SKIP
                    else B_requests(p_id,bunion(enabled, <get_out(x)
                    >))
        )
        []
        B_op?n?o!p_id?x:{x|x<-O,in_bag(get_id(x),enabled)} -> (
                B_requests(p_id,bdiff(enabled, <get_id(x)>))
        )
```

```

Bref_requests(p_id,enabled) =
        B_op?n?o!p_id?x:{x|x<-I,member(get_id(x),Bref_I)} -> (
                Bref_addevent!n!o!p_id!x -> if #enabled > 2 then
                   SKIP else Bref_requests(p_id,bunion(enabled, <
                   get_out(x)>))
        )
        []
        B_op?n?o!p_id?x:{x|x<-O,in_bag(get_id(x),enabled)} -> (
                Bref_requests(p_id,bdiff(enabled, <get_id(x)>))
        )

simple_B(p_id) = B_state(p_id) ||| B_requests(p_id,<>)

simple_Bref(p_id) = Bref_state(p_id,0) ||| Bref_requests(p_id,<>)


-- Controller and State Machine
channel B_inevent:ID.E
channel B_consumed: ID.DL
channel B_hasevent: ID.Token
channel B_getevent: ID.E


channel Bref_inevent:ID.E
channel Bref_consumed: ID.DL
channel Bref_hasevent: ID.Token
channel Bref_getevent: ID.E

remove(b,y) = (if (null(b)) then
        <>
          else if (head(b) == y) then
        tail(b)
          else
        <head(b)>^remove(tail(b),y)
        )

controller_B(id,events,deffered) =
    if card(events) > 1 or #deffered > 0 then
        SKIP
    else
```

```
          ((B_addevent?n?o!id?e ->
          controller_B(id,union(events,{(n,o,id,e)}),deffered) --
             Event Arrival

                    []-- State Machine without Deferred Events
                      -- treatment because there is no event of this
                         type
          (card(events) > 0) & (
               |~| ev: events @ (
                    B_inevent?o!ev-> (
                         B_consumed.o.TRUE ->
                         controller_B(id,diff(events,{ev}),deffered)
                         []
                         B_consumed.o.FALSE ->
                         controller_B(id,diff(events,{ev}),deffered)
                         []
                         B_consumed.o.DEFER ->
                         controller_B(id,diff(events,{ev}),deffered^<ev
                            >)
                         )
                    )
               )
          )

          )

controller_Bref(id,events,deffered) =
     if card(events) > 1 or #deffered > 0 then
          SKIP
     else
          ((Bref_addevent?n?o!id?e ->
          controller_Bref(id,union(events,{(n,o,id,e)}),deffered) --
             Event Arrival

                    []-- State Machine without Deferred Events
                      -- treatment because there is no event of this
                         type
          (card(events) > 0) & (
               |~| ev: events @ (
                    Bref_inevent?o!ev-> (
                         Bref_consumed.o.TRUE ->
```

```
                        controller_Bref(id, diff(events,{ev}),deffered)
                        []
                        Bref_consumed.o.FALSE ->
                        controller_Bref(id, diff(events,{ev}),deffered)
                        []
                        Bref_consumed.o.DEFER ->
                        controller_Bref(id, diff(events,{ev}),deffered^<
                            ev>)
                        )
                    )
                )
            )

            )

bare_B(id) = (simple_B(id) [|{|B_addevent|}|] controller_B(id
    ,{},<>)) \ {|B_addevent|}

bare_Bref(id) = (simple_Bref(id) [|{|Bref_addevent|}|]
    controller_Bref(id,{},<>)) \ {|Bref_addevent|}

block_B(id) = (bare_B(id)) \ {|B_inevent, B_consumed|}

block_Bref(id) = (bare_Bref(id)) \  {| Bref_inevent, Bref_consumed,
     Bref_get_att, Bref_set_att|}

channel loop

Loop = loop -> Loop

assert block_B(<bb>);Loop \ {|loop|} [T= block_Bref(<bb>);Loop \{|
    loop|}
```

C.3 CSP model of the refinement for the decomposition of a block

We show three listings one for each part of the decomposition of the block Management-Department described in Section 8.4.3.

The first listing corresponds to the CSP specification for the first part of the decomposition of block ManagementDepartment.

```
ID = {<bb>,<bb,stm>}


datatype token = bb | stm



datatype DL = TRUE | FALSE | DEFER

OR(a,b) = if (a == TRUE) or (b == TRUE) then TRUE else FALSE

AND(a,b) = if (a == TRUE) and (b == TRUE) then TRUE else FALSE

DL_OR(a,b) = if (a == TRUE or a == FALSE) and (b == TRUE or b ==
    FALSE)
            then OR(a,b)
            else (
                if (a == TRUE or b == TRUE)
                then TRUE
                else DEFER
            )

MyInt = {0,1,2}
MyNat = {0}

SetInt = Set(MyInt) --{{0},{1},{2},{0,1},{1,2},{0,2},{0,1,2},{}}--
    Set(MyInt)

datatype MSG = mc_I | mc_O | ml_I | ml_O | ma_I | ma_O | NOEVENT
nametype E = (MyNat,ID,ID,MSG)

datatype Token = manageClient_I | manageClient_O | manageLoan_I |
    manageLoan_O | manageAccount_I | manageAccount_O | noevent

get_id(mc_I) = manageClient_I
get_id(mc_O) = manageClient_O
get_id(ml_I) = manageLoan_I
get_id(ml_O) = manageLoan_O
get_id(ma_I) = manageAccount_I
get_id(ma_O) = manageAccount_O


get_id(NOEVENT) = noevent

get_out(mc_I) = manageClient_O
get_out(ml_I) = manageLoan_O
get_out(ma_I) = manageAccount_O



get_event((_,_,_,e))=e
get_n((n,_,_,_))=n
get_id1((_,id,_,_))=id
get_id2((_,_,id,_))=id

--subtype S = init
subtype S = NOEVENT

subtype OPS = mc_I | mc_O | ml_I | ml_O | ma_I | ma_O


subtype I = mc_I | ml_I | ma_I

subtype O = mc_O | ml_O | ma_O


nametype Bag = Seq(Token)
empty_bag = <>
in_bag(t,<>) = false
in_bag(t,<x>^l) = if t == x then true else in_bag(t,l)
bunion(b1,b2) = b1^b2
bdiff(b1,<>) = b1
bdiff(b1,<x>^l) = bdiff(bdiff_aux(b1,x),l)
bdiff_aux(<>,x) = <>
bdiff_aux(<y>^l,x) = if x == y then l else <y>^bdiff_aux(l,x)

prefix(x,y) = if #x > #y then false
              else checkPrefix(x,y)

checkPrefix(x,y) = (if head(x) != head(y) then false
          else if tail(x) == <> and tail(y) != <>
```

```
        then true
        else if tail(x) == <> and tail(y) == <>
        then false
        else checkPrefix(tail(x), tail(y)))

reverse(<>) = <>
reverse(<x>^s) = reverse(s)^<x>

drop_two(x) = reverse(tail(tail(reverse(x))))

ManagementDepartment_I = {manageClient_I,manageLoan_I,
    manageAccount_I}
ManagementDepartment_O = {manageClient_O,manageLoan_O,
    manageAccount_O}

ClientManager_I = {manageClient_I}
ClientManager_O = {manageClient_O}


channel ManagementDepartment_op: MyNat.ID.ID.OPS
channel ManagementDepartment_OP: ID.ID.OPS

channel ClientManager_op: MyNat.ID.ID.OPS
channel ClientManager_OP: ID.ID.OPS
channel ManagementDepartment_addevent: MyNat.ID.ID.MSG
channel ClientManager_addevent: MyNat.ID.ID.MSG


ManagementDepartment_state(id) = SKIP
ClientManager_state(id) = SKIP


ManagementDepartment_requests(p_id,enabled) =
        ManagementDepartment_op?n?o!p_id?x:{x|x<-I,member(get_id(x)
            ,ManagementDepartment_I)} -> (
                ManagementDepartment_addevent!n!o!p_id!x -> if #
                    enabled > 2 then SKIP else
                    ManagementDepartment_requests(p_id,bunion(
                    enabled, <get_out(x)>))
        )
        []
        ManagementDepartment_op?n?o!p_id?x:{x|x<-O,in_bag(get_id(x)
           ,enabled)} -> (
                ManagementDepartment_requests(p_id, bdiff(enabled, <
                   get_id(x)>))
        )


ClientManager_requests(p_id, enabled) =
        ClientManager_op?n?o!p_id?x:{x|x<-I,member(get_id(x),
           ClientManager_I)} -> (
                ClientManager_addevent!n!o!p_id!x -> if #enabled >
                   2 then
                SKIP else ClientManager_requests(p_id,bunion(
                   enabled, <get_out(x)>))
        )
        []
        ClientManager_op?n?o!p_id?x:{x|x<-O,in_bag(get_id(x),
           enabled)} -> (
                ClientManager_requests(p_id, bdiff(enabled, <get_id(
                   x)>))
        )


simple_ManagementDepartment(p_id) = ManagementDepartment_state(p_id
   ) ||| ManagementDepartment_requests(p_id,<>)

simple_ClientManager(p_id) = ClientManager_state(p_id) |||
   ClientManager_requests(p_id,<>)

-- Controller and State Machine
--datatype NEWE = E | NOEVENT
channel ManagementDepartment_inevent:ID.E
channel ManagementDepartment_consumed: ID.DL
channel ManagementDepartment_hasevent: ID.Token
channel ManagementDepartment_getevent: ID.E


channel ClientManager_inevent:ID.E
channel ClientManager_consumed: ID.DL
channel ClientManager_hasevent: ID.Token
channel ClientManager_getevent: ID.E
```

```

remove(b,y) = (if (null(b)) then
        <>
          else if (head(b) == y) then
        tail(b)
          else
        <head(b)>^remove(tail(b),y)
        )



controller_ManagementDepartment(id,events,deffered) = if card(
    events) > 1 or #deffered > 0 then
                            SKIP
                                else


        ((ManagementDepartment_addevent?n?o!id?e ->
            controller_ManagementDepartment(id,union(events,{(n,o,id
            ,e)}),deffered) -- Event Arrival

                []-- State Machine
        (card(events) > 0) & (
            |~| ev: events @ (
                ManagementDepartment_inevent?o!ev-> (
                    ManagementDepartment_consumed.o.TRUE ->
                        controller_ManagementDepartment(id, diff(
                        events,{ev}),deffered)--Add
                        TreatDeferredEvents later
                    []
                    ManagementDepartment_consumed.o.FALSE ->
                        controller_ManagementDepartment(id, diff(
                        events,{ev}),deffered)
                    []
                    ManagementDepartment_consumed.o.DEFER ->
                        controller_ManagementDepartment(id, diff(
                        events,{ev}),deffered^<ev>)
                    )
                )
            )
        )

        )

controller_ClientManager(id,events,deffered) = if card(events) > 1
    or #deffered > 0 then
                              SKIP
                                  else


        ((ClientManager_addevent?n?o!id?e ->
            controller_ClientManager(id,union(events,{(n,o,id,e)}),
            deffered) -- Event Arrival

                  []-- State Machine
        (card(events) > 0) & (
             |~| ev: events @ (
                  ClientManager_inevent?o!ev-> (
                       ClientManager_consumed.o.TRUE ->
                           controller_ClientManager(id,diff(events,{ev
                           }),deffered)--Add TreatDeferredEvents later
                       []
                       ClientManager_consumed.o.FALSE ->
                           controller_ClientManager(id,diff(events,{ev
                           }),deffered)
                       []
                       ClientManager_consumed.o.DEFER ->
                           controller_ClientManager(id,diff(events,{ev
                           }),deffered^<ev>)
                       )
                  )
             )
        )

        )

bare_ManagementDepartment(id) = (simple_ManagementDepartment(id)
    [|{|ManagementDepartment_addevent|}|]
    controller_ManagementDepartment(id,{},<>)) \ {|
    ManagementDepartment_addevent|}

bare_ClientManager(id) = (simple_ClientManager(id) [|{|
    ClientManager_addevent|}|] controller_ClientManager(id,{},<>)) \
      {|ClientManager_addevent|}

stm_ManagementDepartment(id) = State(id)

State(id) = ManagementDepartment_inevent!id^<stm>?e -> (
    (get_id(get_event(e)) == manageClient_I &
        ManagementDepartment_consumed!id^<stm>!TRUE ->
     ClientManager_op.get_n(e).get_id1(e).id!mc_I ->
         ClientManager_op.get_n(e).get_id1(e).id!mc_O ->
     ManagementDepartment_op!get_n(e)!get_id1(e)!id!mc_O -> State(
         id))
    []
    ((not(get_id(get_event(e)) == manageClient_I)) &
        ManagementDepartment_consumed!id^<stm>!FALSE -> State(id))
)


block_ManagementDepartment(id) = (bare_ManagementDepartment(id))
        \union({|ManagementDepartment_inevent,
        ManagementDepartment_consumed|},{|ManagementDepartment_op.n
            .x.y.z | n: MyNat, x: ID, y: ID,
        z: MSG, member(get_id(z),union(ManagementDepartment_I,
            ManagementDepartment_O)),
        prefix(id,x) or prefix(id,y)|})

block_ManagementDepartmentRef(id) = (bare_ManagementDepartment(id)
        [{|ManagementDepartment_inevent,
            ManagementDepartment_consumed,
        ManagementDepartment_op|} ||
        {|ManagementDepartment_inevent,
            ManagementDepartment_consumed,
        ClientManager_op, ManagementDepartment_op.n.x.id.w | n:
            MyNat, x: ID,
        w: {mc_O}|}]
        stm_ManagementDepartment(id))\union({|
            ManagementDepartment_inevent,
        ManagementDepartment_consumed|},{|ManagementDepartment_op.n
            .x.y.z | n: MyNat, x: ID, y: ID,
        z: MSG, member(get_id(z),union(ManagementDepartment_I,
            ManagementDepartment_O)),
        prefix(id,x) or prefix(id,y)|})

block_ClientManager(id) = (bare_ClientManager(id)) \  {|
    ClientManager_inevent, ClientManager_consumed|}

block_System(id) = block_ManagementDepartment(id)

block_SystemRef(id) =
        (block_ManagementDepartmentRef(id)

        [|{|ClientManager_op|}|]

        (block_ClientManager(id))
        )
        \{|ClientManager_op|}

channel loop

Loop = loop -> Loop


TEST = block_SystemRef(<bb>)


assert block_System(<bb>);Loop \ {|loop|} [T=
        (block_SystemRef(<bb>);Loop)
```

The second listing corresponds to the CSP specification for the second part of the decomposition of block ManagementDepartment.

```
ID = {<bb>,<bb,stm>}



datatype token = bb | stm



datatype DL = TRUE | FALSE | DEFER

OR(a,b) = if (a == TRUE) or (b == TRUE) then TRUE else FALSE

AND(a,b) = if (a == TRUE) and (b == TRUE) then TRUE else FALSE
```

```

DL_OR(a,b) = if (a == TRUE or a == FALSE) and (b == TRUE or b ==
    FALSE)
                 then OR(a,b)
                 else (
                     if (a == TRUE or b == TRUE)
                     then TRUE
                     else DEFER
                 )

MyInt = {0,1,2}
MyNat = {0}

SetInt = Set(MyInt) --{{0},{1},{2},{0,1},{1,2},{0,2},{0,1,2},{}}--
    Set(MyInt)

datatype MSG = mc_I | mc_O | ml_I | ml_O | ma_I | ma_O | NOEVENT
nametype E = (MyNat,ID,ID,MSG)

datatype Token = manageClient_I | manageClient_O | manageLoan_I |
    manageLoan_O | manageAccount_I | manageAccount_O | noevent

get_id(mc_I) = manageClient_I
get_id(mc_O) = manageClient_O
get_id(ml_I) = manageLoan_I
get_id(ml_O) = manageLoan_O
get_id(ma_I) = manageAccount_I
get_id(ma_O) = manageAccount_O


get_id(NOEVENT) = noevent

get_out(mc_I) = manageClient_O
get_out(ml_I) = manageLoan_O
get_out(ma_I) = manageAccount_O



get_event((_,_,_,e))=e
get_n((n,_,_,_))=n
get_id1((_,id,_,_))=id
```

```
get_id2((_,_,id,_))=id

--subtype S = init
subtype S = NOEVENT

subtype OPS = mc_I | mc_O | ml_I | ml_O | ma_I | ma_O


subtype I = mc_I | ml_I | ma_I

subtype O = mc_O | ml_O | ma_O


nametype Bag = Seq(Token)
empty_bag = <>
in_bag(t,<>) = false
in_bag(t,<x>^l) = if t == x then true else in_bag(t,l)
bunion(b1,b2) = b1^b2
bdiff(b1,<>) = b1
bdiff(b1,<x>^l) = bdiff(bdiff_aux(b1,x),l)
bdiff_aux(<>,x) = <>
bdiff_aux(<y>^l,x) = if x == y then l else <y>^bdiff_aux(l,x)

prefix(x,y) = if #x > #y then false
              else checkPrefix(x,y)

checkPrefix(x,y) = (if head(x) != head(y) then false
          else if tail(x) == <> and tail(y) != <>
          then true
          else if tail(x) == <> and tail(y) == <>
          then false
          else checkPrefix(tail(x), tail(y)))

reverse(<>) = <>
reverse(<x>^s) = reverse(s)^<x>

drop_two(x) = reverse(tail(tail(reverse(x))))

ManagementDepartment_I = {manageClient_I,manageLoan_I,
    manageAccount_I}
ManagementDepartment_O = {manageClient_O,manageLoan_O,
    manageAccount_O}

ClientManager_I = {manageClient_I}
ClientManager_O = {manageClient_O}

LoanManager_I = {manageLoan_I}
LoanManager_O = {manageLoan_O}



channel ManagementDepartment_op: MyNat.ID.ID.OPS
channel ManagementDepartment_OP: ID.ID.OPS

channel ClientManager_op: MyNat.ID.ID.OPS
channel ClientManager_OP: ID.ID.OPS

channel LoanManager_op: MyNat.ID.ID.OPS
channel LoanManager_OP: ID.ID.OPS

channel ManagementDepartment_addevent: MyNat.ID.ID.MSG
channel ClientManager_addevent: MyNat.ID.ID.MSG
channel LoanManager_addevent: MyNat.ID.ID.MSG
channel ManagementDepartmentRef_addevent: MyNat.ID.ID.MSG


ManagementDepartment_state(id) = SKIP
ClientManager_state(id) = SKIP
LoanManager_state(id) = SKIP
ManagementDepartmentRef_state(id) = SKIP

ManagementDepartment_requests(p_id, enabled) =
        ManagementDepartment_op?n?o!p_id?x:{x|x<-I, member(get_id(x)
            ,ManagementDepartment_I)} -> (
                ManagementDepartment_addevent!n!o!p_id!x -> if #
                    enabled > 2 then SKIP else
                    ManagementDepartment_requests(p_id, bunion(
                    enabled, <get_out(x)>))
        )
        []
        ManagementDepartment_op?n?o!p_id?x:{x|x<-O, in_bag(get_id(x)
            ,enabled)} -> (
                    ManagementDepartment_requests(p_id, bdiff(enabled, <
                        get_id(x)>))
          )

ManagementDepartmentRef_requests(p_id, enabled) =
          ManagementDepartment_op?n?o!p_id?x:{x|x<-I, member(get_id(x)
              ,ManagementDepartment_I)} -> (
                    ManagementDepartmentRef_addevent!n!o!p_id!x -> if #
                        enabled > 2 then SKIP else
                        ManagementDepartmentRef_requests(p_id, bunion(
                        enabled, <get_out(x)>))
          )
          []
          ManagementDepartment_op?n?o!p_id?x:{x|x<-O, in_bag(get_id(x)
              ,enabled)} -> (
                    ManagementDepartmentRef_requests(p_id, bdiff(enabled
                        , <get_id(x)>))
          )

ClientManager_requests(p_id, enabled) =
          ClientManager_op?n?o!p_id?x:{x|x<-I, member(get_id(x),
              ClientManager_I)} -> (
                    ClientManager_addevent!n!o!p_id!x ->
                    if #enabled > 2 then SKIP else
                    ClientManager_requests(p_id, bunion(enabled, <
                        get_out(x)>))
          )
          []
          ClientManager_op?n?o!p_id?x:{x|x<-O, in_bag(get_id(x),
              enabled)} -> (
                    ClientManager_requests(p_id, bdiff(enabled, <get_id(
                        x)>))
          )

LoanManager_requests(p_id, enabled) =
          LoanManager_op?n?o!p_id?x:{x|x<-I, member(get_id(x),
              LoanManager_I)} -> (
                    LoanManager_addevent!n!o!p_id!x ->
                    if #enabled > 2 then SKIP else
                    LoanManager_requests(p_id, bunion(enabled, <get_out(
                        x)>))
          )
          []
          LoanManager_op?n?o!p_id?x:{x|x<-O,in_bag(get_id(x),enabled)
             } -> (
                 LoanManager_requests(p_id,bdiff(enabled, <get_id(x)
                    >))
          )

simple_ManagementDepartment(p_id) = ManagementDepartment_state(p_id
    ) ||| ManagementDepartment_requests(p_id,<>)

simple_ClientManager(p_id) = ClientManager_state(p_id) |||
    ClientManager_requests(p_id,<>)

simple_LoanManager(p_id) = LoanManager_state(p_id) |||
    LoanManager_requests(p_id,<>)

simple_ManagementDepartmentRef(p_id) =
    ManagementDepartmentRef_state(p_id) |||
    ManagementDepartmentRef_requests(p_id,<>)


-- Controller and State Machine
--datatype NEWE = E | NOEVENT
channel ManagementDepartment_inevent:ID.E
channel ManagementDepartment_consumed: ID.DL
channel ManagementDepartment_hasevent: ID.Token
channel ManagementDepartment_getevent: ID.E


channel ClientManager_inevent:ID.E
channel ClientManager_consumed: ID.DL
channel ClientManager_hasevent: ID.Token
channel ClientManager_getevent: ID.E

channel LoanManager_inevent:ID.E
channel LoanManager_consumed: ID.DL
channel LoanManager_hasevent: ID.Token
channel LoanManager_getevent: ID.E

channel ManagementDepartmentRef_inevent:ID.E
channel ManagementDepartmentRef_consumed: ID.DL
channel ManagementDepartmentRef_hasevent: ID.Token
channel ManagementDepartmentRef_getevent: ID.E

remove(b,y) = (if (null(b)) then
          <>
            else if (head(b) == y) then
          tail(b)
            else
          <head(b)>^remove(tail(b),y)
          )



controller_ManagementDepartment(id,events,deffered) = if card(
    events) > 1 or #deffered > 0 then
                         SKIP
                            else


          ((ManagementDepartment_addevent?n?o!id?e ->
              controller_ManagementDepartment(id,union(events,{(n,o,id
              ,e)}),deffered) -- Event Arrival

                 []-- State Machine
          (card(events) > 0) & (
              |~| ev: events @ (
                 ManagementDepartment_inevent?o!ev-> (
                     ManagementDepartment_consumed.o.TRUE ->
                         controller_ManagementDepartment(id,diff(
                         events,{ev}),deffered)--Add
                         TreatDeferredEvents later
                     []
                     ManagementDepartment_consumed.o.FALSE ->
                         controller_ManagementDepartment(id,diff(
                         events,{ev}),deffered)
                     []
                     ManagementDepartment_consumed.o.DEFER ->
                         controller_ManagementDepartment(id,diff(
                         events,{ev}),deffered^<ev>)
                     )
```

```
                    )
                )
            )

            )

controller_ClientManager(id,events,deffered) = if card(events) > 1
    or #deffered > 0 then
                                SKIP
                                    else


            ((ClientManager_addevent?n?o!id?e ->
                controller_ClientManager(id,union(events,{(n,o,id,e)}),
                deffered) -- Event Arrival

                    []-- State Machine
            (card(events) > 0) & (
                |~| ev: events @ (
                    ClientManager_inevent?o!ev-> (
                        ClientManager_consumed.o.TRUE ->
                            controller_ClientManager(id,diff(events,{ev
                            }),deffered)--Add TreatDeferredEvents later
                        []
                        ClientManager_consumed.o.FALSE ->
                            controller_ClientManager(id,diff(events,{ev
                            }),deffered)
                        []
                        ClientManager_consumed.o.DEFER ->
                            controller_ClientManager(id,diff(events,{ev
                            }),deffered^<ev>)
                        )
                    )
                )
            )

            )

controller_LoanManager(id,events,deffered) = if card(events) > 1 or
     #deffered > 0 then
                                SKIP
                                else


            ((LoanManager_addevent?n?o!id?e -> controller_LoanManager(id,union(events,{(n,o,id,e)}),deffered) -- Event Arrival

                    []-- State Machine
            (card(events) > 0) & (
                |~| ev: events @ (
                    LoanManager_inevent?o!ev-> (
                        LoanManager_consumed.o.TRUE -> controller_LoanManager(id,diff(events,{ev}),deffered)--Add TreatDeferredEvents later
                        []
                        LoanManager_consumed.o.FALSE -> controller_LoanManager(id,diff(events,{ev}),deffered)
                        []
                        LoanManager_consumed.o.DEFER -> controller_LoanManager(id,diff(events,{ev}),deffered^<ev>)
                        )
                    )
                )
            )

            )


    controller_ManagementDepartmentRef(id,events,deffered) = if card(events) > 1 or #deffered > 0 then
                                SKIP
                                    else


            ((ManagementDepartmentRef_addevent?n?o!id?e -> controller_ManagementDepartmentRef(id,union(events,{(n,o,id,e)}),deffered) -- Event Arrival

                    []-- State Machine
            (card(events) > 0) & (
                |~| ev: events @ (
                    ManagementDepartmentRef_inevent?o!ev-> (
                        ManagementDepartmentRef_consumed.o.TRUE ->
                            controller_ManagementDepartmentRef(id, diff(
                            events,{ev}), deffered)--Add
                            TreatDeferredEvents later
                        []
                        ManagementDepartmentRef_consumed.o.FALSE ->
                            controller_ManagementDepartmentRef(id, diff(
                            events,{ev}), deffered)
                        []
                        ManagementDepartmentRef_consumed.o.DEFER ->
                            controller_ManagementDepartmentRef(id, diff(
                            events,{ev}), deffered^<ev>)
                        )
                    )
                )
            )

            )

bare_ManagementDepartment(id) = (simple_ManagementDepartment(id)
    [|{|ManagementDepartment_addevent|}|]
    controller_ManagementDepartment(id,{},<>)) \ {|
    ManagementDepartment_addevent|}

bare_ClientManager(id) = (simple_ClientManager(id) [|{|
    ClientManager_addevent|}|] controller_ClientManager(id,{},<>)) \
     {|ClientManager_addevent|}

bare_LoanManager(id) = (simple_LoanManager(id) [|{|
    LoanManager_addevent|}|] controller_LoanManager(id,{},<>)) \ {|
    LoanManager_addevent|}

bare_ManagementDepartmentRef(id) = (simple_ManagementDepartmentRef(
    id) [|{|ManagementDepartmentRef_addevent|}|]
    controller_ManagementDepartmentRef(id,{},<>)) \ {|
    ManagementDepartmentRef_addevent|}

stm_ManagementDepartment(id) = State1(id)

```

```
State1(id) = ManagementDepartment_inevent!id^<stm>?e -> (
    (get_id(get_event(e)) == manageClient_I &
        ManagementDepartment_consumed!id^<stm>!TRUE ->
     ClientManager_op.get_n(e).get_id1(e).id!mc_I ->
         ClientManager_op.get_n(e).get_id1(e).id!mc_O ->
     ManagementDepartment_op!get_n(e)!get_id1(e)!id!mc_O -> State1(
         id))
    []
    ((not(get_id(get_event(e)) == manageClient_I)) &
        ManagementDepartmentRef_consumed!id^<stm>!FALSE -> State1(id
        ))
)



stm_ManagementDepartmentRef(id) = State2(id)

State2(id) = ManagementDepartmentRef_inevent!id^<stm>?e -> (
    (get_id(get_event(e)) == manageClient_I &
        ManagementDepartmentRef_consumed!id^<stm>!TRUE ->
     ClientManager_op.get_n(e).get_id1(e).id!mc_I ->
         ClientManager_op.get_n(e).get_id1(e).id!mc_O ->
     ManagementDepartment_op!get_n(e)!get_id1(e)!id!mc_O -> State2(
         id))
    []
    (get_id(get_event(e)) == manageLoan_I &
        ManagementDepartmentRef_consumed!id^<stm>!TRUE ->
    LoanManager_op.get_n(e).get_id1(e).id!ml_I -> LoanManager_op.
        get_n(e).get_id1(e).id!ml_O ->
    ManagementDepartment_op!get_n(e)!get_id1(e)!id!ml_O -> State2(
        id))
    []
    ((not(get_id(get_event(e)) == manageClient_I) or
    not(get_id(get_event(e)) == manageLoan_I ) )
   & ManagementDepartmentRef_consumed!id^<stm>!FALSE -> State2(id)
        )
)


block_ManagementDepartment(id) = (bare_ManagementDepartment(id)
        [{|ManagementDepartment_inevent,
         ManagementDepartment_consumed,
         ManagementDepartment_op|}  ||
         {|ManagementDepartment_inevent, ManagementDepartment_consumed,
         ClientManager_op, ManagementDepartment_op.n.x.id.w | n: MyNat, x: ID,
         w: {mc_O}|}]
         stm_ManagementDepartment(id)
    )\union({|ManagementDepartment_inevent,
         ManagementDepartment_consumed|},{|ManagementDepartment_op.n.x.y.z | n: MyNat, x: ID, y: ID,
         z: MSG, member(get_id(z),union(ManagementDepartment_I, ManagementDepartment_O)),
         prefix(id,x) or prefix(id,y)|})

block_ManagementDepartmentRef(id) = (bare_ManagementDepartmentRef(id)
         [{|ManagementDepartmentRef_inevent, ManagementDepartmentRef_consumed,
         ManagementDepartment_op|}  ||
         {|ManagementDepartmentRef_inevent, ManagementDepartmentRef_consumed,
         ClientManager_op, LoanManager_op,
         ManagementDepartment_op.n.x.id.w | n:MyNat, x: ID,w: {mc_O, ml_O,ma_O}|}]
         stm_ManagementDepartmentRef(id)
    )\union({|ManagementDepartment_inevent,
         ManagementDepartment_consumed|},{|ManagementDepartment_op.n.x.y.z | n: MyNat, x: ID, y: ID,
         z: MSG, member(get_id(z),union(ManagementDepartment_I, ManagementDepartment_O)),
         prefix(id,x) or prefix(id,y)|})

block_ClientManager(id) = (bare_ClientManager(id)) \ {| ClientManager_inevent, ClientManager_consumed|}

block_LoanManager(id) = (bare_LoanManager(id)) \ {| LoanManager_inevent, LoanManager_consumed|}

block_System(id) = (block_ManagementDepartment(id)

         [{|ManagementDepartment_op,ClientManager_op|} ||
         {|ClientManager_op|}]

         (block_ClientManager(id) )
         )
         \{|ClientManager_op|}



block_SystemRef(id) =
         (block_ManagementDepartmentRef(id)

         [{|ManagementDepartment_op,
         ClientManager_op,LoanManager_op|} ||
         {|ClientManager_op,LoanManager_op|}]

         (block_ClientManager(id) [ {|ClientManager_op|}  ||
         {|LoanManager_op|} ] (
         block_LoanManager(id)))
         )
         \{|ClientManager_op,LoanManager_op|}

channel loop

Loop = loop -> Loop


TEST = block_SystemRef(<bb>)


assert block_System(<bb>);Loop \ {|loop|} [T=
         (block_SystemRef(<bb>);Loop)
```

Finally, the last listing corresponds to the third part of the decomposition of block ManagementDepartment.

```
ID = {<bb>,<bb,stm>}


datatype token = bb | stm



datatype DL = TRUE | FALSE | DEFER

OR(a,b) = if (a == TRUE) or (b == TRUE) then TRUE else FALSE

AND(a,b) = if (a == TRUE) and (b == TRUE) then TRUE else FALSE

DL_OR(a,b) = if (a == TRUE or a == FALSE) and (b == TRUE or b ==
    FALSE)
                then OR(a,b)
                else (
                    if (a == TRUE or b == TRUE)
                    then TRUE
                    else DEFER
                )

MyInt = {0,1,2}
MyNat = {0}

SetInt = Set(MyInt) --{{0},{1},{2},{0,1},{1,2},{0,2},{0,1,2},{}} --
    Set(MyInt)

datatype MSG = mc_I | mc_O | ml_I | ml_O | ma_I | ma_O | NOEVENT
nametype E = (MyNat,ID,ID,MSG)

datatype Token = manageClient_I | manageClient_O | manageLoan_I |
    manageLoan_O | manageAccount_I | manageAccount_O | noevent

get_id(mc_I) = manageClient_I
get_id(mc_O) = manageClient_O
get_id(ml_I) = manageLoan_I
get_id(ml_O) = manageLoan_O
get_id(ma_I) = manageAccount_I
get_id(ma_O) = manageAccount_O


get_id(NOEVENT) = noevent

get_out(mc_I) = manageClient_O
get_out(ml_I) = manageLoan_O
get_out(ma_I) = manageAccount_O
```

```



get_event((_,_,_,e))=e
get_n((n,_,_,_))=n
get_id1((_,id,_,_))=id
get_id2((_,_,id,_))=id

--subtype S = init
subtype S = NOEVENT

subtype OPS = mc_I | mc_O | ml_I | ml_O | ma_I | ma_O


subtype I = mc_I | ml_I | ma_I

subtype O = mc_O | ml_O | ma_O


nametype Bag = Seq(Token)
empty_bag = <>
in_bag(t,<>) = false
in_bag(t,<x>^l) = if t == x then true else in_bag(t,l)
bunion(b1,b2) = b1^b2
bdiff(b1,<>) = b1
bdiff(b1,<x>^l) = bdiff(bdiff_aux(b1,x),l)
bdiff_aux(<>,x) = <>
bdiff_aux(<y>^l,x) = if x == y then l else <y>^bdiff_aux(l,x)

prefix(x,y) = if #x > #y then false
              else checkPrefix(x,y)

checkPrefix(x,y) = (if head(x) != head(y) then false
         else if tail(x) == <> and tail(y) != <>
         then true
         else if tail(x) == <> and tail(y) == <>
         then false
         else checkPrefix(tail(x), tail(y)))

reverse(<>) = <>
reverse(<x>^s) = reverse(s)^<x>
```

```

drop_two(x) = reverse(tail(tail(reverse(x))))

ManagementDepartment_I = {manageClient_I ,manageLoan_I ,
    manageAccount_I}
ManagementDepartment_O = {manageClient_O ,manageLoan_O ,
    manageAccount_O}

ClientManager_I = {manageClient_I}
ClientManager_O = {manageClient_O}

LoanManager_I = {manageLoan_I}
LoanManager_O = {manageLoan_O}

AccountManager_I = {manageAccount_I}
AccountManager_O = {manageAccount_O}


channel ManagementDepartment_op: MyNat.ID.ID.OPS
channel ManagementDepartment_OP: ID.ID.OPS

channel ClientManager_op: MyNat.ID.ID.OPS
channel ClientManager_OP: ID.ID.OPS

channel LoanManager_op: MyNat.ID.ID.OPS
channel LoanManager_OP: ID.ID.OPS

channel AccountManager_op: MyNat.ID.ID.OPS
channel AccountManager_OP: ID.ID.OPS

channel ManagementDepartment_addevent: MyNat.ID.ID.MSG
channel ClientManager_addevent: MyNat.ID.ID.MSG
channel LoanManager_addevent: MyNat.ID.ID.MSG
channel AccountManager_addevent: MyNat.ID.ID.MSG
channel ManagementDepartmentRef_addevent: MyNat.ID.ID.MSG


ManagementDepartment_state(id) = SKIP
ClientManager_state(id) = SKIP
LoanManager_state(id) = SKIP
AccountManager_state(id) = SKIP
```

```
ManagementDepartmentRef_state(id) = SKIP

ManagementDepartment_requests(p_id,enabled) =
        ManagementDepartment_op?n?o!p_id?x:{x|x<-I,member(get_id(x)
            ,ManagementDepartment_I)} -> (
                ManagementDepartment_addevent!n!o!p_id!x -> if #
                    enabled > 2 then SKIP else
                    ManagementDepartment_requests(p_id,bunion(
                    enabled, <get_out(x)>))
        )
        []
        ManagementDepartment_op?n?o!p_id?x:{x|x<-O,in_bag(get_id(x)
            ,enabled)} -> (
                ManagementDepartment_requests(p_id,bdiff(enabled, <
                    get_id(x)>))
        )

ManagementDepartmentRef_requests(p_id,enabled) =
        ManagementDepartment_op?n?o!p_id?x:{x|x<-I,member(get_id(x)
            ,ManagementDepartment_I)} -> (
                ManagementDepartmentRef_addevent!n!o!p_id!x -> if #
                    enabled > 2 then SKIP else
                    ManagementDepartmentRef_requests(p_id,bunion(
                    enabled, <get_out(x)>))
        )
        []
        ManagementDepartment_op?n?o!p_id?x:{x|x<-O,in_bag(get_id(x)
            ,enabled)} -> (
                ManagementDepartmentRef_requests(p_id,bdiff(enabled
                    , <get_id(x)>))
        )

ClientManager_requests(p_id,enabled) =
        ClientManager_op?n?o!p_id?x:{x|x<-I,member(get_id(x),
            ClientManager_I)} -> (
                ClientManager_addevent!n!o!p_id!x ->
                if #enabled > 2 then SKIP else
                ClientManager_requests(p_id,bunion(enabled, <
                    get_out(x)>))
        )
        []
        ClientManager_op?n?o!p_id?x:{x|x<-O,in_bag(get_id(x),
            enabled)} -> (
                ClientManager_requests(p_id,bdiff(enabled, <get_id(
                    x)>))
        )

LoanManager_requests(p_id,enabled) =
        LoanManager_op?n?o!p_id?x:{x|x<-I,member(get_id(x),
            LoanManager_I)} -> (
                LoanManager_addevent!n!o!p_id!x ->
                if #enabled > 2 then SKIP else
                LoanManager_requests(p_id,bunion(enabled, <get_out(
                    x)>))
        )
        []
        LoanManager_op?n?o!p_id?x:{x|x<-O,in_bag(get_id(x),enabled)
            } -> (
                LoanManager_requests(p_id,bdiff(enabled, <get_id(x)
                    >))
        )

AccountManager_requests(p_id,enabled) =
        AccountManager_op?n?o!p_id?x:{x|x<-I,member(get_id(x),
            AccountManager_I)} -> (
                AccountManager_addevent!n!o!p_id!x ->
                if #enabled > 2 then SKIP else
                AccountManager_requests(p_id,bunion(enabled, <
                    get_out(x)>))
        )
        []
        AccountManager_op?n?o!p_id?x:{x|x<-O,in_bag(get_id(x),
            enabled)} -> (
                AccountManager_requests(p_id,bdiff(enabled, <get_id
                    (x)>))
        )

simple_ManagementDepartment(p_id) = ManagementDepartment_state(p_id
    ) ||| ManagementDepartment_requests(p_id,<>)

simple_ClientManager(p_id) = ClientManager_state(p_id) |||
    ClientManager_requests(p_id,<>)
```

```

simple_LoanManager(p_id) = LoanManager_state(p_id) |||
    LoanManager_requests(p_id,<>)

simple_AccountManager(p_id) = AccountManager_state(p_id) |||
    AccountManager_requests(p_id,<>)

simple_ManagementDepartmentRef(p_id) =
    ManagementDepartmentRef_state(p_id) |||
    ManagementDepartmentRef_requests(p_id,<>)


-- Controller and State Machine
--datatype NEWE = E | NOEVENT
channel ManagementDepartment_inevent:ID.E
channel ManagementDepartment_consumed: ID.DL
channel ManagementDepartment_hasevent: ID.Token
channel ManagementDepartment_getevent: ID.E


channel ClientManager_inevent:ID.E
channel ClientManager_consumed: ID.DL
channel ClientManager_hasevent: ID.Token
channel ClientManager_getevent: ID.E

channel LoanManager_inevent:ID.E
channel LoanManager_consumed: ID.DL
channel LoanManager_hasevent: ID.Token
channel LoanManager_getevent: ID.E

channel AccountManager_inevent:ID.E
channel AccountManager_consumed: ID.DL
channel AccountManager_hasevent: ID.Token
channel AccountManager_getevent: ID.E

channel ManagementDepartmentRef_inevent:ID.E
channel ManagementDepartmentRef_consumed: ID.DL
channel ManagementDepartmentRef_hasevent: ID.Token
channel ManagementDepartmentRef_getevent: ID.E

remove(b,y) = (if (null(b)) then
        <>
          else if (head(b) == y) then
        tail(b)
          else
        <head(b)>^remove(tail(b),y)
        )



controller_ManagementDepartment(id,events,deffered) = if card(
    events) > 1 or #deffered > 0 then
                                SKIP
                                  else


        ((ManagementDepartment_addevent?n?o!id?e ->
            controller_ManagementDepartment(id,union(events,{(n,o,id
            ,e)}),deffered) -- Event Arrival

                []-- State Machine
        (card(events) > 0) & (
            |~| ev: events @ (
                ManagementDepartment_inevent?o!ev-> (
                    ManagementDepartment_consumed.o.TRUE ->
                        controller_ManagementDepartment(id, diff(
                        events,{ev}),deffered)--Add
                        TreatDeferredEvents later
                    []
                    ManagementDepartment_consumed.o.FALSE ->
                        controller_ManagementDepartment(id, diff(
                        events,{ev}),deffered)
                    []
                    ManagementDepartment_consumed.o.DEFER ->
                        controller_ManagementDepartment(id, diff(
                        events,{ev}),deffered^<ev>)
                    )
                )
            )
        )

        )
```

```

controller_ClientManager(id,events,deffered) = if card(events) > 1
    or #deffered > 0 then
                              SKIP
                                  else


          ((ClientManager_addevent?n?o!id?e ->
              controller_ClientManager(id,union(events,{(n,o,id,e)}),
              deffered) -- Event Arrival

                    []-- State Machine
          (card(events) > 0) & (
              |~| ev: events @ (
                    ClientManager_inevent?o!ev-> (
                         ClientManager_consumed.o.TRUE ->
                             controller_ClientManager(id,diff(events,{ev
                             }),deffered)--Add TreatDeferredEvents later
                         []
                         ClientManager_consumed.o.FALSE ->
                             controller_ClientManager(id,diff(events,{ev
                             }),deffered)
                         []
                         ClientManager_consumed.o.DEFER ->
                             controller_ClientManager(id,diff(events,{ev
                             }),deffered^<ev>)
                         )
                    )
              )
          )

          )

controller_LoanManager(id,events,deffered) = if card(events) > 1 or
     #deffered > 0 then
                              SKIP
                                  else


          ((LoanManager_addevent?n?o!id?e -> controller_LoanManager(
              id,union(events,{(n,o,id,e)}),deffered) -- Event Arrival
```

```

                    []-- State Machine
          (card(events) > 0) & (
               |~| ev: events @ (
                    LoanManager_inevent?o!ev-> (
                         LoanManager_consumed.o.TRUE ->
                            controller_LoanManager(id, diff(events,{ev}),
                            deffered)--Add TreatDeferredEvents later
                         []
                         LoanManager_consumed.o.FALSE ->
                            controller_LoanManager(id, diff(events,{ev}),
                            deffered)
                         []
                         LoanManager_consumed.o.DEFER ->
                            controller_LoanManager(id, diff(events,{ev}),
                            deffered^<ev>)
                         )
                    )
               )
          )

          )

controller_AccountManager(id,events,deffered) = if card(events) > 1
     or #deffered > 0 then
                              SKIP
                                   else


          ((AccountManager_addevent?n?o!id?e ->
               controller_AccountManager(id,union(events,{(n,o,id,e)}),
               deffered) -- Event Arrival

                    []-- State Machine
          (card(events) > 0) & (
               |~| ev: events @ (
                    AccountManager_inevent?o!ev-> (
                         AccountManager_consumed.o.TRUE ->
                            controller_AccountManager(id, diff(events,{ev
                            }),deffered)--Add TreatDeferredEvents later
                         []
```

```
                        AccountManager_consumed.o.FALSE ->
                            controller_AccountManager(id, diff(events,{ev
                            }),deffered)
                        []
                        AccountManager_consumed.o.DEFER ->
                            controller_AccountManager(id, diff(events,{ev
                            }),deffered^<ev>)
                        )
                    )
                )
            )

            )

controller_ManagementDepartmentRef(id,events,deffered) = if card(
    events) > 1 or #deffered > 0 then
                            SKIP
                                else


            ((ManagementDepartmentRef_addevent?n?o!id?e ->
                controller_ManagementDepartmentRef(id,union(events,{(n,o
                ,id,e)}),deffered) -- Event Arrival

                    []-- State Machine
            (card(events) > 0) & (
                |~| ev: events @ (
                    ManagementDepartmentRef_inevent?o!ev-> (
                        ManagementDepartmentRef_consumed.o.TRUE ->
                            controller_ManagementDepartmentRef(id, diff(
                            events,{ev}),deffered)--Add
                            TreatDeferredEvents later
                        []
                        ManagementDepartmentRef_consumed.o.FALSE ->
                            controller_ManagementDepartmentRef(id, diff(
                            events,{ev}),deffered)
                        []
                        ManagementDepartmentRef_consumed.o.DEFER ->
                            controller_ManagementDepartmentRef(id, diff(
                            events,{ev}),deffered^<ev>)
                        )
                        )
                    )
                )

                )

bare_ManagementDepartment(id) = (simple_ManagementDepartment(id)
    [|{|ManagementDepartment_addevent|}|]
    controller_ManagementDepartment(id,{},<>)) \ {|
    ManagementDepartment_addevent|}

bare_ClientManager(id) = (simple_ClientManager(id) [|{|
    ClientManager_addevent|}|] controller_ClientManager(id,{},<>)) \
     {|ClientManager_addevent|}

bare_LoanManager(id) = (simple_LoanManager(id) [|{|
    LoanManager_addevent|}|] controller_LoanManager(id,{},<>)) \ {|
    LoanManager_addevent|}

bare_AccountManager(id) = (simple_AccountManager(id) [|{|
    AccountManager_addevent|}|] controller_AccountManager(id,{},<>))
     \ {|AccountManager_addevent|}

bare_ManagementDepartmentRef(id) = (simple_ManagementDepartmentRef(
    id) [|{|ManagementDepartmentRef_addevent|}|]
    controller_ManagementDepartmentRef(id,{},<>)) \ {|
    ManagementDepartmentRef_addevent|}


stm_ManagementDepartment(id) = State2(id)

State2(id) = ManagementDepartment_inevent!id^<stm>?e -> (
     (get_id(get_event(e)) == manageClient_I &
         ManagementDepartment_consumed!id^<stm>!TRUE ->
      ClientManager_op.get_n(e).get_id1(e).id!mc_I ->
          ClientManager_op.get_n(e).get_id1(e).id!mc_O ->
      ManagementDepartment_op!get_n(e)!get_id1(e)!id!mc_O -> State2(
          id))
     []
     (get_id(get_event(e)) == manageLoan_I &
         ManagementDepartment_consumed!id^<stm>!TRUE ->
     LoanManager_op.get_n(e).get_id1(e).id!ml_I -> LoanManager_op.
         get_n(e).get_id1(e).id!ml_O ->
     ManagementDepartment_op!get_n(e)!get_id1(e)!id!ml_O -> State2(
         id))
     []
     ((not(get_id(get_event(e)) == manageClient_I) or
     not(get_id(get_event(e)) == manageLoan_I ) )
     & ManagementDepartment_consumed!id^<stm>!FALSE -> State2(id))
)




stm_ManagementDepartmentRef(id) = State(id)

State(id) = ManagementDepartmentRef_inevent!id^<stm>?e -> (
     (get_id(get_event(e)) == manageClient_I &
         ManagementDepartmentRef_consumed!id^<stm>!TRUE ->
      ClientManager_op.get_n(e).get_id1(e).id!mc_I ->
          ClientManager_op.get_n(e).get_id1(e).id!mc_O ->
      ManagementDepartment_op!get_n(e)!get_id1(e)!id!mc_O -> State(
          id))
     []
     (get_id(get_event(e)) == manageLoan_I &
         ManagementDepartmentRef_consumed!id^<stm>!TRUE ->
     LoanManager_op.get_n(e).get_id1(e).id!ml_I -> LoanManager_op.
         get_n(e).get_id1(e).id!ml_O ->
     ManagementDepartment_op!get_n(e)!get_id1(e)!id!ml_O -> State(id
         ))
     []
     (get_id(get_event(e)) == manageAccount_I &
         ManagementDepartmentRef_consumed!id^<stm>!TRUE ->
     AccountManager_op.get_n(e).get_id1(e).id!ma_I ->
         AccountManager_op.get_n(e).get_id1(e).id!ma_O ->
     ManagementDepartment_op!get_n(e)!get_id1(e)!id!ma_O -> State(id
         ))
     []
     ((not(get_id(get_event(e)) == manageClient_I) or not(get_id(
         get_event(e)) == manageLoan_I )
     or not(get_id(get_event(e)) == manageAccount_I)) &
         ManagementDepartmentRef_consumed!id^<stm>!FALSE -> State(id)
```

```
          )
)


block_ManagementDepartment(id) = (bare_ManagementDepartment(id)
          [{|ManagementDepartment_inevent,
              ManagementDepartment_consumed,
          ManagementDepartment_op|} ||
          {|ManagementDepartment_inevent,
              ManagementDepartment_consumed,
          ClientManager_op, LoanManager_op,
          ManagementDepartment_op.n.x.id.w | n:MyNat, x: ID,w: {mc_O,
              ml_O}|}]
          stm_ManagementDepartment(id)
     )\union({|ManagementDepartment_inevent,
          ManagementDepartment_consumed|},{|ManagementDepartment_op.n
              .x.y.z | n: MyNat, x: ID, y: ID,
          z: MSG, member(get_id(z),union(ManagementDepartment_I,
              ManagementDepartment_O)),
          prefix(id,x) or prefix(id,y)|})

block_ManagementDepartmentRef(id) = (bare_ManagementDepartmentRef(
    id)
          [{|ManagementDepartmentRef_inevent,
              ManagementDepartmentRef_consumed,
          ManagementDepartment_op|} ||
          {|ManagementDepartmentRef_inevent,
              ManagementDepartmentRef_consumed,
          ClientManager_op, LoanManager_op,AccountManager_op,
          ManagementDepartment_op.n.x.id.w | n:MyNat, x: ID,w: {mc_O,
              ml_O,ma_O}|}]
          stm_ManagementDepartmentRef(id))\union({|
              ManagementDepartment_inevent,
          ManagementDepartment_consumed|},{|ManagementDepartment_op.n
              .x.y.z | n: MyNat, x: ID, y: ID,
          z: MSG, member(get_id(z),union(ManagementDepartment_I,
              ManagementDepartment_O)),
          prefix(id,x) or prefix(id,y)|})

block_ClientManager(id) = (bare_ClientManager(id)) \ {|
    ClientManager_inevent, ClientManager_consumed|}
```

```

block_LoanManager(id) = (bare_LoanManager(id)) \  {|
    LoanManager_inevent, LoanManager_consumed|}

block_AccountManager(id) = (bare_AccountManager(id)) \  {|
    AccountManager_inevent, AccountManager_consumed|}

block_System(id) = (block_ManagementDepartment(id)

          [{|ManagementDepartment_op,ClientManager_op,LoanManager_op
              |} ||
          {|ClientManager_op,LoanManager_op|}]

          (block_ClientManager(id) [ {|ClientManager_op|}  ||
          {|LoanManager_op|} ] (block_LoanManager(id)))
          )
          \{|ClientManager_op,LoanManager_op|}



block_SystemRef(id) =
          (block_ManagementDepartmentRef(id)

          [{|ManagementDepartment_op,
          ClientManager_op,LoanManager_op,AccountManager_op|} ||
          {|ClientManager_op,LoanManager_op,AccountManager_op|}]

          (block_ClientManager(id) [ {|ClientManager_op|}  ||
          {|LoanManager_op,AccountManager_op|} ] (
          block_LoanManager(id) [ {|LoanManager_op|} ||
          {|AccountManager_op|}] block_AccountManager(id)))
          )
          \{|ClientManager_op,LoanManager_op,AccountManager_op|}

channel loop

Loop = loop -> Loop


TEST = block_SystemRef(<bb>)

```

```

assert block_System(<bb>);Loop \ {|loop|} [T=
         (block_SystemRef(<bb>);Loop)
```