Università degli Studi dell'Insubria
Dipartimento di Informatica e Comunicazione

Dottorato di Ricerca in Informatica
XXI Ciclo

A SysML-based approach to requirements analysis and specification of real-time systems

Ph.D Thesis of
Pietro Colombo

Advisors
Prof. Alberto Coen Porisini
Prof. Luigi Lavazza

Supervisor of the Doctoral program
Prof. Gaetano Aurelio Lanzarone

to my wonderful family

Acknowledgments

I sincerely thank Alberto Coen Porisini for trusting, supporting and helping me since my graduation. I am also deeply grateful to Luigi Lavazza for his guidance and for stimulating my research activity all throughout the PhD studies. Alberto and Luigi have been representing for me indispensable points of reference.

A great thank to Vieri del Bianco and Sabrina Sicari who encouraged and advised me in countless occasions.

I am also thankful to numerous people of DICOM who shared with me enjoyable moments of everyday life (at coffee breaks, at launch time, on trip to mountain, on the train...). Many thanks to Mauro Santabarbara, Alberto Trombetta, Mauro Ferrari, Antonella Zanzi, Ignazio Gallo, Cristina Ghiselli, Michele Chinosi, Moreno Carullo, Andrea Perego, Andrea Spiriti, Gianmarco Gaspari, Brunella Gerla, Stefano Braghin, Loris Bozzato, Paolo Brivio.

Finally, a special thank to my family whose love has been fundamental.

Grazie di cuore

Varese, December 2008.

Abstract

Model-based development is particularly promising in the area of real-time and embedded systems, since it potentially increases the level of automatism and decreases the possible defects, improving the efficiency of the process and the quality of the product.

Model based approaches are effectively supported by notations such as SysML, a modeling language for Systems Engineering that has been recently adopted by the Object Management Group (OMG). SysML is of industrial origin, and it is likely that it will be widely adopted in industry for the development of real-time and embedded systems. Potential obstacles to the adoption of the language on a large scale are the lack of a methodology that drives the modeling activities and the full support for the definition of temporal aspects.

The main goal of this PhD work concerns the definition of model-based methodological guidelines to the usage of SysML for the analysis and specification of requirements and the early modeling of real-time systems.

Contents

1 Introduction **1**

1.1 The proposed solution . . . 2
1.2 Organization . . . 5
1.3 Publications . . . 6

2 SysML **7**

2.1 Structural elements . . . 9
2.1.1 Block . . . 9
2.1.2 Additional Structural Types . . . 10
2.1.3 Port . . . 10
2.1.4 Constraint Block . . . 11
2.1.5 Diagrams . . . 11
2.2 Behavioral elements . . . 13
2.2.1 Activity . . . 13
2.2.2 Diagrams . . . 14
2.3 Crosscutting constructs . . . 18
2.3.1 Allocations . . . 18
2.3.2 Requirements . . . 18
2.3.3 Profiles and Model Libraries . . . 19
2.3.4 Diagrams . . . 20

3 Modeling real-time systems with SysML **23**

3.1 Methodological guidelines to the use of SysML . . . 23
3.1.1 Requirements specification . . . 24
3.1.2 Methodological guidelines . . . 26
3.1.3 Real-time aspects . . . 27
3.2 The GRC problem . . . 29
3.2.1 The definition . . . 29
3.2.2 Towards the solution of the GRC problem . . . 30
3.3 Modeling the GRC with SysML . . . 31
3.3.1 Requirements definition . . . 31
3.3.2 Structural aspects . . . 31
3.3.3 Behavioral aspects . . . 36
3.3.4 Refining User Requirements . . . 50
3.4 A first assessment on SysML . . . 52
3.5 Related work . . . 53

4 Problem Frames **57**
4.1 The problem and the world 58
4.1.1 Phenomena 59
4.1.2 Domains 59
4.1.3 Interfaces 60
4.2 Descriptions 60
4.2.1 Domain properties 61
4.2.2 Requirements 61
4.2.3 Machine specification 61
4.3 Notational support 61
4.3.1 Context diagrams 62
4.3.2 Problem diagrams 62
4.4 Problem frames 63
4.4.1 Required behavior 64
4.4.2 Commanded behavior 65
4.4.3 Information display 66
4.4.4 Simple workpieces 66
4.4.5 Transformation 67
4.5 Concerns 68
4.6 Frame flavours 69
4.7 Frame Variants 70
4.8 Problem decomposition 71
4.9 Final remarks 73

5 Applying Problem Frames **75**
5.1 Informal System Requirements 75
5.1.1 The Requirements 76
5.1.2 The Origins of Requirements 79
5.2 The model of requirements 79
5.3 The PFs based model of requirements 81
5.3.1 Store/Collect Mission Data and Provide MIC 82
5.3.2 Collect and store data from truck 83
5.3.3 Tracking and Display 84
5.3.4 Alarm detection and notification 85
5.3.5 Map Annotation 86
5.4 Lessons learned 87

6 A SysML & Problem Frames based approach **91**
6.1 The Sluice Gate example 92
6.2 Representing PFs concepts 94
6.3 A PFs & SysML based methodology 96
6.3.1 SysML support for Problem Frames 96
6.3.2 Approach 96
6.4 Related work 106

7 A SysML based PFs catalogue **109**
7.1 Required behavior: one way traffic lights 109
7.2 Information display: odometer display 113
7.3 Commanded behavior: sluice gate control 118
7.4 Simple workpieces: Party plan editing 123
7.5 Transformation: Mailfiles analysis 129
7.6 Final remarks 131

8 Case study **133**
8.1 System description 133
8.2 The context of the problem 137
8.3 Looking inside the domains 144
8.4 The requirements 155
8.4.1 Fixed cycle operating mode 155
8.4.2 Fixed cycle pedestrian actuated operating mode 161
8.4.3 Semi-actuated operating mode 165
8.4.4 Fully-actuated operating mode 169
8.4.5 Manual mode 177
8.4.6 Preempted mode 180
8.4.7 Traffic lights states check 187
8.4.8 Change mode 193

9 Conclusions **199**

List of Figures

2.1 Diagram Taxonomy 8
2.2 Package diagram example 12
2.3 Block Definition diagram example 12
2.4 Internal Block diagram example 13
2.5 Parametric diagram example 13
2.6 State Machine diagram example 15
2.7 Activity diagram example 16
2.8 Sequence diagram example 17
2.9 Use Case diagram example 17
2.10 Allocation examples 21
2.11 Requirement diagram example 21

3.1 The support for requirements definition provided by SysML 25
3.2 The SysML proposed modeling approach 26
3.3 The railroad crossing regions 29
3.4 Annotated GRC 30
3.5 The Requirements Diagram for the GRC 32
3.6 The Block Definition Diagram for the railroad crossing system . . 34
3.7 The Internal Block Diagram for the block RailroadCrossing 35
3.8 The structure of the Controller block 36
3.9 State Machine Diagram for the block Gate 37
3.10 Activities allocated to `train` defined by means of a Block Definition Diagram 38
3.11 The Activity Diagram that defines the basic behavior of sensors . . 39
3.12 The State Machine Diagram that models the basic behavior of the train 39
3.13 The Activity diagram that describes the behavior of trains 40
3.14 The Activity diagrams that describes the action `MovingToSensor` . 41
3.15 Activities concurring the definition of the controller's behavior . . 42
3.16 The Activity Diagram that describes the controller's behavior . . . 43
3.17 The State Machine Diagram for the Controller block 44
3.18 CheckRaise and CheckLower activities behaviors 45
3.19 State Machine Diagram for the TrackState block 47
3.20 Definition of invariant properties for the Controller block 48

3.21 The Parametric Diagram which shows the allocation of the properties of the Controller block . . . 49
3.22 The Utility and Safety properties . . . 50
3.23 Safety and utility properties allocated to the `Controller` states . . 51

4.1 Domain types . . . 62
4.2 A simple Context diagram . . . 62
4.3 The notation to express the role of a domain . . . 63
4.4 A simple Problem diagram . . . 63
4.5 Required behavior: problem frame diagram . . . 65
4.6 Commanded behavior: problem frame diagram . . . 65
4.7 Information display: problem frame diagram . . . 66
4.8 Simple workpieces: problem frame diagram . . . 67
4.9 Transformation: problem frame diagram . . . 67
4.10 Commanded behavior frame concern . . . 69
4.11 A description variant of the Required behavior frame . . . 71

5.1 Visualization of mission data and alarms . . . 76
5.2 The geographical display . . . 77
5.3 The problem diagram for the monitor system . . . 80
5.4 The Problem diagram for the Collect and store mission data / provide MIC requirement . . . 82
5.5 The workpiece Problem diagram for the “Collect and store data from truck” requirement . . . 83
5.6 The problem diagram for the truck and its driver and devices, with indicative requirements . . . 83
5.7 The Display Problem diagram for the “Tracking and display” requirement . . . 84
5.8 The Problem diagram for the “Alarm detection” requirement with configurable alarm rules . . . 85
5.9 The problem diagram for the “Alarm detection and notification” requirement . . . 85
5.10 The workpiece Problem diagram for the “Map editing” requirement 86
5.11 The problem diagram for the “Map editing visual feedback” requirement . . . 86
5.12 The problem diagram for the “Map editing visual feedback” requirement . . . 87
5.13 The problem diagram for the “answer queries” requirement . . . 87
5.14 Problem Frame Diagram: Data repository enquiry . . . 88

6.1 The sluice gate. . . . 92
6.2 The sluice gate commanded behavior frame. . . . 93
6.3 Requirements: reaction to commands and events. . . . 94
6.4 Problem domain behavior. . . . 95
6.5 `bdd`, `ibd`: Representation of the Sluice Gate Control problem. . . . 98
6.6 `req`: User Requirements decomposition. . . . 100
6.7 `req`: System Requirements decomposition. . . . 101
6.8 `bdd`, `ibd`: Decomposition of the `Gate&Motor` domain. . . . 103

6.9 stm: The dynamics of the Gate and Motor 104
6.10 act: The activities associated with the components of the problem domain 105
6.11 bdd, par: Formal specification of the Lower Command requirement. 107

7.1 The one way traffic lights controlled behavior frame 110
7.2 The bdd representing a Lights Controller and two Light Units . . . 110
7.3 The ibd representing the structure of the problem domain 111
7.4 The stm specifying the behavior of light units 111
7.5 The stm representing the required behavior of the two light units . 112
7.6 The act diagram representing the behavior of the lights controller 112
7.7 The stm diagram representing the behavior of the lights controller 112
7.8 The odometer display: an information display problem frame . . . 113
7.9 The odometer display: problem context 114
7.10 The odometer display: the ibd representing the structure of the problem context 114
7.11 The act diagram specifying the behavior of the car 115
7.12 The act diagram specifying the behavior of the display. 115
7.13 The constraint specifying speed reporting 116
7.14 The par diagram describing the allocation of the requirements CountersTravelReq to the problem domains 117
7.15 Display frame concern 118
7.16 The commanded behavior problem frame diagram for the sluice gate controller 118
7.17 The sluice gate controller: context diagram 119
7.18 The structure of the problem domain 119
7.19 Elements of the Gate&Motor domain 120
7.20 The structure of the Gate&Motor domain 120
7.21 The behavior of the gate and motor 121
7.22 The act representing the behavior of the motor 121
7.23 The behavior of the gate and motor by means of a stm diagram . . 122
7.24 The state machine representing user requirements 123
7.25 The specification of the sluice gate controller 123
7.26 The problem diagram for the Party plan editing problem 124
7.27 The Party plan editing problem: context diagram 125
7.28 The internal structure of the problem domain Party Plan 126
7.29 Representation of the shared phenomena B 127
7.30 The state machine representing the user requirements 128
7.31 The state machine representing the Party editor machine 128
7.32 The mail file analysis transformation frame 129
7.33 The Mail file analyser: context diagram 129
7.34 The Mail files domain 130
7.35 The Report domain 130
7.36 The rules of the analysis specified as a constraint 131
7.37 The specification of the analyser 131

8.1 Intersection topography 134

8.2 The Context diagram for the intersection controller problem 136
8.3 The `bdd` that defines domains and phenomena of the intersection controller problem 144
8.4 The `ibd` that describes the architecture of the context of the intersection controller problem 144
8.5 The `stm` that describes the behavior of the domain *PedestrianPresenceDetector* 144
8.6 The `stm` that describes the behavior of the domain *VehiclePresenceDetector* 145
8.7 The `stm` that describes the behavior of the domain *EmergencyVehicleDetector* 146
8.8 The `bdd` that describes the internal components of the Block *VehicleTrafficStandard* 147
8.9 The `ibd` that describes the internal architecture of the Block *VehicleTrafficStandard* 148
8.10 The `act` that describes behavior of the Block *Switch* 149
8.11 The `stm` that describes the behavior of the Block *Lamp* 149
8.12 The `act` that describes the behavior of the Block *Sensor* 150
8.13 The `act` that describes the behavior of the Block *Controller* 151
8.14 The `stm` that describes the behavior of the Block *Controller* 152
8.15 The `act` that describes the activity *CheckState* 153
8.16 The `stm` that describes the behavior of the Block *VehicleTrafficStandard* 154
8.17 The Problem diagram for the intersection controller problem . . . 156
8.18 The Problem diagram for the fixed cycle operating mode 157
8.19 The Fixed cycle operating mode problem fits the Required behavior frame 157
8.20 The `stm` that expresses the requirements for the fixed cycle operating mode problem 158
8.21 The `stm` representing the machine specification for the fixed cycle operating mode problem 160
8.22 The Problem diagram for the fixed cycle pedestrian actuated operating mode 161
8.23 The Fixed cycle pedestrian actuated operating mode problem fits the Controlled behavior frame 162
8.24 The `stm` that expresses the requirements for the fixed cycle pedestrian actuated operating mode problem 163
8.25 The `stm` representing the machine specification for the fixed cycle pedestrian actuated operating mode problem 164
8.26 The Problem diagram for the semi-actuated operating mode 165
8.27 The semi actuated operating mode problem fits the Controlled behavior frame 166
8.28 The `stm` that expresses the requirements for the semi-actuated operating mode problem 167
8.29 The `stm` representing the machine specification for the semi-actuated operating mode problem 168
8.30 The Problem diagram for the fully actuated operating mode 170

8.31 The Fully actuated operating mode problem fits the Controlled behavior frame . . . 170
8.32 The stm that expresses the requirements for the fully-actuated operating mode problem . . . 174
8.33 The act diagram that describes the action *StoreSignals* . . . 174
8.34 The act diagram that describes the action *AnalyseData* . . . 174
8.35 The act diagram that describes the action *CalculateRate* . . . 175
8.36 The act diagram that describes the action *UpdateTimers* . . . 176
8.37 The Problem diagram for the manual operating mode problem . . 177
8.38 The Manual operating mode problem fits the Controlled behavior frame . . . 178
8.39 The stm diagram that describes the requirements of the problem Manual operating mode . . . 178
8.40 The stm diagram that describes the machine specification of the problem Manual operating mode . . . 179
8.41 The Problem diagram for the preempted operating mode problem 180
8.42 The Preempted operating mode problem fits the Controlled behavior frame . . . 181
8.43 The stm diagram that describes the requirements of the problem Preempted operating mode . . . 188
8.44 The stm diagram that describes the machine specification of the problem Preempted operating mode . . . 188
8.45 The problem frame diagram that describes the Traffic lights states check problem . . . 188
8.46 The Traffic lights states check problem fits the Required behavior frame . . . 188
8.47 The stm diagram that depicts the requirements of the Traffic lights states check problem . . . 189
8.48 The act diagram that describes the action *CheckVtsStates* . . . 190
8.49 The stm diagram that described the machine specification for the Traffic lights states check problem . . . 191
8.50 The Problem diagram for the Change mode problem . . . 192
8.51 The Change mode problem fits the Controlled behavior frame . . . 193
8.52 The stm diagram that depicts the requirements of the Mode change problem . . . 198

List of Tables

5.1 Truck and load data 80
5.2 Mission data 81

Chapter 1

Introduction

System development requires that the quality of both the production process and the resulting products are guaranteed. Quality can be obtained by means of methodologies, notations and tools that support the development. The adoption of these means by industry requires that they are accessible, easy to use and cost-effective.

Nowadays, model-based approaches represent an attractive solution for the software industry. Models lead software engineers to dominate the intrinsic complexity of systems by abstracting away from useless details, and by allowing designers/analysts to focus on the most relevant features. Moreover, models are usually platform independent, thus keeping application development separate from the underlying platform technology in the early stages of the lifecycle, and easing the porting. Whenever sufficiently expressive modeling languages are used, models can also be processed for various purposes, such as to validate and/or verify properties, to generate code, to generate test cases, etc.

In the last years several languages have been specifically defined to provide effective notations to support model-based methodologies. For instance, the Unified Modeling Language (UML) [57, 56] has achieved the status of standard for modeling (general purpose) software applications.

UML provides several mechanisms to extend the language in order to satisfy special purpose modeling requirements. For instance, many UML Profiles have been specifically defined to improve the basic expressive capabilities of the language and to support the specification of special purpose systems such as real-time and embedded systems.

The Object Management Group (OMG) has recently adopted SysML [53] (System Modeling Language) as a modeling language for Systems Engineering. SysML is a UML Profile that tries to overcome some of the weaknesses that arise when modeling systems with relevant non software parts. SysML is based on a subset of UML 2 [57, 56] and introduces new features that are expected to better support specification, analysis, design, verification, and validation of systems that include hardware, software, data, personnel, procedures, and facilities. SysML has been defined with the cooperation of industry leaders in the fields of Information and Communication Technologies, Avionics, Automotive and Academia. SysML

fully supports model-based specification and analysis, design, verification and validation techniques of a broad range of systems, thus it is likely that it will be widely adopted in industry. Moreover, it provides innovative features to better address the specification of requirements.

Since real-time and embedded systems are not comprised uniquely of software, but tend to involve hardware, devices, and often people, SysML is a natural candidate to support the development of this type of systems. Nevertheless, since SysML was released recently, there is still little evidence of its suitability to effectively support the development of real-time embedded systems.

Similarly to UML, SysML does not provide an adequate support for defining temporal properties. Moreover, currently there are no proposals concerning development methodologies based on SysML. The literature reports just a few experiences in using the language, but they mainly focus on describing the nature and the role of SysML diagrams, rather than suggesting methodological guidelines.

The main goal of this PhD work concerns the definition of a SysML based approach to the analysis of requirements and the early modeling of real-time and embedded systems. The aim is to help the modeler using SysML for organizing requirements and for defining a high level abstract model of the system.

1.1 The proposed solution

A first step towards the definition of a SysML based approach to requirements analysis consists in identifying the concepts and the type of use that such language has to support. Such requirements are addressed by the model for requirements and specifications of Gunter et al. [26], which introduces the artefacts underlying requirements analysis techniques and describes the relationships among such artefacts.

The choice of adopting such model is motivated by the fact that it is a sound and rigorous approach to requirements analysis and specifications, and it is independent from specific engineering aspects such as languages and notations to be used.

The model is essentially based on the following artifacts:

- the problem domain, which describes the behavioral and structural characteristics of the world where the problem is located.
- the requirements, i.e., the description of the user needs with respect to the solution of the problem, in other words the expectations of the user concerning the behavior of the problem domain after the implementation and deployment of the system.
- the machine domain, i.e, a model of the system that satisfies the requirements.

The main goal of the modeling activity consists of the specification of a machine that, once connected to the environment of the problem, satisfies the requirements.

Starting from these basic concepts, we propose an approach that consists in using specific SysML diagrams and elements for describing different aspects of the requirements, the problem domain and the machine domain. According to the SysML specifications, the language can also be combined with other notations in order to define properties that SysML alone cannot express. Since the support provided by SysML for expressing temporal properties is poor, this possibility is exploited by embedding, into SysML models, properties that are expressed in TRIO [23], a formal language for the specification and analysis of real-time systems.

Although the resulting modeling approach drives the analyst from the level of identifying the problem to the one of defining the specification, it does not provide any hint for approaching the analysis of complex problems, for which the passage from the identification to the specification is not direct. Moreover, consider that most of the realistic problems are too complex for being handled as a single problem and a further analysis activity is required. Such activity consists in structuring the complex problem as a collection of subproblems that are simpler to describe and to solve and in composing their descriptions in order to move towards the solution of the complete problem.

The Problem Frames approach proposed by Jackson in [37] is a requirements analysis methodology based on the principle of problem decomposition. The approach is built on the reference model of Gunter and provides all the methodological guidelines that are necessary for approaching the analysis of complex problems.

The approach of Jackson is based on the following concepts

- *domain*, a physical entity of the environment where the problem is located.
- *phenomenon*, an internal property of a domain.
- *requirement*, the user expectation with regard to the solution of the problem expressed in terms of relationships among the internal phenomena of domains.
- *machine specification*, the specification of the behavior of the machine that once connected with the other domains of the problem satisfies the requirements.

The core of this analysis technique is to recursively decompose a problem into simpler sub-problems until the identified sub-problems are of the simple type defined by some frames. A problem frame characterizes a class of elementary problems for which domain types and interfaces are known, and for which basic frame concerns, i.e., guidelines that must be addressed to go towards the solution of the problem, are provided.

In [37] Jackson defines five basic problem frames, by describing the shape of the problems and their concerns. Although only five basic frames are proposed, additional frames can be proposed. Notice that, according to Jackson, Problem Frames provide a context in which previously captured experiences can be effectively exploited for the analysis of other problems [36]. The application of Problem Frames to the analysis of complex problems is a way to evaluate whether other basic frames need to be defined.

Although the Problem Frames approach provides guidelines for driving the analyst through the different activities of system analysis, and a great interest for the technique was shown by the requirements engineering research community, at present the approach is not widely used in industry.

Problem Frames are usually presented in literature by means of simple case studies that aim at illustrating particular aspects of the approach. Case studies of realistic complexity are still missing. Notice that the empirical evidence of the applicability of the technique to realistic problems may represent a first step towards its application in industrial processes.

A further weakness that hinders the adoption of the technique in industry concerns the lack of an effective notation and tools that support the approach. Jackson proposes a simple notation to describe only parts of the structural characteristics of a problem but does not support the definition of the behavioral aspects.

Several papers [43, 8] discussed the integration of UML with Problem Frames in order to provide a more effective notation to the Problem Frames methodology. The results of this integration showed that although the combination improves the effectiveness and usability of Problem Frames, UML does not support the approach at the correct abstraction level.

We propose a combined approach to requirements analysis that combines SysML with the methodological guidelines of Problem Frames.

SysML performs better than UML since it provides constructs to support modeling at the correct level of abstraction and since it overcomes some of the weaknesses of UML.

The approach consists of five main activities:

- *Requirements definition*, during which the user requirements are introduced by means of informal descriptions
- *Context analysis*, which consists in the analysis of the problem and identification and description of the problem domains and shared phenomena
- *Domain definition*, which consists in connecting the domains among them by defining the structural organization of the whole system.
- *Domain refinement*, which consists in defining the relevant behavioral properties of the domains
- *Requirements refinement*, during which the informal requirements are precisely specified by using a formal notation.

All these activities are supported by means of SysML diagrams and constructs.

Both the modeling language and the requirements engineering approach take advantage from the combined approach. From the point of view of SysML, the modeling language is enriched with a sound requirements engineering approach, while from the Problem Frame perspective, the analysis approach is supported by an expressive notation.

A first step towards the validation of the approach is to test its effectiveness by applying it to the modeling of the catalogue of basic Problem Frames proposed in

[37]. Such catalogue is composed of a simple case study for each of the five basic frames. Although the intrinsic complexity of the proposed problems is relatively low, such basic frames play a fundamental role, since the whole decomposition process aims at reducing complex problems to a set of simple problems that should fit these frames.

In order to complete the validation of the approach and to test its scalability, we apply it to the modeling of a case study of industrial complexity. The adopted case study concerns the modeling of a controller for a four way intersection system.

1.2 Organization

This thesis is organized as follows:

- Chapter 2 introduces the fundamental characteristics of SysML, describing its main constructs and its diagrams. Some simple examples are used to clarify the role and the use of SysML constructs and diagrams. Notice that this chapter provides a reference for all the subsequent uses of the language.

- Chapter 3 proposes a set of guidelines for using SysML for the requirements analysis of real-time system. The guidelines are based on the reference model for requirements and specification of Gunter et al. [26]. This chapter also describes the application of the approach to the modeling of the Generalized Railroad Crossing (GRC) problem [30], a well known benchmark for real-time systems.

- Chapter 4 introduces the Problem Frames approach. It describes the basic concepts of the approach as well as its notational support and methological issues concerning the application of the approach to the analysis and structuring of problems. The chapter provides a reference for all the subsequent uses of the approach.

- Chapter 5 illustrates how Problem Frames can be used for the requirements analysis of a case study of industrial complexity that concerns a system for monitoring the transportation of dangerous goods. This chapter, besides showing the application of the approach to the decomposition of a complex problem, discusses the identification of two new basic frames.

- Chapter 6 describes the combined approach for requirements analysis and structuring based on Problem Frames and SysML. More specifically, it proposes a set of guidelines for using specific SysML diagrams and constructs as a notational support to different activities of the Problem Frames approach. The chapter also illustrates the application of the guidelines to the analysis of a simple case study concerning a sluice gate irrigation system.

- Chapter 7 proposes a first step towards the validation of the combined approach, more specifically it describes the application of the approach to the basic frames catalogue proposed by Jackson in [37].

- Chapter 8 aims at testing the scalability of the combined approach. It proposes the modeling of a case study of industrial complexity concerning a controller for a four way intersection system supporting multiple operating modes. This chapter shows how it is possible to dominate the intrinsic complexity of a real problem by applying the decomposition techniques, and how it is possible to model the single subproblems and how such subproblems can be recomposed to get the general description of the requirements and machine specifications of the complete problem.
- Chapter 9 draws some conclusions by summarizing the main results and by discussing plans for future works.

1.3 Publications

Part of the material presented in this PhD thesis has already been published:

- A preliminary form of the guidelines and of the case study presented in Chapter 3 are published in [14].
- The application of the Problem Frame approach to the modeling of the requirements of the system for monitoring dangerous goods transportation described in Chapter 5 are published in [12].
- The combined approach Problem Frames - SysML illustrated in Chapter 6 has been published in [13].
- The application of the combined approach for the modeling of the catalogue of basic Problem frame has been published in a preliminary form in [11].

Chapter 2

SysML

In the last years the practices to describe systems are moving from the usage of document centric approaches towards model centric solutions.

In a document centric approach, activities such as requirements description, specification, system design, development, and testing are usually performed by independent work groups and described into distinct documents with different *ad hoc* notations. Some problems may arise in the communication among such groups, due to inconsistent descriptions and to the partial view of the system under development.

A model centric approach exploits the usage of a unique model that describes numerous aspects and properties of the system under development. Moreover, such description is performed by using a unique notation. The shared understanding of system requirements and design provides some benefits. For instance, it favors the validation of requirements and the identification of risks. The usage of a central model helps in managing complex system development since it favors the separation of concerns via multiple views of the integrated model. Moreover, it facilitates impact analysis of requirements, design changes and it supports incremental development. Finally, a model centric approach favors early verification and validation of properties defined on the model thus diminishing the effects of errors at the last development phases. As a result, the adoption of a model centric approach improves the overall design quality by reducing errors and ambiguity and the risk of inconsistent descriptions.

A model centric approach needs to be supported by an effective notation that has to describe in a expressive and intuitive way all the properties and the aspects of the system under development. In 2003, the Object Management Group (OMG) expressed the requirements for such notation in the UML for Systems Engineering Request for Proposal [64]. Since the Unified Modeling Language (UML) [57, 56] was an affirmed standard notation that supported software development, the goal of the request was to extend such language by providing it with the notational elements that are needed to support Systems Engineering activities.

The resulting extended notation was intended to support the modeling of a broad range of systems, including hardware, software, data, personnel, procedures, and facilities. Moreover, it had to support the analysis, specification, design, and

verification of complex systems by describing the involved system properties in a precise and efficient manner. Finally, the notation was intended to favor the communication among the various participants and stakeholders involved in the description of the system.

The Systems Modeling Language (SysML) [53] is a new general purpose language that was expressly defined to satisfy the requirements imposed by the UML for System Engineering Request for Proposal. SysML is based on UML 2, more specifically, it reuses a subset of the elements of UML 2 and provides innovative elements to support the requirements imposed by the Request For Proposal (RFP).

SysML is designed to provide effective constructs for modeling a wide range of system engineering problems. The language supports the specification, analysis, design, verification and validation of systems that may include hardware, software, information, processes, personnel, and facilities.

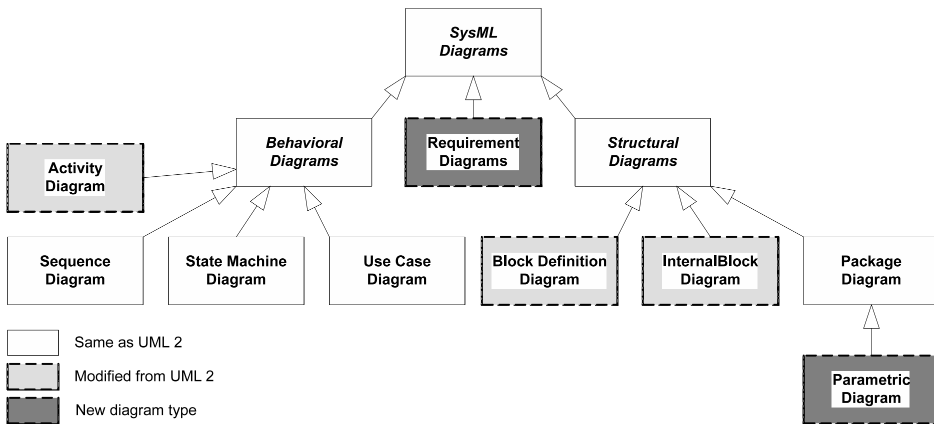

Figure 2.1: Diagram Taxonomy

SysML is a visual modeling language that provides a graphical notation with (informal) semantics and different types of diagrams, which can be used to describe both the behavioral and the structural aspects of a system. The variety of diagrams and a very rich notation allow designers to show in a quite intuitive way the relationships among the different elements involved in a model. As shown in Figure 2.1, most of the diagrams are inherited from UML and are adopted without any change. Other diagrams are adapted so that they are compatible with the new constraints defined by SysML. Moreover, completely new diagrams are defined to extend the language in order to fully support system modeling.

In what follows, a short overview of the most important SysML constructs is provided. First of all structural elements, i.e., the elements used to define static properties of a system, are introduced; then dynamic elements, i.e., the constructs used to define the behavioral aspects are described; finally, cross cutting elements are presented.

A very simple example concerning a control system is used to introduce some of the basic elements of the SysML notation.

2.1 Structural elements

In this section the main SysML constructs supporting the definition of the static aspects of a system are introduced.

2.1.1 Block

Blocks are the basic structural units for system description. A block is a collection of features that describe different aspects of an element of a system. Blocks support the definition of both structural and behavioral properties; they are used to represent the state of an element, and which operations or activities such element may exhibit. Blocks may be recursively composed of other Blocks, and such a mechanism is used to describe the structure of hierarchical systems.

More specifically, blocks are characterized by intrinsic properties named Block-Properties that specify:

- Part: A block may be composed of instances of other blocks. A property of a block may refer to an element of a system that is contained in the block itself. In SysML block instances are named Parts. Parts are used to show the components of the system.
- Reference: A property of a block may refer to an element of a system that is not directly contained in the block itself. References are used to share common information across multiple elements of a model.
- Value: A value property is used to represent a measurable characteristic of a block. A value property may be characterized by a unit and dimension.
- Port: A property used to represent the types of interactions that can occur between blocks.
- Constraint: A property used to constrain other Block, Reference, Value and Port properties.
- Operation: A service that can be requested from a Part to effect behavior. An operation has a signature that defines the formal parameters.

BlocksProperties are used to define information about the system state and behavior, and can be used also to define the relationships among the elements at any level of the structure associated with a system. Blocks can be associated with one another by means of the following types of relationships:

- Dependency: A relationship between two blocks in which a change to the properties of one element affects the properties of the other one.
- ReferenceAssociation: A relationship between two blocks that specifies a connection among their instances.
- PartAssociation: A relationship used to complement the definition of a Part property, in which a block is connected to other blocks owned by the block itself.

- SharedAssociation: A relationship used to complement the definition of a Reference property, in which a block is connected to other external blocks not directly owned by the block itself.
- Generalization: A relationship between a more general block and a more specific one. The extended block is fully consistent with the properties of the more general one (it inherits all its properties), and provides additional properties.
- BlockNamespace Containment: Used to represent that blocks are defined in the namespace of a containing block.

2.1.2 Additional Structural Types

Besides Blocks, SysML provides additional constructs to represent fundamental structural properties of a system.

ValueTypes define values that express information about a system. Values may be used to type properties (Value properties), operation parameters and other SysML elements. ValueTypes may include a dimension and a measure unit associated with the values.

DataTypes are types characterized by pure values. DataTypes includes both primitive types and enumeration types.

Signals represent asynchronous stimula occurring between instances of blocks. A Signal is a generalizable element type and is defined independently from the classes handling it.

2.1.3 Port

Ports are interaction points between a Block or a Part and the environment through which data and signals are exchanged. Ports are BlockProperties, hence, they are part of the definition of a Block. Morever, they are connected with one another by means of connectors. SysML introduces two different kinds of Ports named Standard Port and Flow Port.

Standard Port

Standard Ports are used to specify interaction points through which a Block provides/requires a set of services to/from its environment. A Standard Port is a communication point that receives or sends signals. Receiving or sending signals may correspond to the invocation of services. In such a case, services are defined by means of operations and are collected into interfaces that can be either provided or required by a Block. Standard Ports are commonly used in the context of service oriented architectures for the specification of synchronous communication mechanisms.

FlowPort

Flow Ports are used to specify which kind of item may flow between a Block and its environment. The specification of what can flow is done by associating a

FlowProperty or a FlowSpecification with the FlowPort. A FlowProperty specifies the type (Block, ValueType, DataType or Signal) of a single element that flows from/to the port and a FlowDirection property (in/out/inout), which specifies whether such element enters or leaves the port. FlowSpecification defines an input/output characterized by a set of FlowProperties. Notice that FlowProperties involved in the definition of a FlowSpecification may be characterized by different types of elements and by different FlowDirections.

Flow Ports are categorized into Atomic and Non-Atomic FlowPort depending on whether they are characterized by a FlowProperty or by a FlowSpecification, respectively.

2.1.4 Constraint Block

A Constraint block is an extended block that contains the definition of a property that constrains the properties of one or more elements of a system. Constraints are characterized by a ConstraintProperty, i.e., a BlockProperty characterized by a textual expression that predicates on a set of variables named Constraint parameters. SysML does not prescribe the usage of any specific language to express ConstraintProperties. Properties can be defined by using natural language or formal/mathematical languages. The modeler is free to adopt the language that he/she considers the most suited for the modeling context.

A Constraint block defines a generic form of constraint that can be used in multiple contexts. In fact, Constraint blocks are defined independently from the modeling system: they are used by allocating the parameters of their instances to the properties of domains.

2.1.5 Diagrams

The structure of systems is defined by using the previously introduced constructs within Package Diagrams `pkg`, Block Definition Diagrams `bdd`, Internal Block Diagrams `ibd` and Parametric Diagrams `par`.

Package diagrams `pkgs` are used to organize the model by partitioning the model into packageable elements and establishing relationships between the packages and/or model elements within the package. SysML Package diagrams also allow the modeler to define Views and ViewPoints. A Viewpoint is a specification of the rules for the definition and the usage of a view for the purpose of addressing a set of stakeholder concerns, while a View is a representation of the system from a certain viewpoint. `pkgs` provide mechanisms to group model elements into name spaces, favoring the reuse of blocks and solving homonymity problems. Other packages may access or import model elements defined in a `pkg`. The package diagram in Figure 2.2 shows the definition of a View that imports elements from existing packages and comforms with a given Viewpoint named AnalysisViewpoint.

Block Definition Diagrams describe the system hierarchy by introducing the blocks that represent the system components and the existent relationships among them, such as composition, association, specialization.

`bdds` support the definition of Blocks, DataTypes, ValueTypes, Constraint

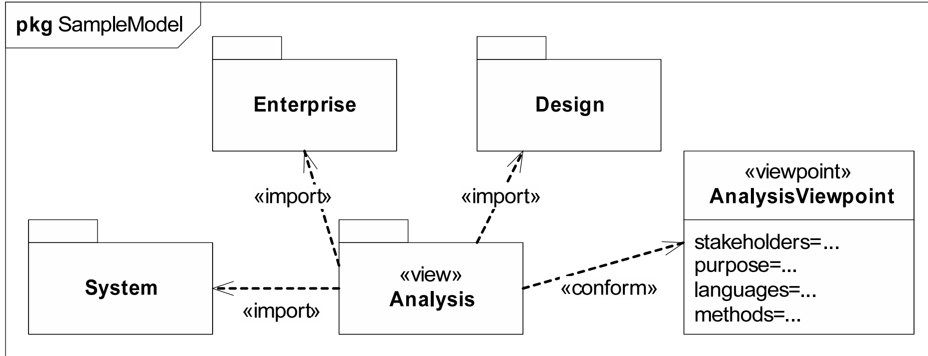

Figure 2.2: Package diagram example

Blocks. All these elements may be defined by specifying all their internal properties. For instance, a Block can be defined as equipped with FlowPorts or StandardPorts, while ValueTypes may provide the definition of the Unit and Dimension that are associated with their value. Moreover, all these elements are graphically represented by an *ad hoc* notation in the `bdd`. `bdds` also support the specification of different types of relationships among such elements such as Dependencies, ReferenceAssociations, PartAssociations, SharedAssociations, Generalizations and BlockNameSpace Containment.

Internal block diagrams describe the internal structure of a block in terms of its properties, and connectors. More specifically, since Parts, Values, References, and Ports are BlockProperties associated with a block, they may be shown within `ibds`. Binding connectors among such elements reflect the relationships defined in the `bdd` where their types are defined.

`bdds` and `ibds` are an extension of the UML Class Diagrams and UML Composite Structure Diagrams, respectively. Such diagrams allow a modeler to separate the definition of model elements from their effective usage. In fact, all the elements defined in a `bdd` are provided with the definition of the properties that will characterize their instances shown in `ibds`.

As an example, consider the `bdd` shown in Figure 2.3 and the `ibd` of Figure 2.4. The `bdd` describes the Blocks composing the package System shown in Figure 2.2, while the `ibd` shows the internals of the Block ControlSystem introduced in the `bdd`.

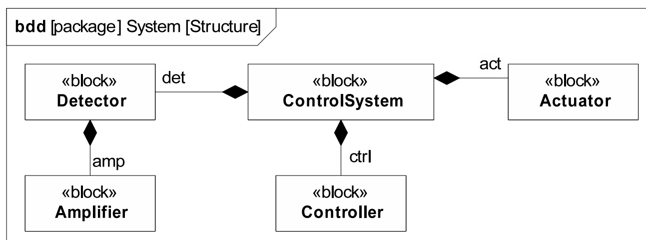

Figure 2.3: Block Definition diagram example

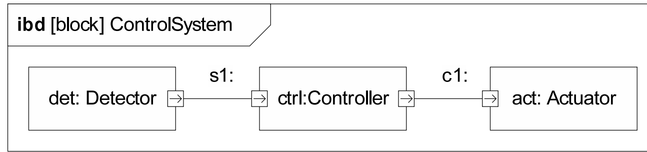

Figure 2.4: Internal Block diagram example

Parametrics diagrams are an extended type of `ibd` that are used to allocate the parameters of Constraint Blocks to system elements properties. Such properties are the same that can be shown in a `ibd`, hence, a `par` is similar to an `ibd`, with the exception that the only connectors that may be shown are binding connectors connected to constraint parameters. `par`s and `ibd`s share a similar function in the modeling process, in fact, Constraints Blocks are defined in `bdd`s like other Blocks and are used in `par`s.

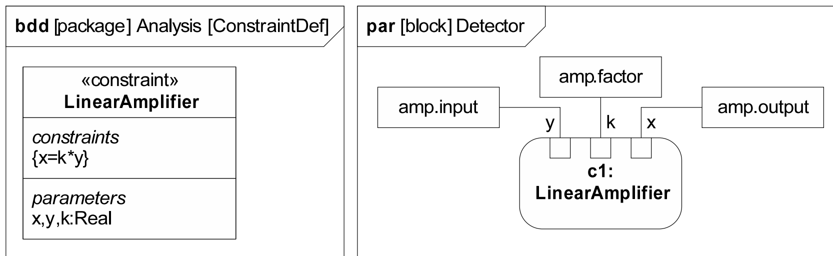

Figure 2.5: Parametric diagram example

An example of constraint definition and allocation is shown in Figure 2.5. More specifically, the figure reports a `bdd` that defines a Constraint Block named LinearAmplifier, and a `par` that shows the usage of such constraint. The constraint is instantiated and its parameters (x, y and k) are allocated to internal parts of the Amplifier. Namely, y = amp.input, k = amp.factor and x = amp.output. As a result, amp.output is constrained to be equal to amp.input times amp.factor.

2.2 Behavioral elements

In this section the main SysML constructs supporting the definition of the behavioral aspects of a system are introduced.

2.2.1 Activity

Activities are fundamental constructs provided by SysML to support the behavior specification. Activities take a set of input data and use them to produce a set of output data. Activities are composed of basic Actions, which represent the basic unit of behavior specification. Basic Actions include CallBehaviorAction, AcceptEventAction and SendSignalAction.

Behavioral modeling concerns the identification of the activities and their basic actions, besides the identification of their inputs, outputs, and conditions for coordinating their execution.

In SysML activities are Blocks, and their instances are executions. Whenever an instance of an activity is created, the activity starts its execution. Moreover when an instance of an Activity is destroyed, the activity stops its execution. Notice that in both cases the vice versa is also guaranteed.

Since Activities are blocks, they can be defined in Block Definition Diagrams. Moreover, the same types of relationships that are used in a `bdd` for standard Blocks can also be used for Activities. Whenever a Composition relationships is defined among two Activity blocks, implicit constraints on the execution of such activities are also imposed. More specifically, in case the execution of the activity involved in the composition relationship terminates, the execution of the contained activity is also stopped. Moreover, in case an instance of the container activity is created, also an instance of the contained one is defined, and both are executed. A Composition relationships can be defined between two activities whenever an activity invokes another activity.

Activities may receive input data, and during their execution they may provide output data. Data processed by activities are represented by means of ObjectNodes, instances of Blocks, DataTypes or ValueTypes that specify the items that flow through the activities during their execution. ObjectNodes are defined in `bdd`s and may be linked to Activity Blocks by using Associations.

ObjectNodes may flow between activities at different rates. SysML provides constructs to specify the rate at which entities flow among activities. Rate specifies the number of ObjectNodes that flow through activities per time interval (it does not refer to the rate at which the value changes over time). The definition of both Discrete and Continuous flows is supported. Continuous rate is a special case where the increment of time between items approaches zero.

SysML allows a modeler to express different behaviors associated with the processing of ObjectsNodes. Items can be either bufferized or not bufferized, moreover special policies may be defined, e.g., to specify whether the arrival of a new item overrides the previous one.

The control on the execution of activities is carried out by sending control values. Control Values are managed by Control Operators, i.e., logical operators that are used to enable or disable actions. Control Values may trigger the execution of activities, and may also block the execution of running activities.

2.2.2 Diagrams

SysML provides Activity Diagrams (`act`), Sequence Diagrams (`seq`), State Machine Diagram (`stm`) and Use Case Diagram (`uc`), to support the definition of the behavioral aspects of systems.

Although different types of diagrams are introduced, all of them share the same behavioral constructs.

State Machine diagrams (`seq`) are used to describe behaviors in terms of systems states and transitions. A State is a period of the life of an instance of a Block during which such instance satisfies some conditions, performs some ac-

tions or waits for some events. A transition is a relationship between two states that specifies that an instance in the first state will enter the second state and will perform specific actions when given event will occur and a given condition is satisfied.

`stm`s describe possible sequences of states and actions through which the element instances can proceed as a reaction to discrete events such as operation invocations.

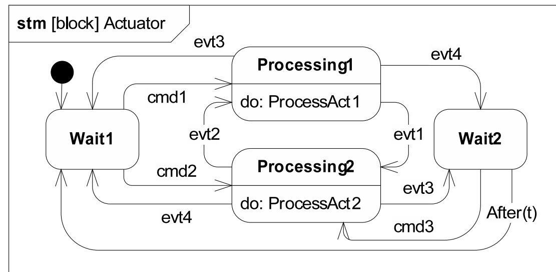

Figure 2.6: State Machine diagram example

As an example, consider the `stm` of Figure 2.6 that shows the four internal states of the Block Actuator, and defines the transitions triggered by timed conditions, by external events named cmd1 and cmd2, and by events named evt1, evt2, evt3 and evt4 generated by the action ProcessAct1 and ProcessAct2 at the end of their processing (the generation of the events is not specified by the current diagram, here the events are simply referred to). Whenever the states Processing1 and Processing2 are reached, the actions ProcessAct1 and ProcessAct2 are executed.

Activity Diagrams (`act`) are used to describe the control flow and the data flow among actions. Activities are defined by composing basic Actions. Such composition is based on operators, such as InitialNode, FinalNode, DecisionNode, MergeNode, ForkNode, JoinNode, which support the definition of the data flow and the control flow among the involved actions.

An `act` can be considered as a special case of a state diagram in which all the states are actions and the transitions are triggered by the completion of the actions in the source state. The purpose of this diagram is to focus on flows driven by internal processing, as opposed to state machine diagrams where the evolution is driven by external events.

A simple example of `act` is shown in Figure 2.7. The `act` describes the behavior of the Block Controller. The Action Elaborate is triggered by a Signal named Input and by an ObjectNode Value, the action generates an output that depending on its value may cause the generation of signals named Cmd1 or Cmd2.

As in UML, Sequence Diagrams describe the control flow between actors and systems or between parts of a system. `sd`s define interactions among instances of blocks. `sd`s are characterized by two dimensions: the vertical dimension represents time, while the horizontal one shows the instances of blocks that are involved in the interaction. Interactions are sets of messages that are exchanged by the involved instances.

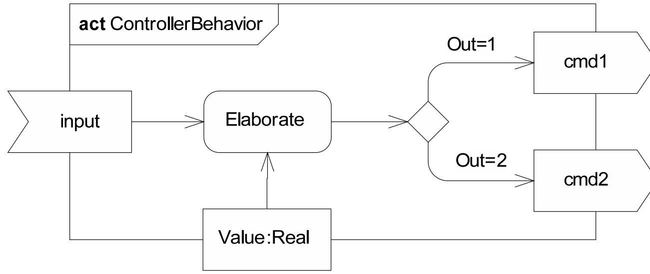

Figure 2.7: Activity diagram example

The life time of the block instances is represented by Lifelines, while the execution time associated with the execution of a given action performed by a block instance is represented by ExecutionSpecification bars. sds provide constructs to support the specification of complex interaction forms. InteractionUse is used to specify that a part of an Interaction is described by an externally defined Sequence diagram. While CombinedFragment introduces interaction operators that contrain the sequence of messages that characterizes an interaction. More specifically, Interaction operators include: Weak sequencing (seq), Alternatives (alt), Option (opt), Break (break), Parallel (par), Sctrict Sequencing (strict), Loop (loop), Critical Region (critical), Negative (neg), Assertion (assert), Ignore (ignore) and Consider (consider). Each operator in turn requires specific operands. For instance, the Option combination fragment is used to model a sequence that will occur only if a certain condition (specified by the interaction operand) is satisfied. In other words, opt is used to model simple -if then- statements.

Messages involved in the definition of interactions may represent asynchronous signals passed between blocks instances, synchronous invocations of an activity, or return values. Moreover special types of messages such as CreationEvent and DestructionEvent are used to create an instance of a block that will be involved in the interaction, and to terminate a lifeline, respectively.

Additional constructs such as DurationConstraint and TimeConstraint can be used to constrain the duration of the message forwarding, and the sending and receiving time of messages, respectively.

A simple example of Sequence diagram is shown in Figure 2.8. The sd shows the basic interactions among instances of Detector, Controller and Actuator. Detector sends a Signal Input to Controller, which in turn elaborates the Signal and depending on the result of the elaboration sends a Signal Cmd1 or Cmd2 to Actuator.

Use Case Diagrams (ucs) are used to describe the usage of systems. ucs specify the use cases (i.e., a collection of functionalities) of the system, and the actors that interact with one of more of these use cases. Actors represent a taxonomy of user types or external systems. Besides introducing actors and use cases, ucs support the definition of relationships among them. Relationships include Generalizations between actors, and Generalizations, Extends, and Includes between use cases, and Communication between actors and use cases.

More specifically, actors are connected to use cases via Communications. Gen-

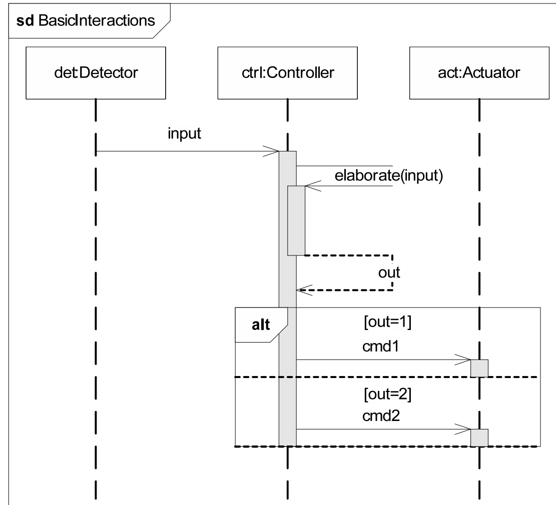

Figure 2.8: Sequence diagram example

eralizations between use cases provide a mechanism to specify variants of the base use case, while generalizations establish compatibility between actors: a specialization of actor A can participate in all the use cases as A. Include provides a mechanism for factoring out common functionalities which are shared among multiple use cases. Extend provides a mechanisms to define optional functionalities, which extend the base use case at defined extension points under certain conditions.

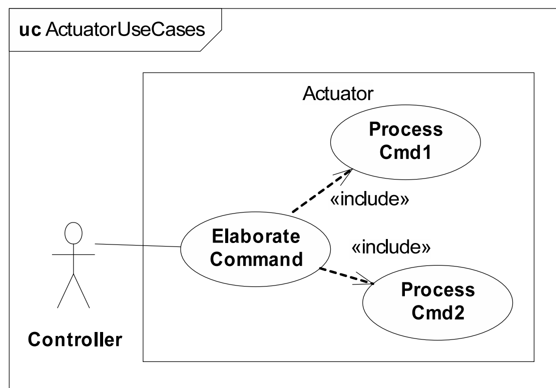

Figure 2.9: Use Case diagram example

The simple example of Use Case diagram in Figure 2.9 shows the basic use cases of the block Actuator and the actor, that is the Controller, that accesses them.

2.3 Crosscutting constructs

In this section constructs supporting the definition of both the structural and behavioral aspects of a system are introduced.

2.3.1 Allocations

Allocation is the term used by system engineers to denote the mapping of elements onto the various structures of a model. Allocations can provide an effective mean for navigating the model by establishing cross relationships, and ensuring that the various parts of the model are properly integrated. The allocation mechanism is based on the definition of the relationship Allocate.

Allocate is a dependency relationship that is used for associating elements of different types at an abstract level. The relationship is used for assessing model consistency and anticipating future design choices.

Behavior allocation relates to the system engineering concept of separation of concerns. It is required that two independent models of functions and structure exist. Then a mapping between elements of such models can be defined by introducing an allocate relationships that specify, for the interested activity, which block provides it.

Flow allocation maps flows defined in behavioral behavioral diagrams into flows defined in structural representations.

Finally, structure allocation supports the separate definition of logical and physical representations of a system. Representations at an abstract level are mapped into representations at a more concrete level.

A further construct named AllocateActivityPartition is used to allocate the elements represented in an Activity diagram to structural elements. More specifically, the Actions within the partition must result in an Allocate dependency between the Activity used by the Action and the element that the partition represens.

2.3.2 Requirements

Requirements specify properties that must be satisfied by the system under development. Requirements may affect different aspects of the system. More specifically: structural requirements specify the static properties of the system to develop; functional requirements describe the functionalities that the system must exhibit; while non functional requirements involve the definition of properties such as performance or temporal aspects.

SysML provides a construct named Requirement (a stereotype of UML Class) to represent text based requirements. Requirement are characterized by two properties named *id* and *text*. The former is used to assign a unique identifier to a requirement, while the latter is the textual description of the properties. Notice that SysML does not force the usage of any particular language to express properties, hence the modeler is free to adopt the language that he/she considers the most suited for his/her needs.

SysML also provides relationships to relate requirements to other requirements and to modeling elements.

Containment relationships support the definition of composite requirements. Exploiting such relationships, a complex requirement can be described as composed of multiple sub-requirements. A recursive usage of the relationship supports the definition of requirements hierarchies.

DeriveReqt is a dependency relationship between two requirements that specifies that a certain requirement (the *client*) is derived from another one (the *supplier*). Notice that no formal semantics is associated with the term Derive.

Copy is a dependency relationship between two requirements (a client and a supplier) that specifies that the textual description of the client requirement is a copy of the textual description of the supplier. Copy relationships are defined to support requirements re-use across several projects. In fact, in order to avoid inconsistent identification of requirements that come from different projects, local copies are defined, and a new id is associated with each copy.

Satisfy is a relationship between a requirement and a system element. Such relationship specifies that the system design element is intended to satisfy the requirement.

TestCase is a method for verifying whether a requirement is satisfied. In SysML, a test case is intended to be used as a general mechanism to represent any of the verification methods for inspection, analysis or test. TestCase is characterized by a verdict property that specifies the verification result.

Verify is a relationship between a requirement and the TestCase that verifies whether a system fulfills the requirement. In this case the supplier is the Requirement while the client is the TestCase.

Refine is a relationship between a model element and a requirement. It can be used to specify that the model element, that is the client, refines the requirement, the supplier. As an example, the textual description of a functional requirement may be refined by means of Activity diagrams. Refine can also be used to specify the opposite situation, that is, it can be used to express that the textual description of a requirement refines a less fine grained model element.

2.3.3 Profiles and Model Libraries

A Profile is a stereotyped package that contains mechanisms to define model elements customized for a specific domain. Model elements are extended version of metaclasses defined in external metamodels. Their customization exploits the usage of extension mechanisms such as Stereotypes, Tagged values and Constraints. A Stereotype is a new type of model element that extends an existing type or meta-class. A Tagged value is an explicit definition of a property as a name-value pair, while a Constraint is a semantic condition on model elements.

SysML provides the following additional relationships to support the definition of Profile and Model Libraries:

- Extension is a relationship between a Metaclass and a Stereotype, and specifies that the Stereotype extends the properties of the Metaclass.
- Generalization is a relationship between two stereotypes, and specifies that one stereotype inherits the characteristics of the other one and extends them by providing additional properties.

- ProfileApplication is a dependency relationship between a package and a Profile, and specifies that the Package uses the elements that are collected in the Profile.
- MetamodelReference is a dependency relationship between a Profile and a Metamodel, and specifies that the Profile refers to elements that are defined in the Metamodel.

SysML also supports the definition of Model Libraries, that is, stereotyped packages that contain model elements which are intended to be reused by other packages. Notice that a Model library differs from a Profile since it does not extend the metamodel. For instance, let us consider the real-time domain. In such context a model library could provide the dimensions and unit for managing time, while a profile could extends the metamodel by introducing innovative constructs like clocks, durations and intervals, to support the definition of temporal properties.

2.3.4 Diagrams

Allocation is supported by different types of constructs and can be applied to the elements shown by different types of diagrams. Allocation are defined in `bdds` and `ibds` by using two BlockProperties named allocatedFrom and allocatedTo. In case of behavioral allocation, a property named allocatedTo can be added to the compartment of Actions shown in an Activity Diagram. Activity partitions can be defined by tracing swimlanes that allocate actions defined in an Activity Diagram to a certain model element.

Alternatively to all the proposed mechanisms, stereotyped comments allocatedFrom and allocatedTo can be applied to all types of model elements in all structural and behavioral diagrams. Notice that no *ad hoc* diagram type is defined to support Allocation.

As an example, consider the `act` diagrams shown in Figure 2.10. Such diagrams express the allocation of the action Elaborate to the block Controller by means of three different notations: Diagram a) by means of the BlockProperty allocatedTo, Diagram b) by means of an element AllocateActivityPartition, and Diagram c) by means of a stereotyped annotation.

The definition of requirements is supported by a dedicated diagram named Requirement Diagram (`req`). `reqs` contain both the definition of Requirements and the relationships that can be defined among them. In case of relationships that involve a model element and a Requirement, model elements can be shown in the diagram as well.

A simple example of Requirement diagram is shown in Figure 2.11. The basic requirements defined in the diagram describe the structure of the whole control system and the basic behavior of the block Controller.

The definition of Profile and Model Libraries is supported by Package diagrams and Block Definition Diagrams. The former are used to specify any type of relationship that can be defined between elements that are represented by stereotyped packages, i.e., Metamodel, Profile and Model Library. The latter is used to introduce the definition of new Stereotypes, Tagged Values and Constraints.

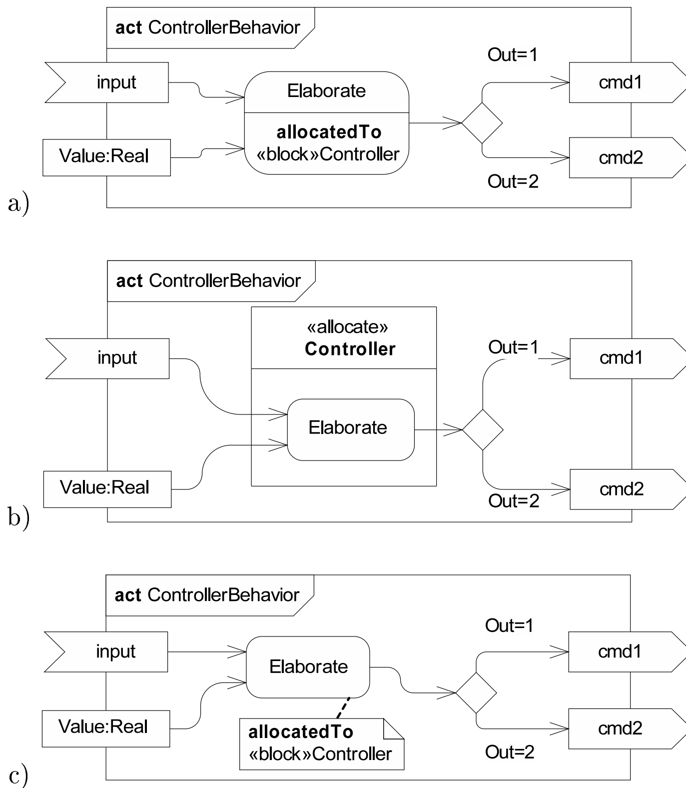

Figure 2.10: Allocation examples

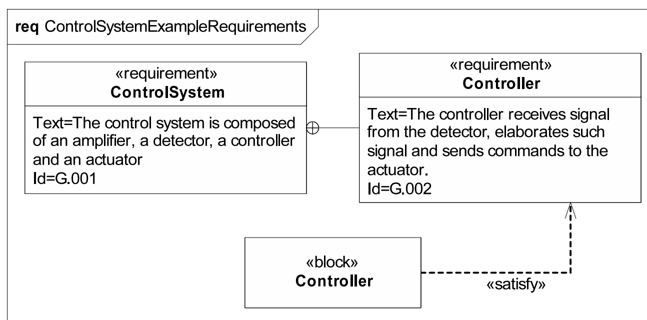

Figure 2.11: Requirement diagram example

Chapter 3

Modeling real-time systems with SysML

This chapter proposes a model-based approach exploiting SysML for the analysis of requirements and the early modeling of real-time and embedded systems. More specifically, it introduces methodological guidelines to help the analyst to organize requirements and to define an abstract model of the system. Successful developments strongly depend on the early stages of requirements organization. Moreover, early models properly support the definition of the basic aspects at the abstraction level needed to support the development phase.

An additional goal of this work is to test the effectiveness of SysML in modeling temporal aspects. In order to evaluate the proposed modeling approach and the language capabilities, SysML is used to model one of the best known benchmarks for real-time systems: the generalized railroad crossing (GRC) problem [29].

The chapter is organized as follows: Section 3.1 illustrates the methodological guidelines for modeling real-time systems using SysML. Section 3.2 introduces the GRC problem. Section 3.3 presents the GRC SysML model. Section 3.4 discusses the effectiveness of SysML, and identifies some qualities and limitations. Finally, Section 3.5 presents the most relevant related work.

3.1 Methodological guidelines to the use of SysML

The System Modeling Language (SysML) is a visual modeling language that provides a graphical notation based on different types of diagrams and that can be used to describe both the structural and the behavioral aspects of a system [54]. Most of the diagrams are inherited from UML, even though new diagrams are defined to extend the language in order to fully support system modeling.

SysML supports a model-centric approach to the definition of systems. The modeling process is carried out through several complex activities performed by different actors (e.g., analysts, designers and testers) that cooperate in order to define the whole system. Such actors can take advantage of the single notation provided by SysML to describe the different artefacts built during system defini-

tion, thus easing communications and facilitating the common understanding of the system.

SysML is intended to be methodology independent [54] and therefore it is possible to use it in the context of different approaches and methodologies. This chapter presents methodological guidelines based on the reference model by Gunter et al. [26] for the definition of hybrid[1] and real-time systems.

3.1.1 Requirements specification

This section describes the requirements specification phase, concentrating on the nature of the requirements analysis and description. Given a software development problem, the following types of artefacts are introduced:

- The Problem Domain, which specifies some given, established properties of the environment where the result of the development —i.e., the machine— will operate. These descriptions, concerning both structural and behavioral aspects, are "indicative", that is, they are (generally) given and cannot be modified.

- The Machine Domain, which specifies the target of the development activity, which is the final artefact that we aim at releasing to the user. Notice that the machine includes both hardware and software components, although often hardware is given and the development only addresses (a part of) the software components.

- The Requirements, which specify the user expectations concerning the behavior of the 'system'. Requirements are given by means of "optative" descriptions.

The main goal of the modeling activity consists in the specification of a machine that —once integrated in the environment where the problem lies— satisfies the user requirements.

According to Gunter's reference model, the Machine Specification describes the intersection of the Machine Domain and the Problem Domain, where the interaction takes place. In fact, the Machine Domain and the Problem Domain share phenomena, such as events, states, data, information, etc.

The Machine Specification is a rigorous description of the structural and behavioral properties of the Machine. At the problem definition level, the behavior of the machine is specified by taking into account only the shared phenomena (i.e., no internal details of the machine have to be mentioned).

SysML supports the definition of system requirements through a new type of dedicated diagram, named Requirements Diagram (`req`).

The basic building block of Requirements Diagrams is the `requirement`, which is defined as a stereotype[2] of a UML Class. Each requirement element comprises

[1] A hybrid system is a dynamic system that exhibits both continuous and discrete dynamic behavior

[2] Notice that SysML, being defined as an extension of a part of the UML meta-model, uses the same extension mechanisms provided by UML (e.g., tagged values, stereotypes and profiles).

an identifier and a textual description to informally provide the semantics of the `requirement` block.

The language also allows the modeler to define relations among requirements and between requirements and other elements of the model. In particular, SysML introduces a derivation relation (as an extension of UML dependency) and a containment relation (as an extension of UML association). The containment relation is used to decompose complex requirements into sets of simpler ones. The derive relation is used to state that a requirement is derived from another one; however it has not a formal semantics and can be used to show extensions of properties.

As a result, SysML Requirements Diagrams can be seen as a sort of hypertexts, with derivation and containment relations playing the role of links. Requirements Diagrams can specify structural and behavioral characteristics, as well as features that must be provided or constraints that must be satisfied. Nevertheless, the precise nature of every `requirement` is not represented explicitly, that is, it must be inferred from the textual description.

In order to model requirements concepts according to the reference model by Gunter et al. we introduce the following stereotypes : `User-Requirements`, `Problem-Domain-Requirements` and `Machine-Spec-Requirements`, to characterize `Requirement` blocks.

The overall system requirements can be viewed as the union of the description of the Problem domain, the User requirements and the Machine specification. Figure 3.1 sketches the proposed requirements classification, while an example of `req` diagram is reported in Figure 3.5.

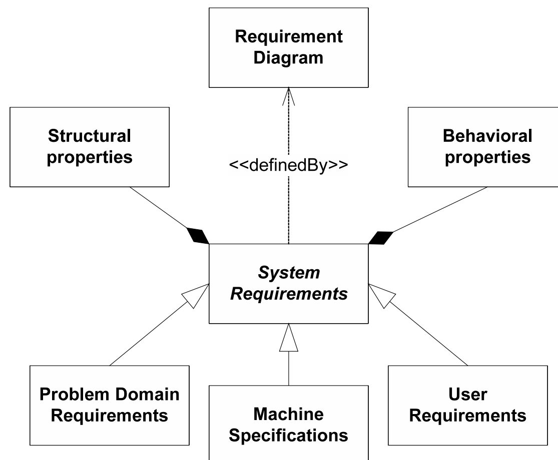

Figure 3.1: The support for requirements definition provided by SysML

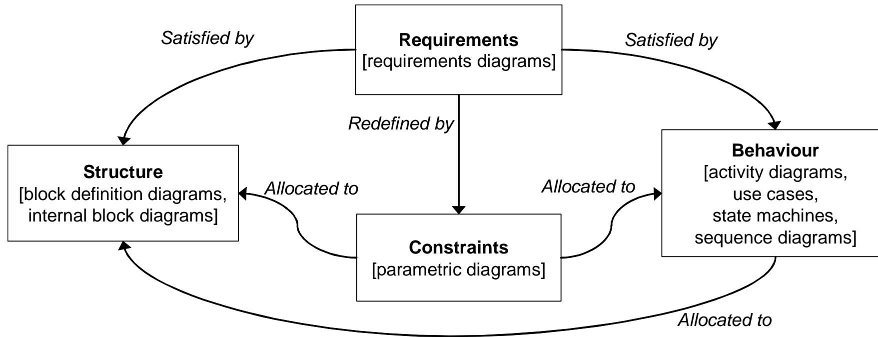

Figure 3.2: The SysML proposed modeling approach

3.1.2 Methodological guidelines

The model of requirements based exclusively on a Requirements Diagram is just a little more than structured text. Such description is generally not even sufficient to guarantee the basic properties of requirements (completeness, consistency, etc.). Thus, we need a more precise and detailed description of the environment, the user requirements and the machine specification, in which both the structural and the behavioral aspects are defined.

In order to address this issue SysML provides different types of diagrams and constructs.

The choice of which constructs or diagrams to use in the definition of a required property is mainly related to the characteristics of the requirement itself. A complex requirement (characterized by an hybrid nature) can be recursively decomposed into simpler homogeneous structures. As a result of such decomposition we obtain elementary properties whose nature can be structural, behavioral or transversal, i.e., associated with both static and dynamic elements.

Figure 3.2 shows how SysML diagrams can be used to describe properties according to their nature. Block Definition Diagrams and Internal Block Diagrams allow one to refine the definition of structural requirements. Activity Diagrams, State Machine Diagrams, Use Case Diagrams and Sequence Diagrams support the definition of the behavior of a system. Moreover, Parameter Diagrams describe cross-cutting constraints that possibly concern both structural and behavioral elements of the system. In fact, the expressiveness of constraints largely depends on the language used to write them.

The proposed requirements specification approach is composed of the following activities:

1. Refine the elements of the Requirements Diagram classified as `Problem--Domain-Requirements` and `Machine-Spec-Requirement` into precise descriptions of their structure. This is done by defining a `bdd` in which a block is introduced for each element of the Requirement diagram that defines both

structural and behavioral aspects. Moreover an `ibd` describing the architecture of the whole system is introduced by instantiating each block defined in the `bdd`. Note that also some aspects of the machine structure need to be modeled. In fact, there are many problems that require that the machine "keeps track" of what is happening in the environment. For this purpose, the machine builds an internal "model" of the environment. This situation is well described by the "model building frame" in [37]. Therefore, `bdds` and `ibds` are often used to describe the internal models of the machines.

2. Describe the behavior of the blocks previously defined in the `bdd`. This can be done by defining activities by means of `activity` blocks in a `bdd`, and providing for each block an Activity Diagram, or by defining a State Machine Diagram that describes the internal state evolution. Activity Diagrams are used whenever behaviors involve more than one block or whenever it is necessary to specify a flow of data among blocks and activities. State Machine Diagrams are used when the behavior is associated with a single block, and it can be described by means of state evolution.

3. Refine the `User-Requirement` to describe the desired behavior of the environment. This can be done by using `bdd`, `act` and `stm` as described in the previous step. Usually, there is no need to use any structural description, since in general user requirements concern the properties of the environment, already described in the model of the problem domains.

Notice that although we presented the requirement specification activities as a sequence of steps, the process may be iterative, with frequent adjustments and refinements involving also the Requirements Diagram.

Following the proposed guidelines the modeler has to describe elementary properties by means of the diagrams provided by the language. This activity leads him/her to organize the definition in several diagrams, each of which providing a view on an aspect of the same property. The usage of cross cutting diagrams and constructs allows the modeler to logically compose in a unique model the involved parts. For instance, an activity can be decomposed into subactivities that can be allocated to different structural components.

3.1.3 Real-time aspects

Temporal properties, like any other kind of non functional properties, can be informally expressed by means of Requirements Diagrams as introduced in the previous section. Temporal requirements can be either imposed by the user or given characteristics of (some of) the problem domains that are contained in the system environment. In either case the analyst needs to refine the informal properties expressed by means of textual descriptions.

SysML inherits the time modeling capabilities of UML and therefore it does not provide any specific construct for expressing temporal properties. Like in UML, time is considered a single aspect of behavioral modeling. Concepts such as *time* and *duration* are expressed by means of metaclasses of the `SimpleTime` package, which provides also actions to observe the passing of time. `SimpleTime` features do not support distributed models, in other words, the package can be

used only for the specification of systems characterized by a unique centralized time model that is shared by all the components of the system. This weakness can be partly justifiable in the case of the UML basic profile, since it is not meant for the definition of real-time system and most of the specified models are not characterized by time dependent behaviors. However, this cannot be accepted in the case of SysML, since such modeling language is designed for system engineering.

This limitation is particularly onerous in the advanced phases of the development process and may affect schedulability analysis, simulation and verification activities.

UML overcomes this weakness by introducing several profiles that extend the expressive capabilities of its basic version. Some of them, like the UML Profile for Schedulability, Performance and Time (SPT) [51] and the UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms [50] have been approved by OMG as official UML extensions to express temporal aspects.

SysML has been released only recently, and no official profile has been proposed yet. Moreover, profiles are usually design oriented, thus they do not support the requirements analysis phase at the correct level of abstraction.

As an alternative to profiles, the communities of analysts and modelers proposed mechanisms and notations that aim at refining the semantics of particular elements of a model using formal methods. Several approaches to define temporal properties have been proposed in the literature. Some of them, such as Timed automata [2], use an operational style; others, such as TCTL [1] logic, use a descriptive style.

However, SysML does not natively support the usage of any specific approach, and does not suggest to use any specific language to express temporal properties.

On one hand the lack of supported notations could be considered a weakness of the language; but on the other hand, the independence from any specific methodology and language allows the modeler to use the specification techniques he/she considers the most suitable for the project. For instance, in the development of critical systems the modeler could prefer formal notations that enable the application of formal methods in order to increase the confidence in the correctness of the system. On the contrary, informal notation could be used in the development of less critical systems.

Our approach exploits SysML Constraint blocks to express temporal properties. More specifically, constraints are introduced and attached to model elements in Parametric Diagrams.

The properties in constraint blocks can be written using different languages. We propose to use TRIO, a first order logic language augmented with a temporal domain, arithmetic operators and temporal operators, since it is widely known in the community of real-time analysts and since it has been applied to the specification of several real-time and safety critical systems [9].

A detailed explanation of TRIO formulas and temporal operators can be found in [23].

3.2 The GRC problem

The Generalized Railroad Crossing problem was proposed in 1993 by Heitmeyer [29] as a general benchmark for the development of real-time systems. Since then, it has been used extensively to test several formalisms, methodologies and tools aimed at easing the development of real-time systems [30, 29]. The problem is quite simple, yet it exposes time constraints we usually find in hard real-time systems.

3.2.1 The definition

The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R (see Figure 3.3). Trains travel over R on K tracks in both directions. Trains may proceed at different speeds, and can even pass each other. Only one train per track is allowed to be in R at any moment. Sensors indicate when each train enters and exits regions R and I. A gate function g from the real-time domain to the real interval [0,90] describes the state of the gate according to the inclination of the bar, g(t)=0 indicating that the bar is down (gate closed) and g(t)=90 indicating that the bar is up (gate open). A sequence of occupancy intervals is also defined, where each occupancy interval is the maximal time interval during which one or more trains are in I.

The problem is to develop a system to operate the crossing gate that satisfies the following two properties:

- Safety Property: The gate is closed during all occupancy intervals.
- Utility Property: The gate is open whenever this is possible without violating the safety property, and according to the features of the gate. For instance, when the last train in the crossing leaves and no train is approaching, the gate must open. The utility property is required, since a permanently closed gate would satisfy the safety property.

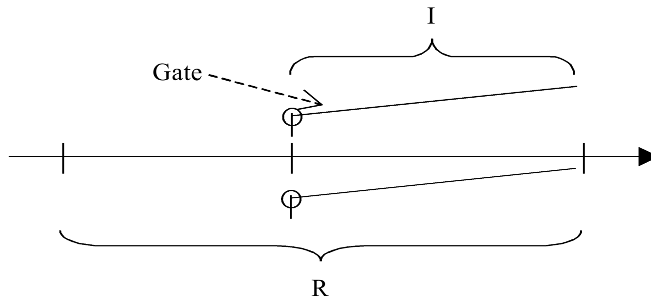

Figure 3.3: The railroad crossing regions

Notice that Figure 3.3 shows trains going in only one direction. We adopt this simplification, since it has been shown that the solution of this simplified problem can be trivially extended to the general case.

3.2.2 Towards the solution of the GRC problem

Let us introduce the relevant points in the interest region (see Figure 3.4):

- point RI indicates the position of the entrance sensor;
- point RO indicates the position of the exit sensor;
- point II indicates the position of the sensor that detects trains entering region I.

Therefore, RI-RO defines zone R and II-RO defines zone I.

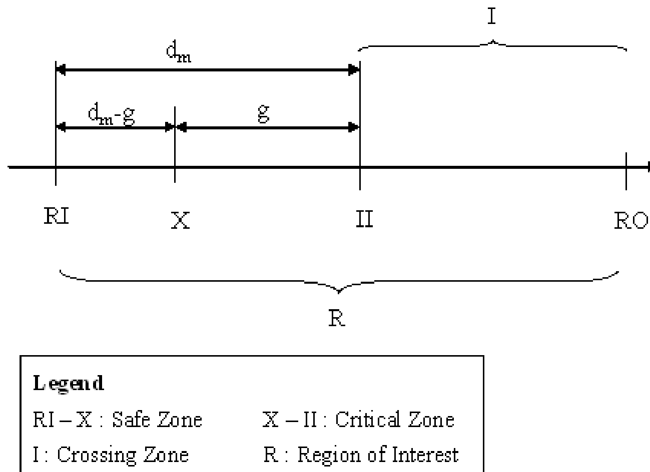

Figure 3.4: Annotated GRC

A set of temporal constants describes maximum and minimum times for crossing the various zones, as well as the gate opening and closing times:

- dm and dM: minimum and maximum time taken by a train to cross RI-II zone;
- hm and hM: minimum and maximum time taken by a train to cross zone I (i.e., II-RO zone);
- g is the time taken by the bars of the gate for moving from the completely open to the completely closed position (or vice versa).

As a consequence, point X is defined as follows: when a train enters zone X-II it is time to start closing the gate, in order to ensure that the bars will be completely lowered when the train arrives at II (i.e., when the train enters the crossing zone I, or II-RO). Zones RI-X and X-RO are also referred to as Safe zone and Critical zone, respectively. The exact position of X depends on the speed of each train, which is not known precisely. Thus the system cannot determine the

right moment when a given train is at point X. However, it is clear that if the system is safe for the fastest train, it is safe also for other trains. In order to have the gate closed when the fastest trains arrive at II, we must begin to close the gate dm-g seconds after the train entered region R. In this way when the fastest trains arrive at II the bars will be down and the crossing will be safe.

In order to satisfy the safety property, the bars can be raised only when both the crossing zone and the critical zone are empty. Similarly, in order to guarantee the utility property, the system must start opening the gate as soon as the critical zone and the crossing zone become empty.

3.3 Modeling the GRC with SysML

This section presents the application of the proposed guidelines to the GRC problem. According to the described approach, we start structuring the informal requirements descriptions provided by the user by means of a SysML Requirements Diagram(`req`). Then, requirements are refined into a more structured and precise way. Initially the structural aspects of problem and machine domains are introduced by means of `bdd` and `ibd` that define properties associated with domains, their internal composition and interconnections. Then, behavioral aspects of the components of the environment and of the machine are described by means of `bdd`, `act`, `stm` and `par` diagrams. Finally, User requirements are defined by means of Constraint Blocks and `par` diagrams using formal notations.

3.3.1 Requirements definition

The first activity consists in organizing requirements in a `req` diagram. The informal properties provided in Section 3.2 are taken into account and organized in the `req` diagram reported in Figure 3.5.

The main goal of modeling is the definition of a control system that operates a gate at a railroad crossing, and is represented by the `RailroadCrossingSpecification` requirement.

The initial informal description presents the whole system as composed of sensors, a gate, tracks partitioned in regions, and trains. All these elements, following the previously introduced criteria, represent the *problem domains*, while the controller is the *machine*, that is the target of the specification activity. Properties associated with problem domains are defined by means of `Problem-Domain-Requirement` elements, while properties associated with the machine are expressed using `Machine-Spec-Requirement` elements.

Additional properties express the expectations of the user concerning the behavior of the system. Such properties are expressed by means of `User-Requirements` elements. According to the previous descriptions, the GRC problem introduces two user requirements: the `UtilityProperty` and `SafetyProperty`.

3.3.2 Structural aspects

According to our guidelines, modeling initially focuses on the description of the structural aspects of the environment where the “machine” will be integrated. This

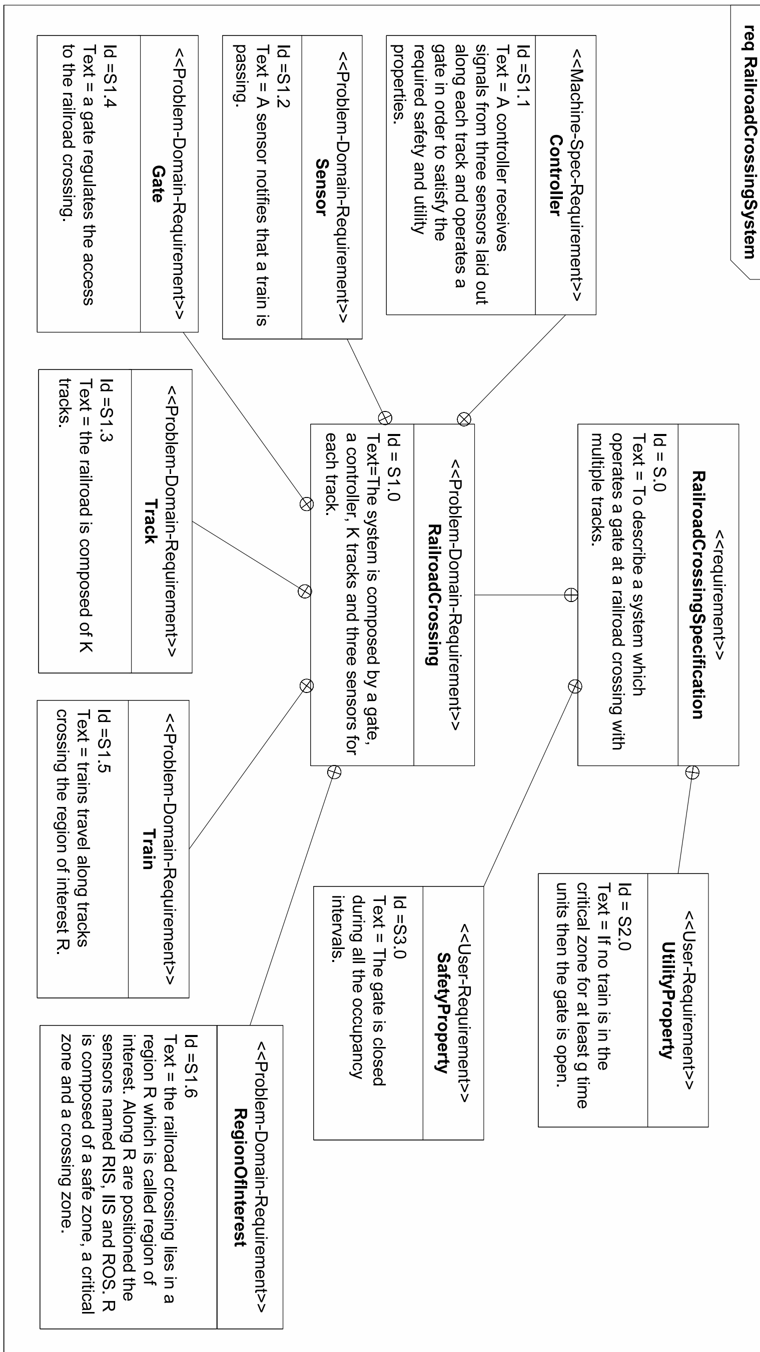

Figure 3.5: The Requirements Diagram for the GRC

activity is similar to the definition of an architecture for a software system [22], although in this case the subject is the problem domain and the "components" have a physical nature (trains, tracks, gate, etc.).

The Block Definition Diagram representing the GRC structure is built starting from the Requirements Diagram of Figure 3.5.

The system is modeled as a single block recursively decomposed into blocks down to the basic elements of the structure. Each block defines a collection of features, properties, and operations that characterize part of the system.

The modeling activity starts identifying domains. More specifically, all the elements classified as `Problem-Domain-Requirements` and `Machine-Spec-Requirements` characterized by both structural and behavioral aspects are described by introducing homonymous blocks in the `bdd`.

Besides identifying domains and defining the corresponding blocks, we also analyze the textual requirements and refine the `bdd` accordingly. For instance, the Requirements Diagram specifies that the controller receives data from three sensors and sends commands to the gate. This is represented in the `bdd` by equipping the `Controller` block with three input flow ports of type `SensorData` and a standard port labeled `ctrlGP`. The latter needs to be connected to an interface in order to operate the gate. Such interface, named `GateI`, is provided by block `Gate` and defines the `raise` and `lower` commands that can be used to operate the gate.

The `Sensor` block owns an attribute that identifies the track on which it is positioned, and is characterized by an output flow port of type `SensorData`. A sensor generates `SensorData` signals, which are sent through the port. A `SensorData` signal contains attributes that identify the originating sensor.

The Block Definition Diagram is shown in Figure 3.6.

System architecture

Once the structural elements have been defined, we have to show how the instances of these components are interconnected.

The instances of blocks previously defined in Figure 3.6 are connected according to the information reported in the Requirements Diagram; the resulting system architecture is shown in Figure 3.7.

A `Gate` instance labeled `gate` and providing the interface `GateI` is directly connected to `ctrl`, an instance of the `Controller` block that requires interface `GateI`. Three instances of the `Sensor` block are connected to `ctrl` through flow ports of type `SensorData`. For each track `i` (with $1 \leq \texttt{i} \leq K$) we have three sensors labelled `RIS[i]`, `ROS[i]` and `IIS[i]`. `RIS[i]` is the sensor at the beginning of the region of interest R, `IIS[i]` is the sensor at the beginning of the crossing zone I and `ROS[i]` is the sensor at the end of the crossing zone of the i-th track. Each sensor produces `SensorData` signals, indicating on which track the sensor operates and the position of the sensor on the track. For example, the `RIS[i]` sensor generates a `SensorData` signal characterized by an attribute `trackID=i` and an attribute `sensorID=RIS`, thus indicating that the sensor is placed on the i-th track, at the entrance of the safe region.

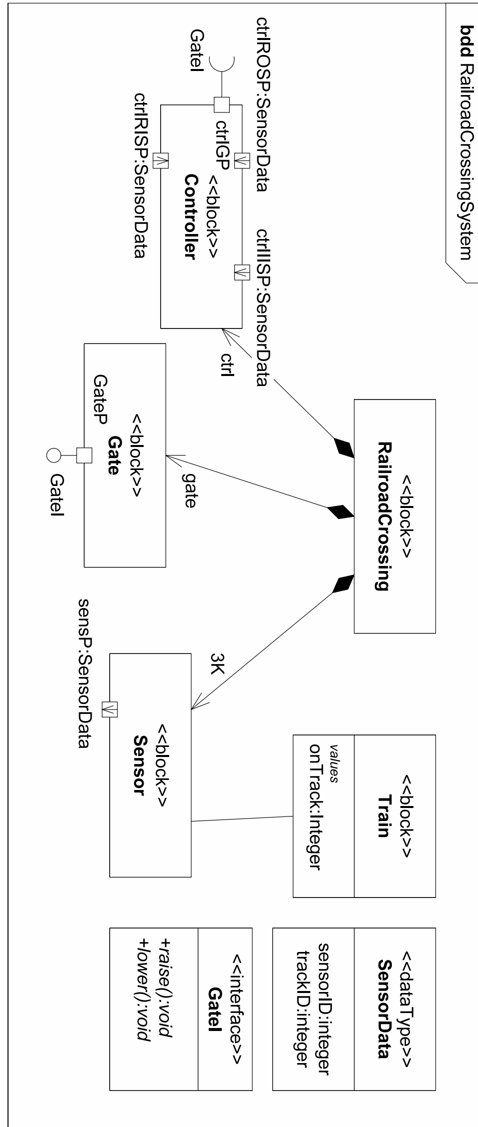

Figure 3.6: The Block Definition Diagram for the railroad crossing system

Figure 3.7: The Internal Block Diagram for the block RailroadCrossing

Structural decomposition

Domains are complex structures that may be decomposed into simpler ones. According to the proposed methodology, both the domain decomposition and the definition of internal properties are represented by means of `bdd` and `ibd`.

Figure 3.8 presents both the `bdd` that describes the internal structure of `Controller`, and the `ibd` that shows the internal architecture of such block.

The `Controller` block defines a set of attributes that are used to keep track of the state of the regions. The `CCR` attribute represents the number of trains currently in the critical region, while `CIR` represents the number of trains in the region of interest. Other attributes define constant values, such as the minimum and the maximum time required to cross a specific region (`dm`, `dM`, `hm`, `hM`) and the maximum number of tracks that compose the system (`K`).

The `Controller` block is composed by blocks of type `TrackState` and `GateState` that represent local models of the tracks and of the gate, respectively. Notice that altought `CCR` and `CIR` attributes are enough to keep track of the current state of the regions monitored by the controller, the usage of local models of the tracks and of the gate improves the analysis activities of the controller.

Block `TrackState` represents the current state of a track. Since we are not interested in the exact position of trains, the position is defined in an abstract way: out of region R, in RI-X, in X-II, in II-RO. The position of the train is determined according to the information provided by the sensors, and taking into account the

flowing of time. According to the specifications, there will be K instances of this block.

Each `TrackState` object receives signals from all the sensors connected to the controller and changes its state whenever the signal comes from a sensor positioned on the track it represents. Notice that (since we are still in the specification stage)

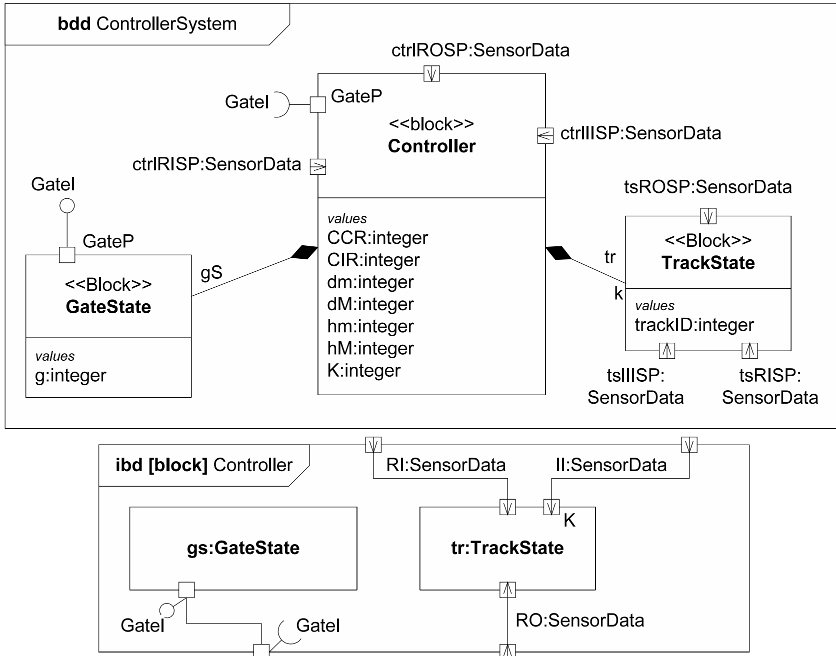

Figure 3.8: The structure of the Controller block

the `GateState` and `TrainState` do not need to be concretely implemented in the `Controller` machine. They represent just abstract knowledge about the domain. In other words, they are just a sort of summary of the sequences of domain events that are generated in the environment and viewed by the machine. This is perfectly coherent with the model building frame [37] and the idea that a machine has often to incorporate a model of the environment in which it has to operate.

3.3.3 Behavioral aspects

Next step consists in specifying the behavior of the structural elements introduced in the `bdd` (of Figure 3.7) according to the textual descriptions defined in the Requirements Diagram. In particular, we address both the indicative descriptions of the behavior of the problem domain and the optative descriptions of the required behavior of the domain and of the machine.

First we identify an element in the static description (e.g., one of the blocks in Figure 3.7); then we look into the Requirements Diagram in order to retrieve

the information concerning the behavior of the element; finally we describe such behavior using the diagrams provided by SysML.

Domain descriptions

The refinement starts from the elements of the environment, whose behavior is usually well understood and fairly stable.

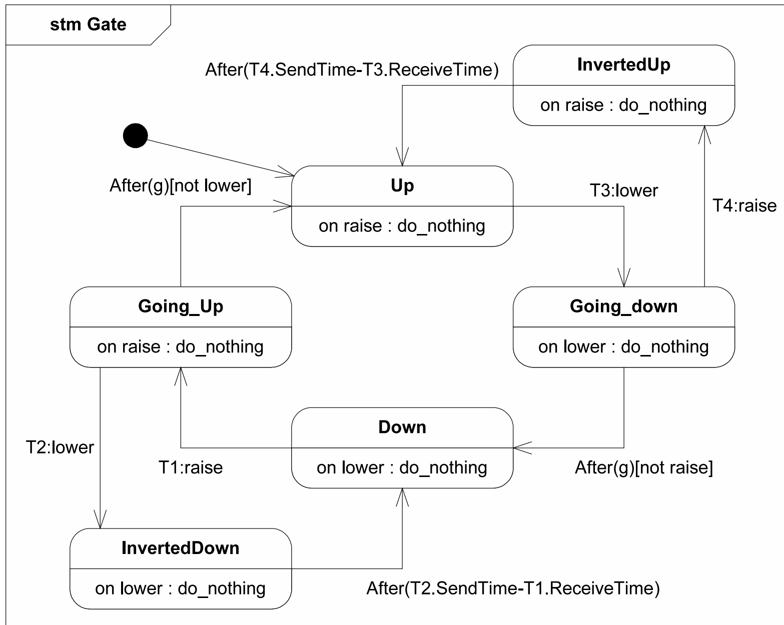

Figure 3.9: State Machine Diagram for the block Gate

Gate The behavior of the gate is described by means of a state machine. The gate block is introduced in the `bdd` of Figure 3.6, and its behavior is informally described in the Requirements Diagram (Figure 3.5) and is refined by the state machine described in Figure 3.9.

The meaning of most of the states and transitions is straightforward. When the gate is closed (i.e., the bar is down), and a `raise` command is received, the bars start to move upwards. If a `lower` signal occurs when the gate is still opening, the bars must start to move down immediately: this is modeled by a transition to state `InvertedDown`. According to the problem definition the bar will reach the closed position after a time equal to the time it has been opening. This behavior is expressed referring to the time when such events occurred.

SysML provides the keywords `ReceiveTime` and `SendTime` to indicate the time when an event was issued and received, respectively. We need to indicate to which of the instances of `raise` or `lower` signal we refer, since we are interested in the `lower` event occurred while the gate was in state `Going_Up`, not to the one

occurred while in Up state. SysML provides a labeling mechanism that can be used to identify individual transition instances. State Machine Diagrams represent transition types, and therefore every transition may occur several times.Thus, in the State Machine Diagram of Figure 3.9, we extended SysML syntax and semantics by applying labels to transitions, so that we can always refer to the last occurrence of a transition. Thus, the transition from InvertedDown to Down occurs after a period equal to the interval between the last raise command and the last lower command.

When the gate is in the state Up, and a lower command is received, the bars start to move downwards. If a raise signal occurs when the gate is still closing, the bars must start to move up. This is described by a transition to state InvertedUp.

The states InvertedUp and InvertedDown require some additional remarks. The state InvertedDown can be reached as a consequence of a lower command sent by the controller whenever a train reaches the critical region while the gate is opening. The state InvertedUp, should never be reached as a consequence of a command sent by the controller. In fact, according to the user requirements, the gate is closing only when a train has just entered the critical region: in this situation the controller should not command the gate to start reopening. The State Machine Diagram for the Controller block (see Figure 3.17) will prevent the gate from entering the state InvertedUp. Therefore, the state machine in Figure 3.9 must be considered the description of the generic behavior of a gate as part of the environment.[3]

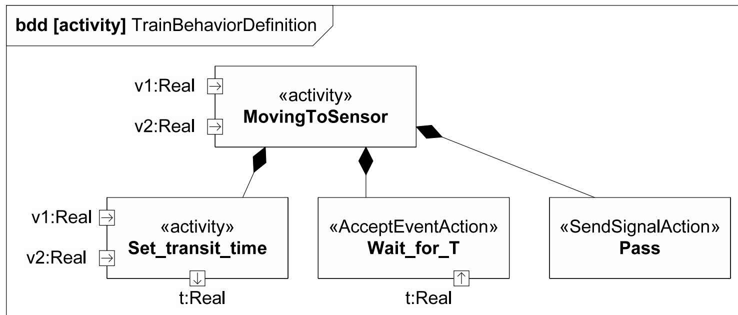

Figure 3.10: Activities allocated to train defined by means of a Block Definition Diagram

Train&Sensors The modeling activity goes on specifying the behavior of trains. Notice that we are interested in the abstract view of trains provided by sensors that allows the controller to maintain a model of the states of the trains.

The behavior of the trains is described by means of Block Definition, Activity and State Machine Diagrams. Figure 3.10 shows a bdd that defines the activities

[3]Notice that, for the sake of simplicity, the proposed state machine does not support double inversion mechanisms.

allocated to the `Train` block, while Figure 3.13 introduces an `act` diagram that specifies the control flow among the basic activities that determine the behavior of trains.

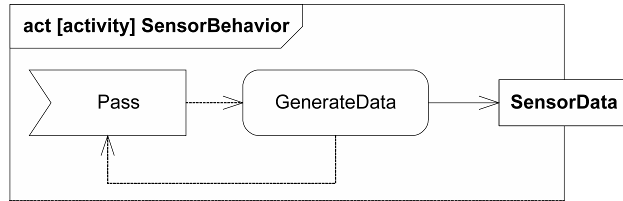

Figure 3.11: The Activity Diagram that defines the basic behavior of sensors

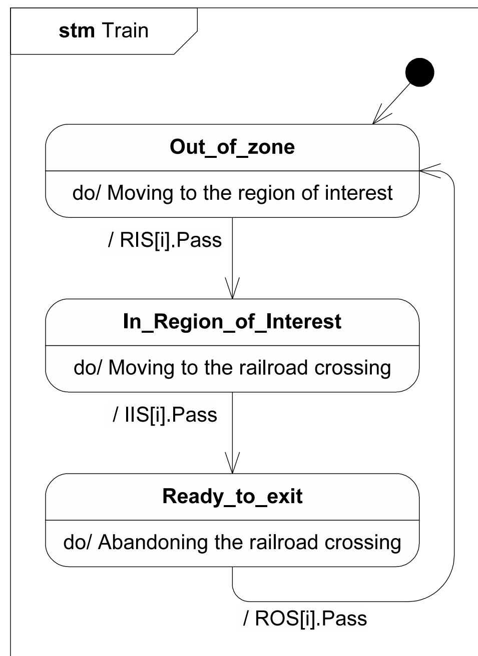

Figure 3.12: The State Machine Diagram that models the basic behavior of the train

The trains simply move through the region of interest and the crossing zone; when a train moves from a zone to another, a `Pass` signal is generated. All the movements between adjacent regions are modeled by means of the `MovingTo-Sensor` activity shown in Figure 3.13 and Figure 3.14. `MovingToSensor` specifies that the time to cross a region is bounded by two predefined values.

Whenever `MovingToSensor` is invoked, and `T` time instants passed since the invocation, then the train has reached the end of the region and a `Pass` event

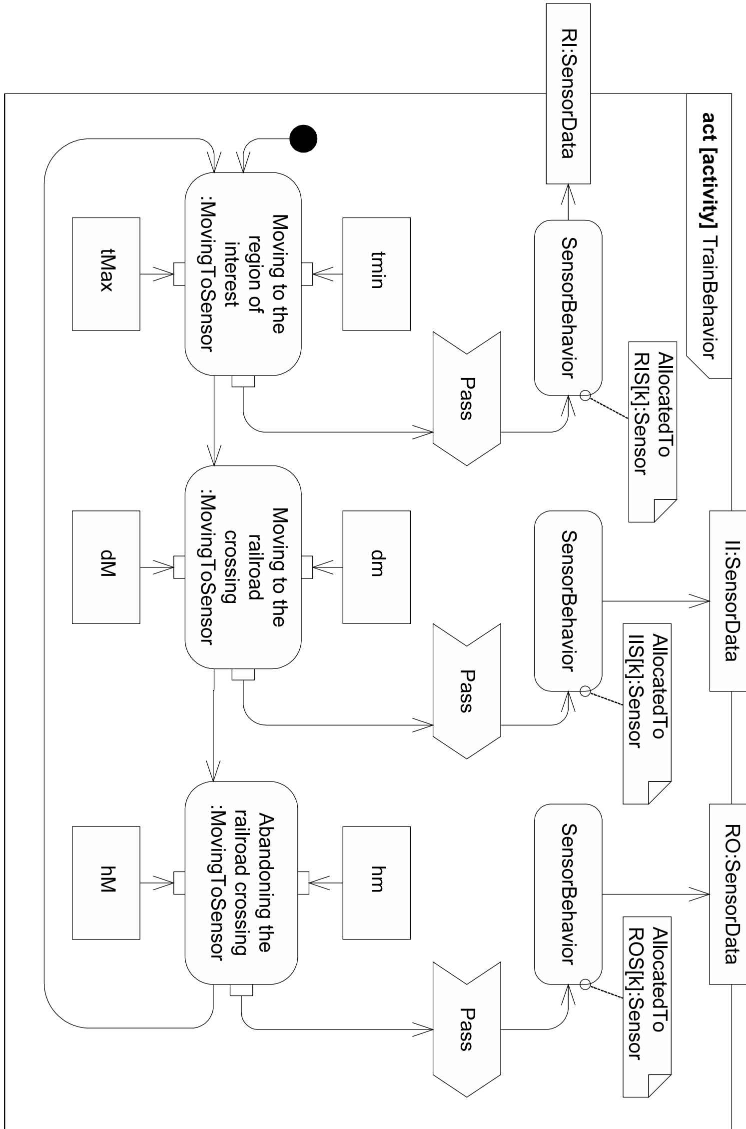

Figure 3.13: The Activity diagram that describes the behavior of trains

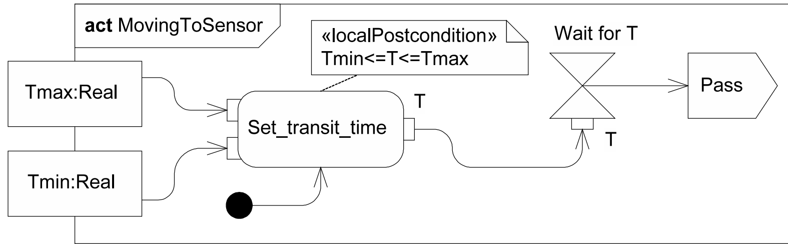

Figure 3.14: The Activity diagrams that describes the action `MovingToSensor`

is generated. Sensors react to such events by generating `SensorData`, which are made available to the rest of the system. Sensors behavior is described by the `act` diagram shown in Figure 3.11. `SensorData` models the shared phenomena occurring between the environment (the trains) and the machine (i.e., the controller).

The definition of the behavior of the trains is completed by the State Machine Diagram shown in Figure 3.12. A train is characterized by three states: at the beginning it is out the region of interest, then it reaches the region of interest and after crossing the critical zone it leaves the railroad crossing. The state machine shows the association of the current state with the activities that it is performing. Whenever a train reaches the `Out_of_zone` state, the `Moving to the region of interest` activity is executed. The activity terminates causing the generation of a `Pass` signal that will be captured and handled by the sensors along the track where the train is traveling.

The state machine shown in Figure 3.12 allows one to simulate the behavior of trains along the tracks of the railroad crossing.The execution of these machines causes the generation of `Pass` signals that once captured by sensors are transformed into `SensorData` signals, which in turn can trigger specific internal behaviors of the controller, like counting the number of trains in each zone of the region of interest, or monitoring the current state of the tracks.

Machine specification

Machine specification starts with the identification of the properties defined in the Requirements Diagram of Figure 3.5 directly associated with the controller.

The Requirements Diagram provides the first indications concerning the behavior of the machine (it receives data from the sensors and sends commands to the gate). Then the behavioral descriptions are refined by Block Definition, State Machine and Activity Diagrams.

The `bdd` shown in Figure3.15 introduces the basic activities allocated to the `Controller` block, while the `act` diagram of Figure 3.16 specifies the control flow that determines the controller behavior.

The Activity Diagram provides an explicit representation of the interactions of the machine with the surrounding environment. The squared boxes on the border of the diagram shown in Figure 3.16 represent incoming data from the sensors flow ports `ctrlIISP`, `ctrlRISP` and `ctrlROSP` (see the Internal Block Diagram in

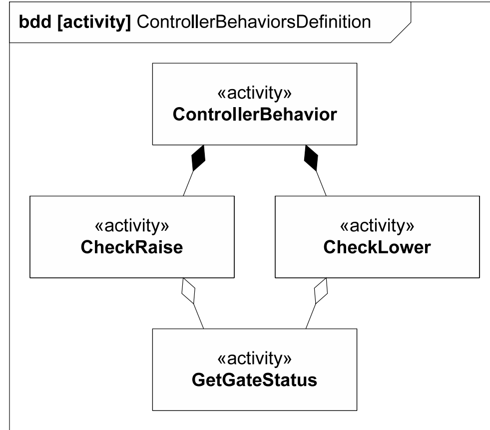

Figure 3.15: Activities concurring the definition of the controller's behavior

Figure 3.7). Notice that each port is connected to K sensors (one sensor for each track). This property derives from the structural characteristics of the problem domain, as described in the `ibd` of Figure 3.7 and in the Requirements Diagram (Figure 3.5).

The effect of the signals from sensors on the state of the controller is specified by the activities `DecCIR`, `DecCCR`, which decrement counters `CIR` and `CRR` respectively, and by activities `IncCIR` and `IncCCR`, which increment them. Counters `CIR` and `CCR` (which are part of the state of the Controller, see Figure 3.6) represent the number of trains in the critical region and in the crossing region, respectively. Therefore, the values of these attributes have to change only when trains enter or exit the zones of the region of interest.

Notice that exiting the region of interest corresponds to exiting the critical region and therefore both events are detected by sensor `ROS`. Conversely, while the entrance in the region of interest is detected by sensor `RIS`, no sensor is positioned at the beginning of the critical region. As a consequence, the controller assumes that a train is entering the critical zone `(dm-g)` seconds after it entered in the interest region. This is expressed by specifying that the receipt of signal `RI` from sensor `RIS` causes the execution of action `Arriving` that waits for `(dm-g)` seconds before activating `IncCCR`[4]. In a similar way when signal `RO` is received from sensor `ROS`, action `Leaving` is triggered and `DecCIR` is immediately executed.

Notice that the State Machine Diagram of Figure 3.17 refers to events `Arriving` and `Leaving`[5] in order to synchronize the behavior specified by the Activity Diagram with the behavior specified by the state machine.

According to the `UserRequirements` reported in the Requirements Diagram, the controller has to react properly to external events, in order to operate the gate

[4]Notice that transmission latencies and reaction times of the components are not taken into account for the sake of simplicity

[5]According to SysML semantics the execution of a send signal action such as `Arriving` or `Leaving` generates an event having the same name

Figure 3.16: The Activity Diagram that describes the controller's behavior

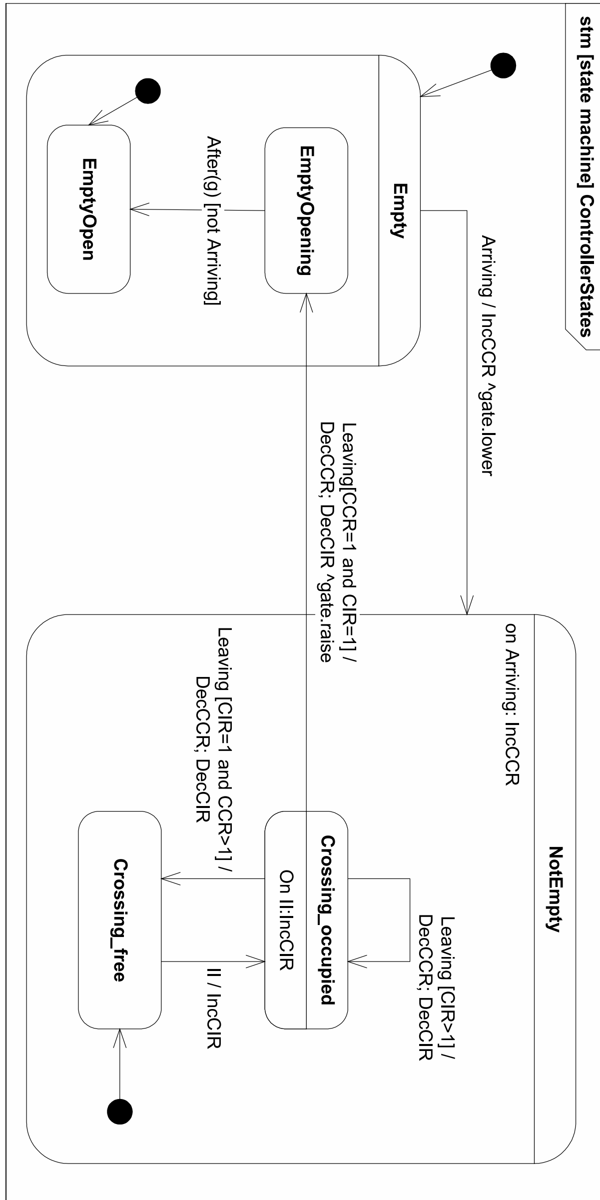

Figure 3.17: The State Machine Diagram for the Controller block

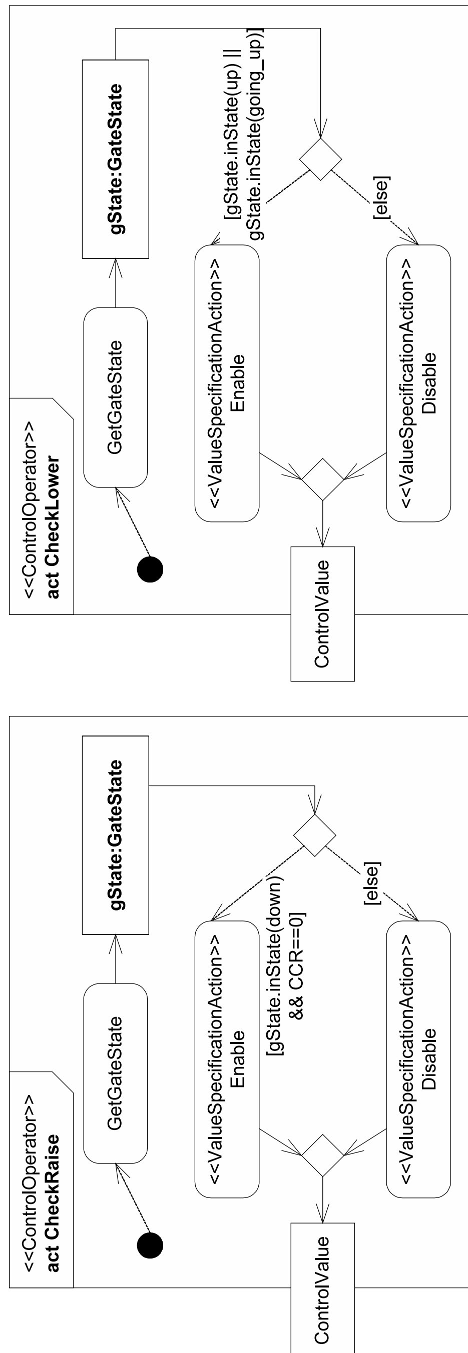

Figure 3.18: CheckRaise and CheckLower activities behaviors

according to the position of trains. This is specified in the Activity Diagram in Figure 3.16 by indicating that data from sensors `ROS` and `RIS` triggers the execution of the activities `CheckRaise` and `CheckLower` respectively. These activities determine whether the gate has to be lowered or raised, according to the values of `CIR` and `CCR`. Actually, the Activity Diagram specifies that two control cycles are performed: 1) the updating of `CCR` is followed by the decision whether to lower (raise) the gate bar, 2) the lower (raise) command is issued, if it is the case.

Since the decision whether to raise or lower the gate bar is crucial, we need to specify the decision activities as clearly as possible. For this purpose we describe `CheckRaise` and `CheckLower` activities by means of the dedicated Activity Diagrams reported in Figure 3.18, where inner activities and flows are described. For instance, from the detail of the `CheckRaise` activity it is possible to see that the raise control value is enabled when the gate bar is down and `CCR` is equal to zero.

Notice that the controller interacts with the other components of the system in order to satisfy the user requirements.

Initially the critical region is empty and the gate is open. Whenever a train enters the critical zone the counter `CCR` is incremented. Whenever a train exits the critical zone the counter `CCR` is decremented. If a train arrives while the critical zone is empty the state is changed to `non-empty`, and a `lower` command is sent to the gate. When the last train leaves the critical zone, the state is changed to `empty`, and a `raise` command is sent to the gate.

Synchronizing local models

According to the previously introduced modeling strategies, the controller exploits local models to keep track of the current state of the tracks.

More specifically, the state is modeled by means of a State Machine Diagram.

Figure 3.19 reports the State Machine Diagram of `TrackState`. It specifies that a train traveling along the track is initially out of the region of interest, then –coherently with the Activity Diagram of Figure 3.13– it passes sensor RIS and enters region R. The involved sensor sends the signal `RI`, which is an instance of the block `SensorData` with attributes `sensorID` set to `RIS` and `trackID` set to the track number on which the sensor is located. All the instances of `TrackState` receive `RI`, but only the instance having attribute `trackID` equal to `RI.trackID` reacts, setting the state to `Train_in_SafeZone`. `dm-g` seconds after entering the safe zone, the train enters the critical zone: this happens only for the fastest trains, but it is safe to consider that all the trains reach point X `dm-g` seconds after entering the safe zone. This transition is modeled by means of the `after` statement provided by UML [57] and inherited by SysML.

The next step consists in specifying that the train enters region I not earlier than `g` seconds nor later than `dM-(dm-g)` seconds after entering the critical zone. Like UML, SysML does not provide any means to specify that a transition is bound to occur in a given time range. In order to express such constraints we introduce the state `Train_Close_to_Crossing` and the state `Error`. Note that these states are introduced to address the limits of the language. For instance, `Error` indicates that the behavior of the system deviated from the specification. In the design phase the `Error` state will be replaced with some exception handling

state, for the sake of robustness.

In case a train reaches point II exactly after it has been in the critical zone for `g` seconds, it directly goes into the `Train_Crossing` state, without passing through the state `Train_Close_to_Crossing`. Notice that in order to express this property it is necessary to extend the SysML syntax of conditions. In particular, we need to specify the occurrence of an event at a given time while plain SysML does not allow conditions to reference events.

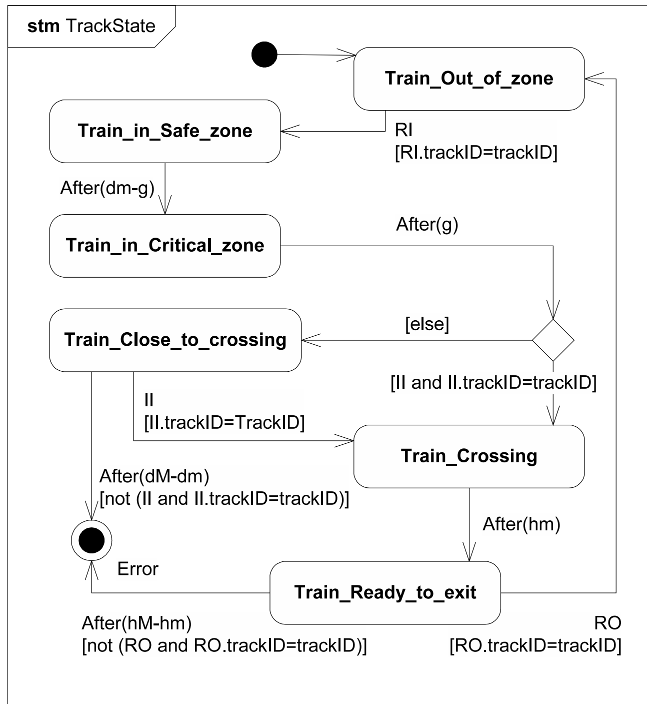

Figure 3.19: State Machine Diagram for the TrackState block

Refining behavioral descriptions

Taking advantage of SysML capabilities, advanced behavioral properties associated with `Controller` are defined by means of a `bdd` comprising `Constraint` blocks. Such blocks are allocated to model elements using `par` diagrams.

Figure 3.20 reports the Block Definition Diagram where the invariant properties of the Controller block are defined using constraint blocks, and using TRIO as specification language.

The controller is required to act according to the number of trains in the various regions. The attributes `CCR` and `CIR` of the `Controller` block defined in Figure 3.6 represent the number of trains in the critical and in the crossing zones, respectively. More precisely, these attributes represent the knowledge of

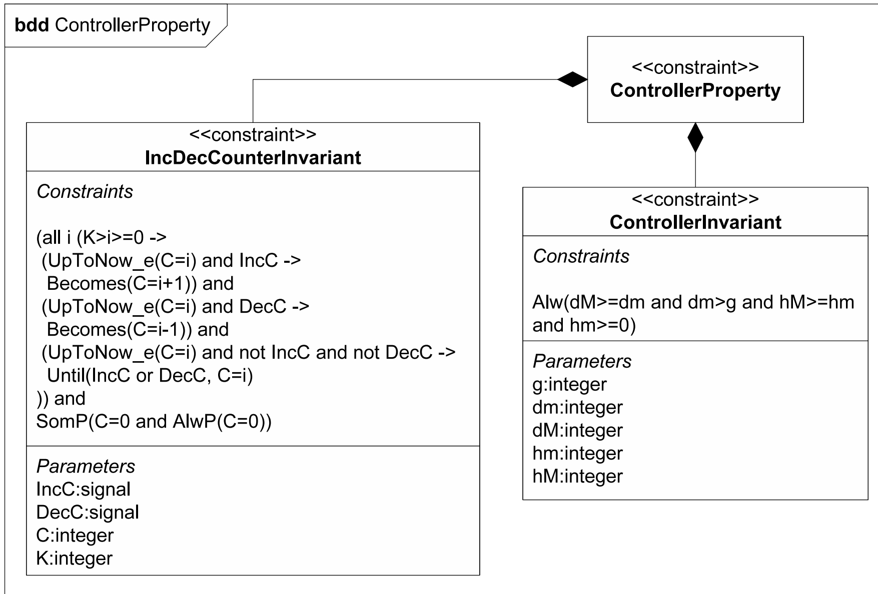

Figure 3.20: Definition of invariant properties for the Controller block

the controller about the number of trains in the mentioned zones. It is necessary to specify how this knowledge is kept coherent with the real situation of the tracks. For this purpose the constraint `IncDecCounterInvariant` is introduced (Figure 3.20). The property `IncDecCounterInvariant` states that the `C` counter – initially zero – is incremented whenever an `IncC` event occurs and decremented whenever a `DecC` event occurs, otherwise the value of the counter remains unchanged. The property is expressed in TRIO as follows:

```
UpToNow_e(C=i) and IncC -> Becomes(C=i+1)
UpToNow_e(C=i) and DecC -> Becomes(C=i-1)
UpToNow_e(C=i) and not IncC and not DecC -> Until(IncC or DecC, C=i)
```

The first part of the TRIO clause states that if `C` was equal to `i` immediately before now, then `C` is incremented if an event `IncC` is occurring now, the second part of the TRIO clause states that if `C` was equal to `i` immediately before now, then `C` is decremented if an event `DecC` is occurring now, while (the third part) it remains unchanged if neither `IncC` nor `DecC` is occurring. This applies for every `i` in the (0 .. `K`-1) range (`K` being the number of tracks). Note that the behavior for `C>K`, or for `C=K` and `IncC` is not specified (constraints in Figure 3.20). In fact, specifying these cases is not necessary, since it will never be the case that on N tracks there are more than N trains. Notice that in real cases we should consider the possibility that malfunctioning sensors report the passage of nonexistent trains, thus making `C` greater than `K`. However, modeling faults is out of the scope of the chapter.

Figure 3.21 reports the Parametric Diagram that specifies how the constraints

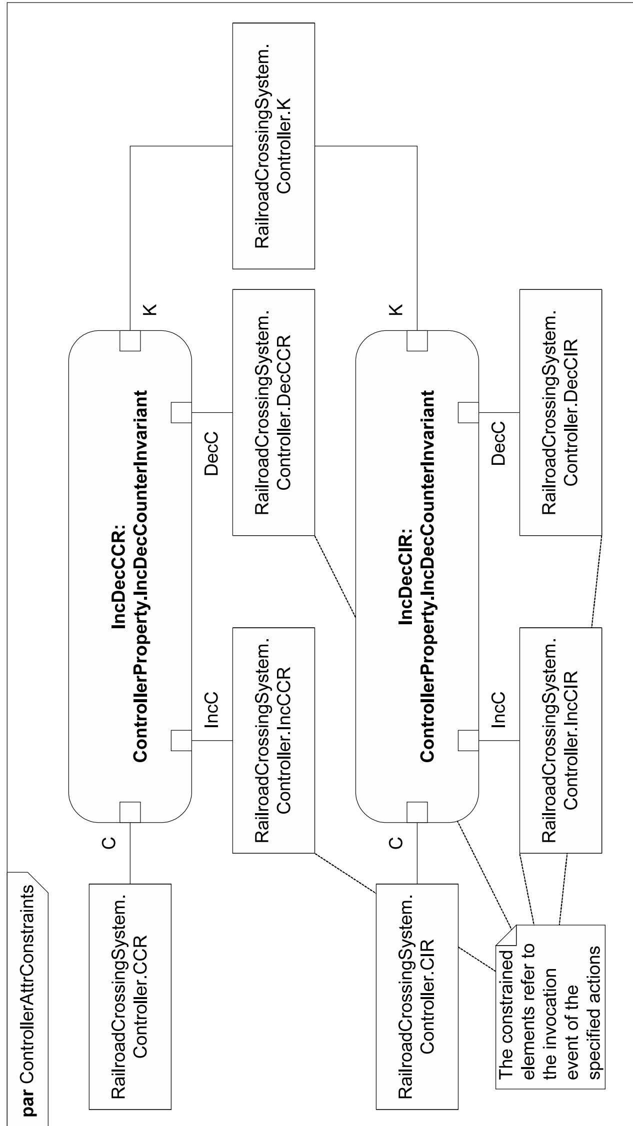

Figure 3.21: The Parametric Diagram which shows the allocation of the properties of the Controller block

defined in Figure 3.20 are used.

3.3.4 Refining User Requirements

The model described so far correctly defines the gate controller but does not formally express the safety property. We need to refine the safety requirement (reported in the Requirements Diagram) so that when there is at least one train in the crossing zone I, the gate is closed.

Also in this case the safety and utility requirements are defined as constraints in the `bdd` reported in Figure 3.22, and their usage is specified in the Parametric Diagram of Figure 3.23.

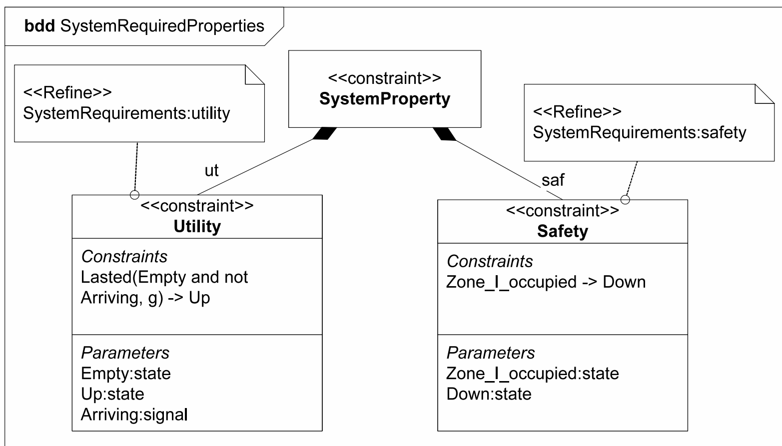

Figure 3.22: The Utility and Safety properties

The safety condition can be written in TRIO in a quite straightforward way:

```
Zone_I_occupied -> Down
```

The Parametric Diagram in Figure 3.23 indicates that `Zone_I_occupied` corresponds to the `NotEmpty::Crossing_occupied` state of the controller, while `Down` refers to the `Down` state of the gate.
Thus, the combination of the specifications reported in Figures 3.22 and 3.23 requires that when the crossing is occupied the gate must be down.

Note that in the Parametric Diagram in Figure 3.23 `Zone_I_occupied` is connected with a state of the representation of the interest region contained in the controller, rather than with the state of the real interest region. This is a sound simplification since we are modeling the GRC under the assumption that everything works well and no fault occurs, there is no difference between reality and its representation in the controller.

The utility condition can be expressed in TRIO as follows:

```
Lasted(Empty and not Arriving, g) -> Open
```

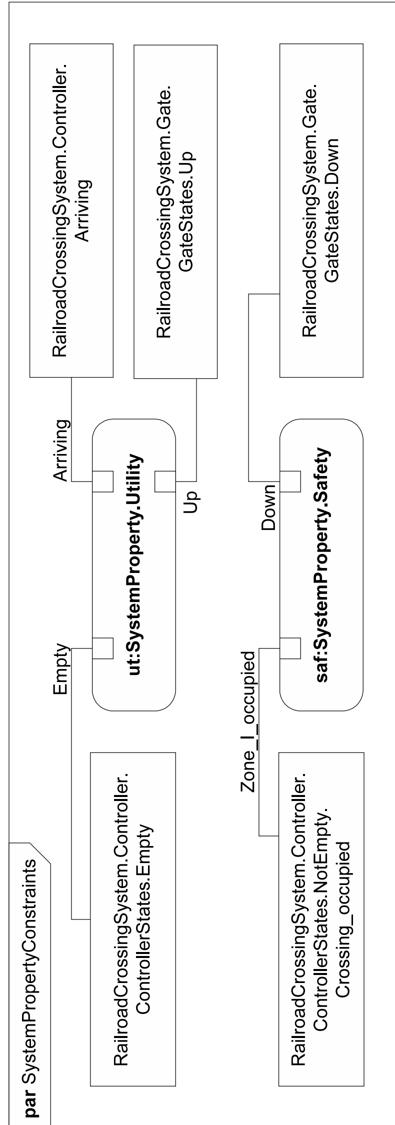

Figure 3.23: Safety and utility properties allocated to the `Controller` states

The TRIO formula states that if for `g` time units the critical zone was empty, the gate must be open. The critical zone is empty if the crossing is empty and no train passed point X (`Arriving`).

As a conclusion, notice that it was quite easy to embed satisfactory formal specifications into SysML diagrams.

3.4 A first assessment on SysML

SysML represents a big step forward in modeling systems; it implements most of the requirements contained in the original request of proposals submitted to OMG in 2003 (see [64]).

In our opinion, SysML is better than UML 2 for modeling embedded and real-time systems. As an example, let us consider a type of system that usually has both real-time and embedded features, such as control systems. The communications among the components of such systems usually require continuous flows of information. UML does not allow modeling continuous data flows. On the contrary, SysML provides constructs to describe streams of data and continuous activities, which can be used to adequately model continuous systems [5].

SysML provides the extension mechanisms inherited from UML: tagged values, stereotypes and profiles. According to UML 2 specifications [57], a profile defines limited extensions to a reference meta-model with the purpose of adapting it to a specific platform or domain. In order to achieve this purpose a profile defines constraints on the meta-classes of the meta-model that it aims at extending. The extensions are defined by means of stereotypes built around the original meta-classes. This mechanism is quite limited, since it does not allow a modeler to redefine the semantics of the elements described by means of meta-classes of the original meta-model. As an example, the mechanisms provided by UML are not powerful enough to change the semantics of the transitions in the State Machine Diagrams as required in [19] or the semantics of the interactions in the sequence diagrams as advocated in [20]. In both cases changes must be defined by directly manipulating the UML meta-model.

We did not use a profile (e.g., a temporal profile) to model the GRC; instead, we used the basic profile of SysML. There are two main reasons for this choice: one is that at the moment no timing profiles are available for SysML, the second is that we were interested in studying the basic capabilities of SysML and how the modeling process could take advantage of the usage of a formal notation.

Focusing the attention on the usage of the standard constructs and diagrams we found the inadequacy of the State Machine Diagrams to model strict timing information. In fact, the run-to-completion semantics inherited from UML provides state machines with a buffer to handle the incoming events; every event is dealt with when the machine has finished consuming the previous one. The consumption of an event can trigger a transition that executes in a non null time. While this behavior is acceptable in the implementation of a system, it is unsuited to model the time requirements and to carry out the analysis of a system. For this purpose instantaneous and possibly simultaneous transitions are needed, as well as the ability to deal with time intervals. Timed Statecharts [40] are suitable to express precise time constraints (see [17], [18] and [19], where the problem

is specifically addressed). Since the syntax and the graphical aspect of Timed Statecharts are very similar to those of SysML state machines, the State Machine Diagrams presented in the previous sections could be interpreted according to the semantics of Timed Statecharts: in this case we could assert that the model represents correctly the time behavior of the system.

However, we must consider that SysML supports both the operational modeling style (e.g., via State Machine Diagrams) and the declarative style (via Constraints). SysML does not feature a proper temporal logic (or analogous formalism) to express time constraints. Although SysML does not prevent the use of any formal language to express constraints, the availability of a standard formal language would be useful, especially considering that UML constraint language OCL does not support the specification of timing issues, and it lacks many of the needed features (see [44] for an OCL proposal of extension with time related features). In practice, modelers have to employ a language like TRIO [23] to represent time constraints. Currently no UML profile fully satisfies the requirements posed by real-time systems modeling. Moreover SysML lacks constructs (either built in or made available by profiles) to define real-time systems; nevertheless this limit can be easily overcome (as done in the previous sections) by introducing the usage of an external (possibly formal) language. Although this solution could cause practical problems (e.g., the SysML tools may not understand constraints written in the "foreign" language), it is fully compliant with the definition of SysML and can be applied for the definition of system properties by means of parametric constraints.

A final remark is that resource management and consumption, which are very important features when dealing with embedded systems, are not directly addressed. On the positive side, the buffered behavior of each activity in Activity Diagrams can be disabled, in order to model activities that do not buffer incoming data. In SysML it is even possible to constrain the rate of incoming or outgoing data that activities can accept (thus preventing buffering). These features are relevant for embedded systems since buffers can consume large amounts of resources.

3.5 Related work

SysML is a recent notation for software and system modeling (the final specification was released in May 2006 [53]). Hardly any methodological guide concerning how to use the language has been proposed yet. The literature reports just a few experiences in using SysML. For instance, some whitepapers are available from Artisan, describing systems like a house heating system [48] or a waste treatment plant [28]. However, these papers are mainly meant to describe the nature and the role of diagrams, rather than suggesting methodological guidelines.

We proposed some preliminary methodological guidelines, involving the integration of concepts from the Problem Frames approach into the modeling processes based on SysML. The work reported in [13] addresses the usage of problem frames in a SysML based modeling environment: on one hand SysML is used as the notation to represent problem frames, on the other hand methodological issues from the problem frame approach are incorporated into the SysML modeling

process.

Concerning the specification of HW-SW systems, there are several proposals that —although not related to SysML— cast some light on the requirements specification activity. A compendium of the system engineering practices can be found in [31].

As far as the ability to model time-dependent issues is concerned, it could be possible to exploit SysML extension mechanisms to define specific constructs for the real-time domain; however, no such profile has been proposed for standardization yet. Since SysML is a UML Profile, we expect that the System engineering community will adapt to SysML some of the profiles originally proposed for UML. In the rest of this section we shortly describe the most important ones.

The OMG standard UML profile for Schedulability, Performance and Time Specification (SPT) [51] takes advantage of a complex time annotation mechanism that enables different kinds of analysis. After annotating a model, analysts can carry out quantitative analysis applying Rate Monotonic Analysis techniques or techniques based on classical queuing theory [25]. In SPT *time* is a quantifiable resource, moreover the profile introduces concepts like *instant* and *duration*, and also constructs to make reference to physical time such as *timers* and *clocks*. SPT is designed for UML 1.4, thus it cannot be directly imported and used in the UML 2 based SysML.

The UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (QoS&FT) [50] is another OMG standard profile that, like SPT, supports the specification of non functional properties with advanced annotation mechanisms. Several types of annotations can be defined and used to deal with different problem domains. Apparently, its generic nature makes QoS&FT an ideal solution for different types of systems, but unfortunately it has some deficiencies. For instance, QoS&FT does not allow defining variables in complex real-time expressions. Moreover, the annotation definition process is cumbersome and requires the creation of components outside the current model.

Other profiles try to overcome the limits of SPT and QoS&FT. Among these, the OMEGA profile [24] aims at defining a rigorous semantics for the constructs used to specify complex temporal properties. However, the OMEGA profile is not a standard.

OMG and the ProMarte team are currently working at the definition of the UML Profile for Modeling and Analysis of Real-time and Embedded systems (MARTE) [55]. MARTE promises to fully support the model-driven development of real-time and embedded systems. It defines the foundations for model-based descriptions, providing the support required from the specification to the detailed design phase. MARTE introduces constructs to annotate models with the information required to perform performance and schedulability analysis. A stable standardized version of MARTE is not yet available. Since SysML and MARTE share common goals, it is expected that OMG will release guidelines for the usage of MARTE in SysML models.

Besides profiles, there are other mechanisms for redefining the semantics of particular elements of the meta-model using formal methods.

Languages like OCL/RT[7, 21, 62] and OTL[44] are used to specify properties of (sets of) elements of UML diagrams. Constraints and properties are expressed

by means of a temporal logic and are associated with elements of UML models as annotations. Another technique is based on ArchiTRIO [15, 60, 61], a formal language based on high order logic that integrates Object Oriented and UML-based concepts. The language supports the definition of both structural and behavioral properties by means of axioms and theorems. As a result we get a UML model composed of elements whose semantics is formally defined by the logical language. Other techniques feature an operational style. As an example, in [20] High Level Petri Nets are used to formally define the semantics of scenarios via Timed Petri Nets, while in [17] the run to completion semantics of the state machines is substituted with the Timed Automata semantics in order to deal with time dependent behaviors.

Being formal, all these notations support different types of verification methods. For instance, OCL/RT and OTL allow an analyst to apply lightweight verification techniques based on model checking; ArchiTRIO allows the analyst to apply theorem proving techniques taking advantage of translations in PVS, while classical reachability methodologies can be carried out for High Level Petri Nets. Although such methodologies support the definition of expressive and verifiable models, they do not comply with the UML standard, depending on *ad hoc* non-UML specification or verification tools.

As a system oriented language, SysML should prove effective in modeling embedded system. During the last years, dedicated UML profiles have been defined and used as system design languages for the definition of Systems on Chips (SoC) [46, 67, 68]. These profiles provide a high level system abstraction layer to the design process of SoC systems. The modeling process takes advantage of the synergies between UML and dedicated SoC modeling languages like SystemC [24] or VHDL [59]. The same components defined at system level are extended and refined at a lower level. In addition, models enriched with dedicated constructs allow the system engineer to automatically obtain code skeletons. In the literature it is easy to find several papers describing the relationships between UML and SystemC [6, 49, 63, 66], and between UML and VHDL [3, 47].

SysML appears as a natural evolution of the aforementioned UML-based notations and techniques. In fact , SysML integrates most of the constructs previously defined by the afore mentioned profiles. It natively integrates basic concepts like blocks (the HW/SW components in the embedded context). It defines flow ports that describe the interaction points among the components, and depict HW/SW signal flows. It provides an easy mechanism for integrating formal notations. It provides cross cutting constructs (like the requirements) that allow the modeler to apply a full model-centric approach to the definition of a system.

As a consequence, the SoC design process could take advantage of the usage of SysML.

Chapter 4

Problem Frames

Traditionally, software development has focused on mechanisms and abstractions that are useful in designing and implementing software applications. Only little attention was paid to the problems that such programs are intended to solve. Hence, the software development can be defined traditionally solution oriented.

Solution oriented approaches are suited to be applied to a small set of problems, i.e., only to those that were previously analysed, described and classified. Such approaches focus on finding innovative solutions and possibly on improving previously proposed ones.

Instead, Problem analysis approaches like the one introduced by Jackson in [37], focus on investigating and describing the context of problems, analysing their internal characteristics, identifying the concerns and the difficulties that need to be addressed when trying to solve those.

For Jackson [37]:

> *Problem analysis takes you from the level of identifying the problem to the level of making the descriptions needed to solve it.*

Although such explanation introduces the goal of the approach, it must not be considered as a definition of an approach. In fact, real problems are complex and cannot be handled directly by means of a single step. Complex problems need to be decomposed into subproblems that are simpler to solve so that the resulting subproblems can be faced separately. Finally, a general description of the whole problem is achieved by composing the single small problems.

Problem analysis approaches are essentially composed of two distinct activities: the former, named context analysis, focuses on the definition of the characteristics of the problem to be solved, while the latter, named structural analysis, focuses on the decomposition of the original problem into simpler subproblems.

The rest of the chapter shortly introduces a problem analysis and structuring approach named Problem Frames (PFs) proposed by Jackson in [37].

The whole approach is based on the concept of *Problem frame*:

> *A problem frame defines the shape of a problem by capturing the characteristics and interconnections of the parts of the world it is concerned with, and the concerns and difficulties that are likely to arise.*

PFs analyse complex realistic problems by decomposing them into subproblems that correspond to known problem frames. Problem frames are at the basis of both context and structural problem analysis and aim at looking outwards to the world where the problem is found, hence, hindering the usage of solution oriented processes.

4.1 The problem and the world

According to the problem frames approach, a software development problem is "a need for a useful machine", while the solution of the problem is "the construction of a suitable machine to satisfy the need" [32]. Such a need consists in the environment of the useful machine described in terms of what is given and what is desired. [16].

Requirements represent what is desired, i.e., a set of properties that are not directly owned by the world but are desired by the user. On the contrary, given properties are a set of intrinsic characteristics of the world not depending on the behavior of the machine.

According to the PFs approach, a first analysis of the problem is operated by identifying the basic constituent parts of the world where the problem is located, named *problem domains*, and the target of the development process, named *machine domain*[1].

The machine is phisically represented by HW and SW components, while problem domains are physical components of the world where the problem is located and are affected by the behavior of the machine.

Domains (both problem domains and machine domains) consist of phenomena, i.e., intrinsic characteristic of the world where the problem is located, and communicate by means of interfaces. Interfaces can be considered as a place where domains partially overlap, so that the phenomena in the interface are shared phenomena that allow connection and communication between distinct domains. Generally one domain controls a set of shared phenomena while other domains have visibility of those phenomena.

An interface that connects a problem domain to the machine is called a *specification interface*. The goal of the requirements analyst is to develop a specification of the behavior that the machine must exhibit at its interface in order to satisfy the user requirements.

The basic philosophy of PFs is based on the concept that requirements are about relationships in the real world, not about functions that the software system must perform. Rather, the desired relationships in the real world are achieved with the help of the machine connected to problem domains.

However, in the requirements analysis phase, the machine is only specified as far as its role in the real world is concerned. For this purpose, only the interface between the machine and the problem domain needs to be specified, while the machine internals are left unspecified (they will be addressed in the design phase).

[1]For the sake of simplicity, in the rest of the description *machine domain* is simply referred to as *machine*

4.1.1 Phenomena

Phenomena may represent basic constituent elements of domains, elements that may affect the behavior of domains, or internal properties concerning both structural and behavioral aspects of domains. Jackson proposes six types of basic phenomena named *Event*, *Entity*, *Value*, *State*, *Truth* and *Role*. Depending on their characteristics, such phenomena are organized into two distinct categories: *individuals*, which represent stand alone primitive elements, and *relations*, which represent relations between individuals.

- *Events.* An event is an indivisible and instantaneous individual happening, that occurs at some point in time. Events are assumed not to occur simultaneously. Hence, an order relation can be defined among the events occurring in a problem.
- *Entities.* An entity is an individual whose properties may change over time. Entities in turn may affect other types of phenomena. For instance entities may generate events.
- *Values.* A value is an individual that exists outside time and cannot change.
- *States.* A state is a relation between an entity and a value. Such relation may change over time.
- *Truths.* A truth is a relation between individuals that cannot change over time.
- *Roles.* A role is a relationship between an event and an individual that participates in defining or listening to it.

Phenomena can be organized also depending on the role they play with respect to the domains which they affect or by which they are controlled.

- *Causal.* Causal phenomena are phenomena that are caused or controlled by domains, and that in turn may cause other phenomena. They include events, roles, and states relating entities.
- *Symbolic.* Symbolic phenomena symbolize other phenomena and the relationships among them. They include values, truths and states.

4.1.2 Domains

Domains are the basic structural elements of a problem. Depending on their role and their nature, domains can be classified into the following classes: machine, designed, given.

- *Machine.* A machine domain is the HW component that provides the runtime execution environment for the software application that will represent the solution of the problem.
- *Designed.* A designed problem domain is the physical representation of an entity that need to be designed. The modeler is free to specify the properties of such domains.

- *Given.* A given problem domain is the physical representation of an entity whose properties are given and cannot be changed.

Domains, regardless of their role and nature, can be thought of as aggregation of phenomena.

Depending on their characteristics, problem domains can be classified into causal, biddable and lexical domains.

- *Causal.* Causal domains are the ones characterized by properties that include predictable causal relationships among their internal causal phenomena. Causal domains may control or be affected by the shared phenomena at the interface with other domains. Notice that in case a causal domain is connected to a machine domain, the causal relationships among the internal causal phenomena of the domain allow one to evaluate the effect of the behavior of the machine at the interface with the domain.

- *Biddable.* Biddable domains are the ones characterized by the lack of predictable causal relationships among its phenomena. They represent entities whose behavior cannot be forced by any phenomena. As an example, biddable domains often represent people, since their behavior cannot be predicted, and they cannot be obliged to perform any actions.

- *Lexical.* Lexical domains are the ones composed of symbolic phenomena representing data. Lexical domains may be affected by causal phenomena that trigger the access to the owned symbolic phenomena. As a consequence, lexical domains can be thought of as a special type of causal domains. As an example consider a data repository. In such case, the repository can be represented by means of a lexical domain, while data by means of symbolic phenomena. The access to data is regulated by read/write commands, i.e., causal phenomena controlled by external domains.

4.1.3 Interfaces

Domains interact by means of interfaces and shared phenomena. Interfaces can be viewed of as a collection of phenomena shared among two or more domains. Shared phenomena, in turn, represent phenomena that are controlled by one of the domains involved by the interface, and are observed by the one or more domains.

Notice that interfaces are abstractions on the actual form of interaction between two domains. An interface does not specify the communication direction, which can be inferred by identifying the controller and observer domains of the shared phenomena. Moreover, the communication through an interface can be bidirectional, i.e., given two domains A and B connected by means of an interface composed of several phenomena, there can be phenomena controlled by A and observed by B and viceversa.

4.2 Descriptions

The problem analysis process at the basis of the Problem Frames approach is essentially based on the description of different aspects of a problem.

Problem Frames provides three basic types of descriptions:

- the description of the properties of the problem domains
- the specification of the requirements of the problem
- the specification of the behavior at the interface of the machine

4.2.1 Domain properties

Problem domains are characterized by structural and behavioral properties that express existing relations among their internal phenomena. The description of such properties are said *indicative*, since they indicate the objective true about the domains, that is what is truth regardless of the behavior of the machine.

4.2.2 Requirements

Requirements are properties that are not directly owned by the problem domain, and they represent the wishes of the user with respect to the solution of the problem.

Jackson states that requirements are *optative* description, in the sense that they describe the options that the user has chosen. The aim of the PFs approach is to show how the machine domain, once connected to the problem domains, is able to satisfy the requirements.

Requirements are properties that refer to internal characteristics of the problem domains. More specifically, Requirements predicate on the internal phenomena of the problem domains. Requirements may refer to phenomena that are not shared among domains; moreover, they cannot directly refer to the phenomena of the machine.

4.2.3 Machine specification

The machine specification is a collection of properties that describes the desired behavior of the machine at the interface with the problem domains. It is a description on the shared phenomena at the interface, consistent with the properties of the world and satisfiable by appropriate actions of the machine.

Similarly to requirements, also the machine specification is an *optative* description.

4.3 Notational support

The Problem Frame approach is supported by two types of diagrams named Contex diagram and Problem diagram. The former describes the context in which the problem is set, while the latter includes also the relationships between the domains and the requirements of the problem, respectively.

4.3.1 Context diagrams

The context diagram shows the various problem domains in the application domain, their connections, and the machine and its connections to the problem domains.

Figure 4.1: Domain types

Domains are represented by means of a notation that explicitely expresses their role and nature (see Figure 4.1). For a given problem, a context diagram explicitely defines which domain is the machine domain, and among the remaining problem domains which ones are given and which ones are designed. Interfaces are represented by means of connections among two or more domains. Notice that such connections are labelled, and the meaning of each label is specified in a legend that defines the shared phenomena involved in the definition.

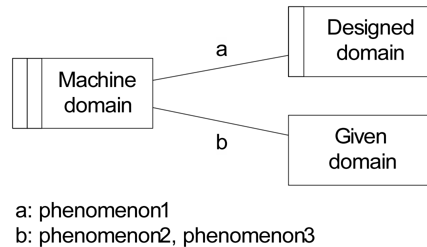

Figure 4.2: A simple Context diagram

Figure 4.2 reports an example of a Context diagram.

Notice that context diagrams simply describe the structural characteristics of the world where the problem is located. They do not express anything neither about the behavioral aspects of domains nor about the requirements.

4.3.2 Problem diagrams

A Problem diagram can be thought of as an extended Context diagram.

With respect to Context diagrams, the most important enhancement refers to the notational support for representing requirements. Requirements are represented by means of a textual description contained into a dotted oval. A dotted connection between the requirement oval and the single domains is defined between the requirement oval and the domains involved by the requirement definition. Requirements predicate on internal phenomena of problem domains. Such phenomena may be simply referred to (indicative description) or constrained (optative description) by requirements. Constraints are represented by means of dashed arrows that connect the oval to the interested domain, while references are represented by means of a dashed line between the oval and the domains.

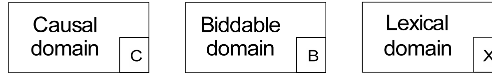

Figure 4.3: The notation to express the role of a domain

The notational elements used to represent the nature and the role of the domains are the same provided by Context diagrams, but in addition the user can also indicate behavioral aspects by means of dedicated elements that represent Causal, Biddable and Given domains (see Figure 4.3). Nevertheless, notice that this notation is not mandatory, and is used by Jackson only to illustrate the role of domains in the definition of basic Problem Frames.

Domain interfaces and shared phenomena are represented by means of connections and a legend. Interfaces are represented by labelled connections between domains, while the legend, which extends the one of Context diagrams, provides information that indicate, for each shared phenomena, which domain controls it.

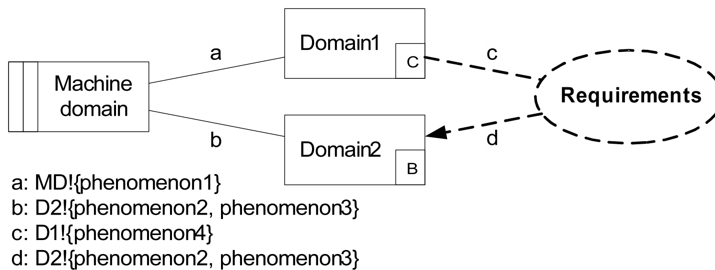

Figure 4.4: A simple Problem diagram

Figure 4.4 reports a simple example of Problem diagram. In order to better clarify the usage of requirements references and constraints, consider that a requirement property of the problem described in Figure 4.4 may indicate that *phenomenon3*, controlled by *Domain2*, is characterized by a value that depends on (optative description) the value of *phenomenon4*, which is controlled by *Domain1*. Notice that, by means of the machine, such requirement defines a dependency between the phenomena of distinct domains.

Problem diagrams represent a starting description for problem analysis.

4.4 Problem frames

A Problem Frame is a description of a recognizable class of problems. Problem Frames can be viewed as problem patterns, since they propose identifiable problem classes by expressing the characteristics of their domains, interfaces and requirements.

Jackson considers that real problems can be decomposed into subproblems that belong to the same problem frame classes. In case of a successfull decomposition, the resulting subproblems are much simpler than the starting problem. As a

consequence it is necessary to identify recurrent problem patterns.

A problem pattern defines:

- the decomposition of the problem world into a set of domains interacting among them.
- the characterization of the problem domains according to their physical behavioral properties.
- the characterization of the domains interfaces according to the types of phenomena that are shared by means of such interfaces.
- the characterization of the requirements and of their relations with problems domains.

Jackson identifies five problem frames [37].

- *Required behavior.* The problem is the definition of a machine that controls some part of the physical world by satisfying certain given conditions
- *Commanded behavior.* The problem is the definition of a machine that controls some part of the physical world according to the commands that are issued by an operator.
- *Information display.* The problem is the definition of a machine that obtains information from some part of the physical world and provides information in a required form to other parts of the world according to certain given rules.
- *Simple workpieces.* The problem is the definition of a machine that creates and handles a certain class of artefacts.
- *Transformation.* The problem is the definition of a machine that transforms some input data organized into a certain format into another format according to certain rules.

Each of the proposed frames has its own problem frame diagram and is characterized by its own requirements, domain properties and role. In the following a short overview of the most important features of each basic frame is provided.

4.4.1 Required behavior

The frame Required behavior captures the following idea:

> *There is some part of the physical world whose behavior is to be controlled so that it satisfies certain conditions. The problem is to build a machine that imposes that control [37].*

The problem is described by means of the Problem diagram shown in Figure 4.5.

The world is composed of a causal problem domain named *Controlled domain*, and a machine domain, named *Control machine*, which represent the machine to be built.

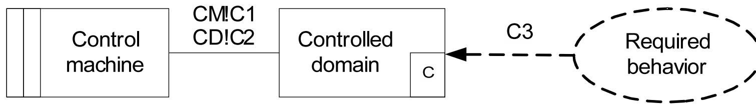

Figure 4.5: Required behavior: problem frame diagram

Controlled domain and *Control Machine* communicate by means of an interface characterized by two sets of shared phenomena named *C1* and *C2*. Notice that the phenomena *C1* are controlled by *Control machine*, while *C2* by *Controlled domain*. Such problem domain is also characterized by a further set of internal phenomena named *C3*.

The requirements *Required behavior* constrain the internal phenomena *C3* of *Controlled domain*.

4.4.2 Commanded behavior

The Commanded behavior frame represents a class of problems that captures the following idea:

> *There is some part of the physical world whose behavior is to be controlled in accordance with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly [37].*

The basic frame is described by means of the Problem diagram shown in Figure 4.6.

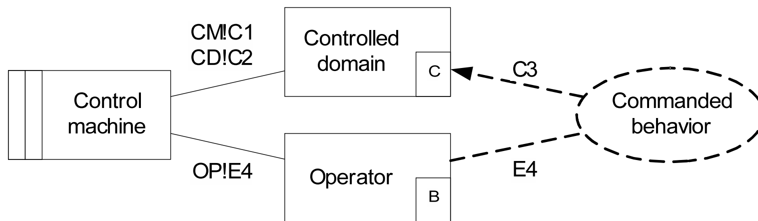

Figure 4.6: Commanded behavior: problem frame diagram

The world is composed of two problem domains named *Operator* and *Controlled domain* and by a machine domain named *Control machine*. *Operator* is a Biddable domain, while *Controlled domain* is a Causal one.

Operator communicates with *Control machine* by means of a dedicated interface. More specifically, *Operator* sends commands represented by the set of shared phenomena named *E4*.

Control machine is also connected to *Controlled domain*. Such domains communicate by means of two sets of shared phenomena named *C1* and *C2*, respectively. *Control machine* controls the set of phenomena *C1* while *Controlled domain* controls the phenomena *C2*. *Controlled domain* is also composed of a further group of internal phenomena named *C3*.

The requirement *Commanded behavior* constrains the value of the internal phenomena *C3* according to the commands *E4* that *Operator* sends to *Control machine*.

4.4.3 Information display

The Information display frame represents a class of problems that fits with the following idea:

> *There is some part of the physical world about whose states and behavior certain information is continuously needed. The problem is to build a machine that will obtain this information from the world and present it at the required place in the required form [37].*

The frame is described by means of the Problem diagram shown in Figure 4.7.

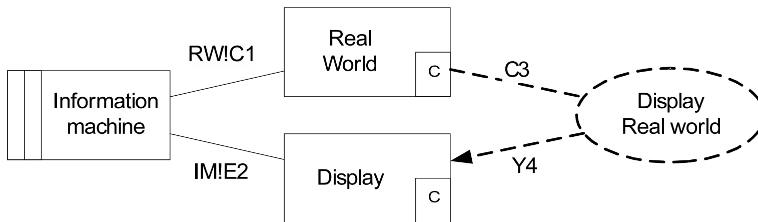

Figure 4.7: Information display: problem frame diagram

The world is composed of two problem domains named *Display* and *Real world*, and by a machine domain named *Information machine*. Both *Display* and *Real world* are causal domains and are directly connected to *Information machine* by means of two dedicated interfaces.

Information machine receives information from *Real world* represented by means of the causal shared phenomena C1, elaborates such information and sends visualization commands to *Display*. Notice that such commands are represented by means of the group of shared phenomena *E2* controlled by *Information machine*.

The requirements *Display Real world* predicate on two groups of internal phenomena of the domains *Real world* and *Display*. More specifically, symbolic internal phenomena named *Y4* of *Display* are constrained with respect to the causal phenomena *C3* of *Real world*.

4.4.4 Simple workpieces

The frame Simple workpieces captures the following idea:

> *A tool is needed to allow a user to create and edit a certain class of computer-processable text or graphic objects, or similar structures, so that they can be subsequently copied, printed, analysed, or used in other ways. The problem is to build a machine that can act as this tool [37].*

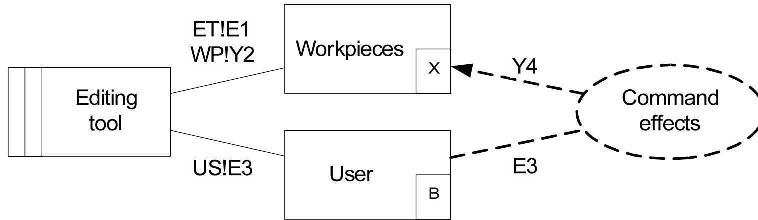

Figure 4.8: Simple workpieces: problem frame diagram

The frame is described by means of the Problem diagram shown in Figure 4.8.

The world is composed of a machine domain named *Editing tool*, a biddable domain *User*, and a lexical domain *Workpieces*.

User sends commands, represented by means of the causal shared phenomena *E3*, to *Editing tool*. *Editing tool* is connected to *Workpieces* by means of an interface characterized by the shared phenomena *Y2* and *E1*. *E1* is a group of causal phenomena controlled by *Editing tool* representing the commands that the machine sends to *Workpieces* to modify its current state, while *Y2* is a set of symbolic phenomena representing the internal state of the domain.

The requirements *Command effects* constrain the internal state of *Workpieces* on the basis of the commands provided by *User* to *Editing tool*. More specifically, *Y4* is a group of symbolic internal phenomena of *Workpieces*, while the causal phenomena *E3* are the same set of phenomena that *User* sends to *Editing tool*.

4.4.5 Transformation

The frame Transformation fits with the following idea:

> *There are some given computer-readable input files whose data must be transformed to give certain required output files. The output data must be in a particular format, and it must be derived from the input data according to certain rules. The problem is to build a machine that will produce the required outputs from the inputs [37].*

The frame is described by means of the Problem diagram shown in Figure 4.9.

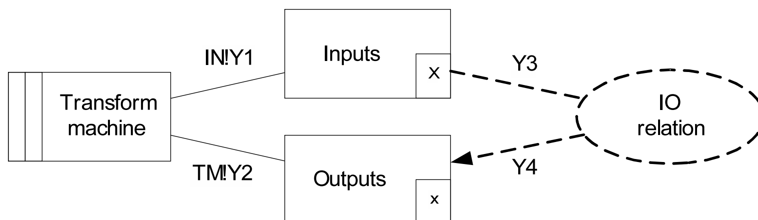

Figure 4.9: Transformation: problem frame diagram

The world is composed of a machine domain named *Transform machine*, and by two lexical domain named *Inputs* and *Outputs*, respectively. *Inputs* is con-

nected to the machine domain by means of an interface, and provides to *Transform machine* some information concerning its internal state represented by the group of shared phenomena *Y1*. *Transform machine* controls a group of symbolic phenomena named *Y2* shared with *Outputs*.

The requirements *IO relation* constrain the internal state of the domain *Outputs* with respect to the internal state of *Inputs*. Notice that the state of these domains is represented by means of two additional group of symbolic phenomena named *Y3* and *Y4*.

Y1 may or may not be the same phenomena as *Y3*; in the same way *Y2* may be or may not be the same phenomena as *Y4*. As a general rule they are likely to be different, since the machine should deal with more detailed phenomena while the requirement should refer to more abstract phenomena [37].

4.5 Concerns

In the Problem Frame approach the analyst that has to define a machine specification that satisfies the requirements of the problem.

According to Gunter et al. [26] the formal criterion for reaching a successful development is the proof of the following implication:

$$machine\ specification \wedge domain\ properties \Rightarrow requirement$$

More specifically, in case the machine behavior complies with the description of the machine specification, and the behavior of the domains satisfies the description of the domain properties, then the requirements will be satisfied.

The analyst has to make descriptions of all the artifacts, and he/she has to fit them together into a correctness argument. Such argument must show that the proposed machine will ensure that the requirement is satisfied in the problem domain.

Notice that requirements, machine specification and domain properties have to be separately described. The approach does not specify any notational support for these activities, hence the analyst is free to use the notation that he/she considers the most suited for the modeling purposes.

The goal of frame concern is to identify the descriptions that are needed and specify how such description must fit together.

Jackson in [37] proposes a different frame concern for each of the previously introduced basic problem frames. As an example consider the frame concern of the Commanded behavior frame. The requirements specify which commands of the operator are sensible and which effects they should cause on the controlled domain in case they were feasible, i.e., if these effects are allowed and realizable in the current state of the controlled domain.

The informal process that drives an analyst to show that the machine satisfies the requirements is described by means of Problem diagrams enriched with comments that describe specific phases of the behavior of the involved domains (see Figure 4.10).

In addition to the Frame concern, each problem frame also defines a few *Specific concerns* that describe additional properties that must be addressed in order to reach an acceptable solution.

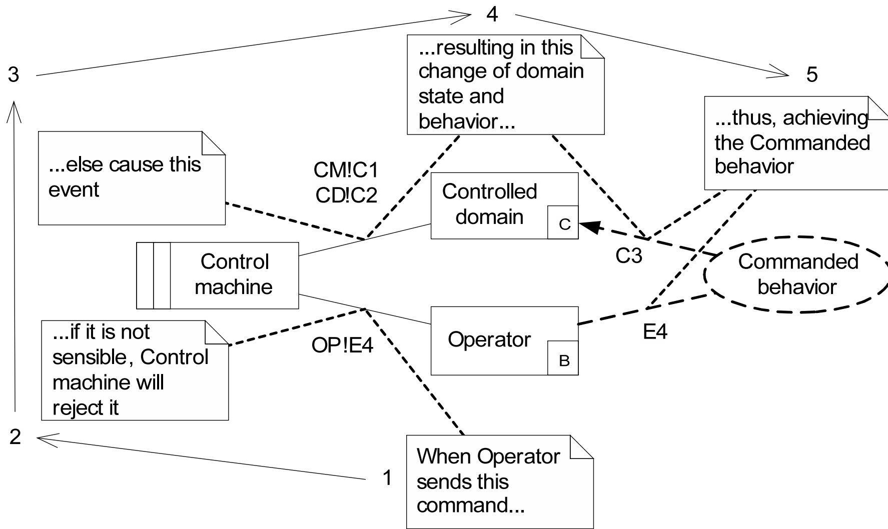

Figure 4.10: Commanded behavior frame concern

- *Initialization.* The initialization concern defines the initial state of the problem context in which the machine will run. The analyst has to specify the initial state of the machine and of the problem domains. If the developer does not consider such states, the machine behavior could not satisfy the requirements.
- *Reliability.* Both Causal and Biddable domains are, in general, unreliable, since there is no guarantee that they conform with the domain properties that are defined for a problem. The reliability concern deals with the possibilities of failure.
- *Breakage.* The breakage concern aims at identifying the sequences of operations that may damage problems domains, and ensuring that the machine always avoid them.

The proposed concerns are some of the most general ones introduced by Jackson in [37]. Notice that the set of basic problem frames proposed by Jackson is not necessarily complete, i.e., it is possible to identify problems that require basic frames different from the proposed ones. As a consequence, also the set of concerns may grow in size.

4.6 Frame flavours

Problem frames provide a very high level classification of problem domains and phenomena, that needs to be refined for a more rigorous problem analysis. Frame flavours address such issue by providing more detailed descriptions from different points of view. Each description insists on a different set of domains and events, and it takes into accounts only the characteristics that are relevant to the view.

Jackson in [37] proposes some flavours:

- *Static flavour.* Static flavours are descriptions of the internal structural characteristics of static domains. A domain can be considered static in case it has no time dimension, it does not generate events and does not change autonomously its state. Usually, static flavours are used for the description of lexical and causal domains.
- *Dynamic flavour.* Dynamic flavours are behavioral descriptions that affect dynamic domain interactions in a relatively local way. An important characteristic of a dynamic domain is its capacity to tolerate externally controlled events of which only some can cause changes in the domain state.
- *Control flavour.* Control flavours are behavioral descriptions that affect domains with numerous internal states. The flavour provides a valid support for the classication of the states and other domain characteristics.
- *Informal flavour.* Informal flavours are descriptions that affect informal domain, i.e., domains for which any formalization of phenomena and relations is approximative. The flavour guides the analyst towards a reliable description of informal domains. The flavour operates by designating a set of ground terms, and suggesting how to use them for descriptions.
- *Conceptual flavour.* Conceptual flavours are descriptions that affect domains representing abstract conceptual entities. The flavour allows the analyst to consider such entities as concrete entities for which it is possible to express relations.

Each of the proposed flavours requires different notations to support the descriptions.

4.7 Frame Variants

Frame variants are analysis artefacts that extends the description capabilities of problem frames and frame flavours. Typically, variants add domains to the problem context of a basic frame or modify the control properties of some of its interfaces.

Frame variants extends the main concerns of the basic frames by allowing one to deal with problems that cannot fit directly in the original basic frames.

Jackson in [37] propose four variants:

- *Description variant.* The description variant adds a description domain to the original basic problem frames. A description domain is a lexical domain that describes some aspects of the requirements or of the domains of a problem. A description variant is represented in Problem diagrams by means of a dedicated notational element (see Figure 4.11). A description domain can be phisically connected to the machine to indicate that the machine uses such description to determine the behavior to be imposed on other domains. A description domain can be also connected to problem domains; in such case the description domain describes the connected problem domain.

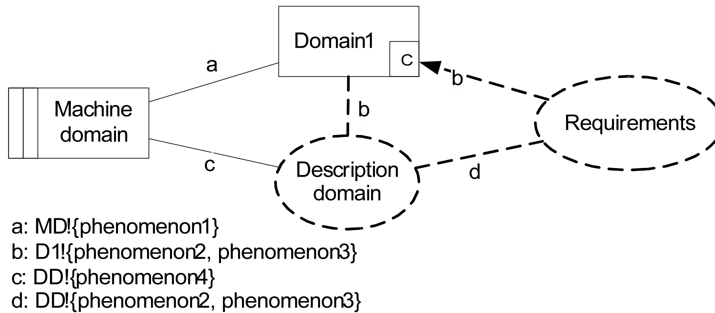

Figure 4.11: A description variant of the Required behavior frame

- *Operator variant.* The operator variant add an operator domain to the original basic problem frame. The operator modifies the behavioral aspects of the original frames by imposing innovative rules in a control problem. Notice that the frame *Controlled behavior* can be considered an operator variant of Required behavior.

- *Connection variant.* The connection variant introduces a connection domain between the machine and a problem domain of the original basic frame. In fact, in an ideal case the machine is directly interfaced with the real world, but often the machine requires a dedicated domain to communicate with the rest of the world. This variant introduces a connection domain, i.e., a domain whose properties and behavior affect the other problem domains.

- *Control variant.* The control variant modifies the control characteristics of the interface phenomena of a basic problem frame. The variant affects shared phenomena typed as events. It allows one to express which events have to occur, when it has to occur, and which individuals have to participate in the event.

Notice that multiple variants can be applied at the same time to the same problem frame.

4.8 Problem decomposition

Most problems are too complex to be modeled as basic problem frames. Real world problems are usually hard to understand, analyse and solve. The key to dominate problem size and complexity is represented by decomposition.

Notice that decomposition represents not only an approach to reach the solution of a problem, but also a process that helps the analyst to understand and analyse the problem itself.

Decomposition is a *divide et impera* approach to the solution of a problem. The initial complex problem is analysed and simpler and smaller sub-problems are derived from the original one. The solution of each sub-problem will contribute to the solution of the whole problem.

Decomposition aims at reducing problem to a set of sub-problems that fit basic problem frames. Decomposition consists in projecting the original problem into sub-problems. The projection mechanism is at the basis of the whole decomposition process. A projected sub-problem is characterized by requirements, machine, problem domains and interfaces that are a subset of their counterpart in the original problem. The application of projection generates sub-problems that may partially share requirements, domains, phenomena and interfaces. In other words sub-problems may share some parts.

Jackson indicates several principles that may help the analyst when decomposing a problem.

- *Subproblems are complete.* The analyst has to consider each sub-problem as a complete problem characterized by its own problem diagram, machine domain, problem domains and requirements. Each sub-problem is independent from the other ones and must be faced separately.
- *Parallel structure.* Each sub-problem is a projection of the original full-problem. The analyst has to consider the sub-problem as fitting into a parallel structure (rather than a hierarchical structure). The overall problem description is obtained by composing the description of all the sub-problems that parallelely describe different aspects of the general problem.
- *Concurrency.* The analyst has to take into account the relationships among the subproblems and how they overlap and interact.
- *Composite problem frames.* The analyst has to take into account composite frames, i.e., recurrent classes of problems characterized by a standard decomposition into basic subproblems.

Once the sub-problems are identified, analysed and solved, their solution has to be used to recompose the solution of the original problem.

Problem composition is a critical activity; as an example, in [34], Jackson states that

> *Problem complexity arises from the interactions of (a problem's) subproblems and of their solution, both as a problem structure and as a solution structure.*

Such interaction can occur whenever multiple sub-problems share a problem domain or a machine domain. The correctness of the composition is the subject of composition concerns [37]:

- *Conflict.* In case two property descriptions refer to the same elements, they can be inconsistent. Inconsistency may affect both indicative or optative descriptions. The concerns aim at guiding the analyst to the conflict resolution by introducing priority criteria.
- *Interference.* Two subproblems interfere whenever one of the problem causes change in a problem domain that is inspected by the other problem. The concern aims at introducing mechanisms for mutual exclusion of atomic changes and inspections.

- *Scheduling*. In case two problems interfere and their interference cannot be addressed by means of atomic operations, the concern aims at resolving the interference by scheduling the operations.

Composition concerns are motivated by the fact that, in problem projection, the division of requirements may cause some information concerning the sub-problem interactions to be lost.

4.9 Final remarks

This chapter provided a very high level introduction to Problem Frames. The reader interested to a complete and detailed presentation should refer to the Jackson's web site [2], which provides numerous references to articles, books and papers that describe in depth the requirements analysis approach.

In addition to the initial proposal of Jackson, numerous authors have researched Problem Frames by providing the requirements engineering community with interesting results in the areas of principles, techniques, processes and semantics of Problem Frames. A roadmap of Problem Frames research was proposed by Cox et al. in [16].

At present open area of research on Problem Frames mainly concern:

- the identification of new basic problem frames. The basic frames proposed by Jackson represent a starting set that need to be expanded.

- the proposal of an effective notation. The notation proposed by Jackson is more suited to illustrate the concepts proposed by the approach rather than to support software development in industrial processes. Moreover, the notation is not complete since it does not support the definition of behavioral aspects. All these limits hinder the usage of Problem Frames in industry.

- the application of the approach to case studies from the real world. The case studies presented by Jackson in his book [37], as well as the examples illustrated in numerous papers in the literature simply illustrate particular aspects of the requirements analysis approach. Thus the application of this approach to real world case studies is still missing. Notice that such application, besides being an effective instrument to validate the scalability of the approach, may also provide several hints for identifying new basic frames.

- the definition of approaches to compose the solution to sub-problems. The ability to compose solutions allows one to take a set of decomposed requirements, to provide individual solution to such requirements and then to address the overall system requirements by recomposing solutions [41]. Innovative approaches, like the one proposed in [41], aim at defining composition techniques that try to overcome the difficulties that arise when recomposing inconsistent requirements.

[2] `http://www.jacksonworkbench.co.uk/stevefergspages/pfa/index.html`

- the definition of tools that support the whole analysis approach. At present no tool supporting the Problem Frame method has been defined yet. This lack, which may also be reconducted to the lack of an effective notation, hinders the application of the approach in industrial processes.

Chapter 5

Applying Problem Frames

This chapter discusses the application of the problem frame methodology to the specification of the requirements of a system in charge of monitoring the transportation of dangerous goods (namely, petrol and other types of fuels) by trucks.

The project aimed at developing a first experimental version of the system, to be used to monitor a relatively small set of trucks (initially ten, expected to grow to about one hundred during the experimentation period) in the italian region of Lombardia (the region of northern Italy including Milan and about 24,000 Km2 wide). The system is expected to provide firemen, police and rescue squads with timely and precise data in case of accidents involving trucks carrying dangerous loads.

The goal of the chapter is to show that using problem frames in the development process of industrial software (of a rather critical nature) is not only possible, but brings relevant benefits in terms of easing the achievement of a fairly complete and clear comprehension of requirements in the early phases of the project.

The chapter is organized as follows: in Section 5.1 the requirements of the monitor system are described informally; Section 5.2 describes the problem domains and the shared phenomena; the main system requirements are modeled by means of problem frames in Section 5.3; finally Section 5.4 illustrates the lessons learned.

5.1 Informal System Requirements

In Italy the vast majority of goods (including dangerous ones) are transported on roads by trucks. The constant presence of potentially dangerous loads on the roads is worrying, because of worsening traffic conditions, because the roads often go through areas that are densely populated, and even because trucks full of fuel could be a target for terrorists.

For all these reasons, the national authority for transportation decided to start a program aimed at controlling the transportation of dangerous goods on the roads. The first step was to create an ICT system to monitor the trucks carrying dangerous loads. For this purpose, a pilot project was started in the

region of Lombardy, which is the most critical from the point of view of traffic, population density and amount of dangerous goods circulating on the roads.

Figure 5.1: Visualization of mission data and alarms

5.1.1 The Requirements

The project has to exploit an already existing infrastructure, created by the companies that own the trucks. In fact, trucks are equipped with an on-board device (a special purpose little computer) that reads the state of the truck and the load from *ad hoc* sensors and communicates these data via GPRS to a central server. In particular, the data provided by the on-board device include: the position and speed of the truck, the temperature and pressure of the load, the amount of fuel on board, and any specific critical events (crash, upset). The data collection infrastructure was created by the transport company for several purposes: to support invoicing, to perform optimizations, to check the correctness of spills.

One of the constraints for the project was that it was not simply possible to change the existing infrastructure. In practice, the only possible modification concerned the format of the transmitted data. Noticeably, it was not possible to change the data refresh and communication frequency: data are transmitted by every truck approximately every minute (except for critical events, which are communicated immediately).

The first requirement for the system was thus to collect the data transmitted by the trucks and to store them in a database, in order to make the data available

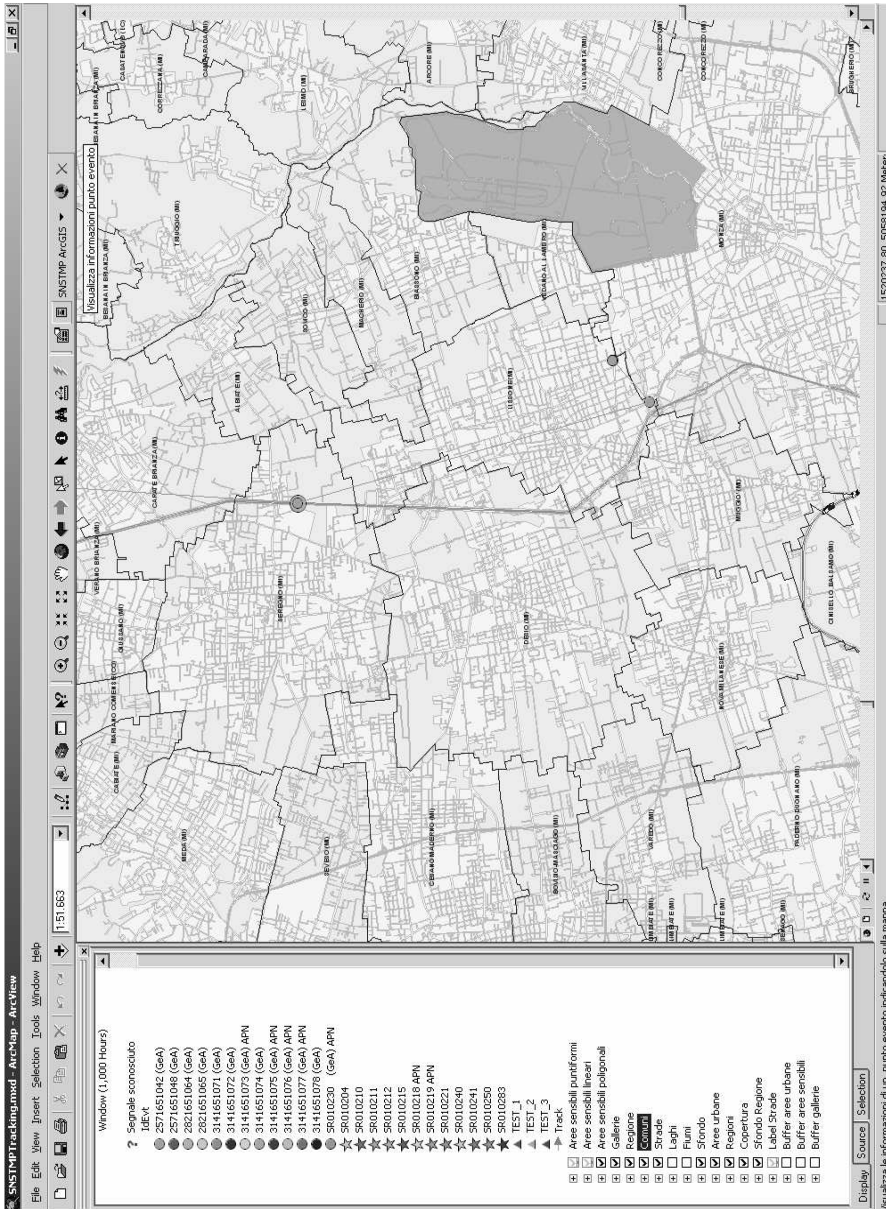

Figure 5.2: The geographical display

for querying.

A second requirement concerned the management of the 'Mission Identification Code' (MIC). In fact, when the project was started a new law was going to be activated, concerning the transportation of dangerous goods. This new law -in the part involving fuels- requires the creation of a mission document describing the planned journey of the trucks, the type and amount of fuels transported, and where it is going to be delivered. Every single mission has to be identified by a MIC. Therefore, the computer system was required to store the mission document and to identify the traveling trucks by means of their MIC[1]. Mission documentation is provided by the truck company, while the MIC is assigned by the monitoring system. This implied that the system provided an interface to the truck company, and that this interface should allow the users to input the required data in the most efficient way, in order to make the activities required to support the monitor system as inexpensive as possible for the company. In Figure 5.1 the data of the selected mission are displayed on the left hand side, while in the center the list of ongoing missions is reported.

A third requirement concerned the tracking of truck positions on a geographic display. It was required that the GPS coordinates provided by the truck were fed to the Geographic system (already in use at the Supervision and Control Center) in order to display the position of trucks on the maps (which were also already available). The geographic display of the released prototype is shown in Figure 5.2.

Another requirement, probably the most important, was that the system had to immediately notify alarms and potentially dangerous situations. Different types of alarms were identified:

1. Some events, like the crash and the upset, are reported by the truck and are simply recognized as alarms by the system.

2. Some states of the load or the truck are considered as alarms: in these cases it is the system that detects the alarm on the basis of the raw data provided by the truck.

3. Some alarm conditions depend (also) on the position of the truck. For instance, high speed in ana uninhabited area or the temperature too high in a gallery are recognized as alarm conditions. In some cases the system has to consider the data from multiple trucks. For instance, the presence of three or more trucks in a long gallery is an alarm.

In Figure 5.1 the alarms are displayed in the bottom part of the window (a single alarm, highlighted in yellow, is reported). Finally, the system had to be able to answer queries from the operator concerning the situation of every known truck.

It is quite relevant that the system had to notify alarms only to the operator, while calling for the intervention of firemen, police, etc. was not among the responsibilities of the system. In fact, the system had to be installed in the Supervision and Control Centare of the Civil Protection Agency, where the operators alert and coordinate all the organizations required for managing the emergency.

[1]This means that the same truck will be associated with different MICs at different times

The system had to be used as a prototype, to gain knowledge about several issues including the reliability and accuracy of the transmitted data, the usability, reliability and scalability of the system, the definition of the alarms, etc.

It is interesting to note that the main goal of the prototype was -as it is often the case- to test the requirements, i.e., to check whether additional or different requirements should be taken into account for the final version of the system.

5.1.2 The Origins of Requirements

The requirements and domain information were provided by multiple sources.

Truck companies had already determined the information that were collected on board and transmitted. The information were available in the technical documentation; in particular, the documentation of the transmission protocol contained all the relevant information.

The requirements concerning the mission data and the MIC were provided by the regulations and directives coming from the national authority for transportations.

The geographic display and the maps of the region were provided by the civil protection. Region Lombardia required that the existing geographic infrastructure were reused, in order to limit the cost of the development.

The requirement concerning the use of a database was originated by two considerations. One is technical: data concerning the position of trucks had to be available when needed, and trucks cannot be queried, therefore the only possible solution is to store the received data. The other consideration is that future uses of the system will possibly require historic data, therefore the users demanded that all the received data were collected in a database.

The rest of the requirements, i.e. those concerning the tracking and display of trucks, the detection of alarms and the queries to be supported, were expressed by the civil protection personnel. The actual requirements were determined by the needs of the civil protection (to know the position and contents of trucks) and the available data, and the existing infrastructure. The existence of similar systems (e.g., a system monitoring the position of fire extinguisher planes and helicopters) provided reference models on which new requirements could be defined and evaluated by comparison.

5.2 The model of requirements

The problem diagram of the monitor system is reported in Figure 5.3. The On-board computer, Sensors and Truck and Load domains are given (the company which owns the Truck is not willing to modify them). The On-board computer acts as an interface to the system; they are reported to express the connection between the real world and what is reported and monitored.

Phenomena labelled as 'a' are controlled by the On-board computer and made visible to the Monitor machine. They are described in Table 5.1.

The asterisk indicates that multiple instances of the data can be present: for instance a set of data for each type of fuel, or for each tank.

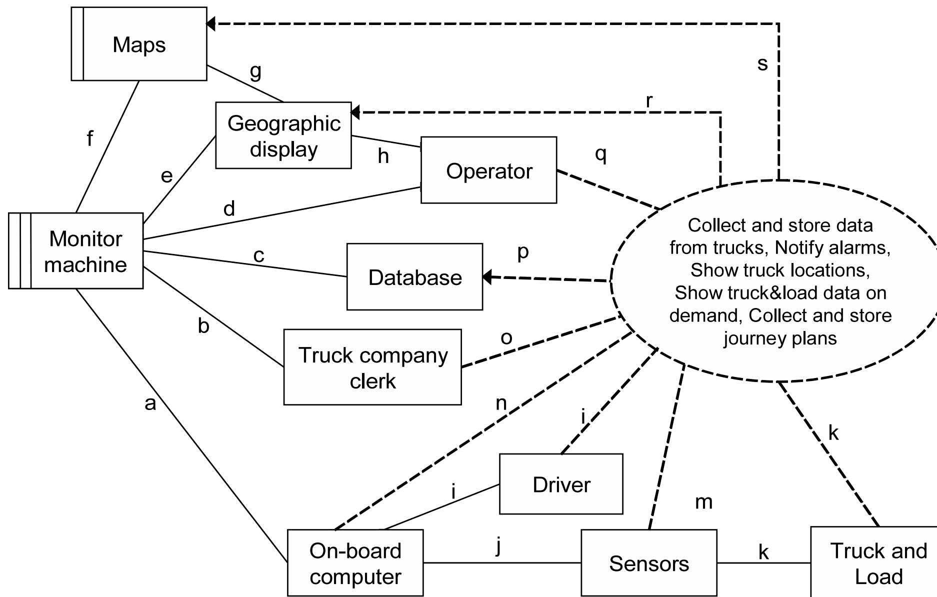

Figure 5.3: The problem diagram for the monitor system

Table 5.1: Truck and load data

Message	Attributes
Direction&Speed	speed, direction
OnBoardComputerOn	time, (type_of_fuel, total_qty)*
OnBoardComputerOff	time, (type_of_fuel, total_qty)*
Load	time, progressive, (tank_id, type_of_fuel, qty, temperature)*
Unload	start_time, end_time, progressive, type, final_total_qty, planned_qty, acual_unloaded_qty, (tank_id, unloaded_qty, temperature)*
OpenClose	OpenClose: Boolean
Completely_empty	Completely_empty: boolean
Overflow	Overflow: Boolean
ManualInput	Code (enumerative)
TankState	(tank_id, type_of_fuel, qty)*
Event	Code(enumerative)

In addition to the attributes described above, each message includes the MIC, the position, the plate of the trailer on which the transmitter is located, the observation time and the transmission time.

Phenomena labelled as 'b' are: Tcc!MissionDocument and MM!MIC. The latter is the Mission Identification Code produced by the Monitor Machine, the former is composed of several data, the most relevant of which are illustrated in Table 5.2.

Table 5.2: Mission data

Datum	**Attributes**
Freight company	Name
Client	Name
Vehicle	Plate
Driver	Name
Origin	Address, date
Destination*	Address, type of fuel, qty

Phenomena labelled as 'c' are controlled by the Monitor machine and written into the Database. The set of phenomena 'c' is given by the union of phenomena 'a' and phenomena 'b'. In practice: all the available data about the trucks, their loads and journeys have to be stored into the database. In addition, phenomena c include MM!alarms, i.e., the alarms computed by the Monitor Machine that are also stored in the database.

Phenomena labelled as 'd' are: Op!queries, MM!data, MM!alarmNotifications, Op!commands. The commands issued by the operator are meant to clear alarms, to retrieve data concerning trucks and/or alarms, or to perform configurations or other operations such as scrolling, zooming, etc.: the latter are not considered here (also because they were already supported by the available SW platform).

Phenomena labelled as 'e' are: MM!displayCommands. The Monitor machine sends commands and coordinates to the geographic display to get the trucks displayed.

Phenomena labelled as 'f' are: MM!geographicAnnotations and Ma!geographicData. The former are definitions (e.g., of critical areas) that are input by the operator through the machine. The latter are the actual maps.

Phenomena labelled as 'g' are essentially equivalent to Ma!geographicData.

Phenomena labelled as 'h' are the visualizations (GD!graphicRepresentation) of the truck situation as available in the database.

Phenomena observed and constrained by the requirements are not described here. They will be described later, together with the problem frames for which they are relevant.

5.3 The PFs based model of requirements

The main requirements for the system are the following:

- Collect and store mission data / provide MIC;

- Collect and store data from truck;
- Tracking and display;
- Answer queries;
- Edit maps / danger areas;
- Detect alarms, store them and notify them, textually and geographically.

Each of these requirements was studied and modeled in terms of problem frames. In some cases a problem frame composition was needed to model the requirements.

Since most of the user did not have the background needed to understand and effectively use problem frames, the approach was used only in the 'background', i.e., they were not used during the discussion with the users. It would have made necessary to present and explain problem frames to the stakeholders, at a cost that was not compatible with the budget.

The design and implementation of the system was based on textual documents that were written having in mind the characterization of the system in terms of frames.

This section illustrates how the requirements for the monitor system are expressed by problem frames.

5.3.1 Store/Collect Mission Data and Provide MIC

It is required that the mission data -provided by the truck company clerk- are stored in the database, after having been checked for completeness and correctness. Note that the usage of a database is imposed by the user requirements. Then, the system computes the MIC as a function of the provided data, stores it in the database, and returns it to the user.

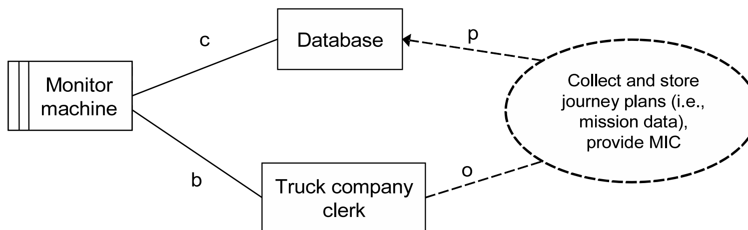

Figure 5.4: The Problem diagram for the Collect and store mission data / provide MIC requirement

The first part of the requirement can be quite clearly classified as a workpiece problem. The rest of the problem can be seen as part of the same workpiece problem frame if it is acceptable that: a) the workpiece is formed also by data derived from those provided by the user, and b) a return value is possible. Overall, these conditions seem acceptable, and the requirement can therefore be classified as a workpiece problem frame.

As in any workpiece frame, the requirements state that the content of the database (p) as resulting from the data and commands (c) provided by the Monitor Machine as a consequence of editing commands (b) are correct with respect to the same commands (o=b). The requirements also contain consistency rules for the database contents. For instance, the quantity of loaded or unloaded fuel must be non negative and less than the maximum capacity of the truck.

5.3.2 Collect and store data from truck

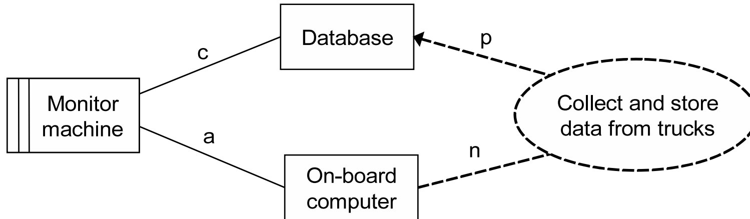

Figure 5.5: The workpiece Problem diagram for the "Collect and store data from truck" requirement

The requirement states the fact that Monitor Machine receives data from the On-board Computer must be interpreted as an order to store the same data in the database. The machine checks the syntactic and semantic correctness of the data, then correct data are stored.

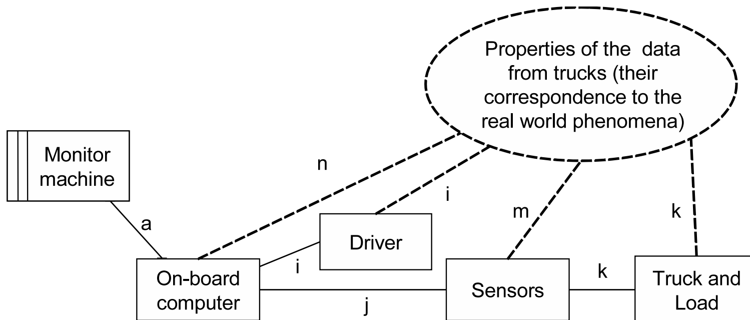

Figure 5.6: The problem diagram for the truck and its driver and devices, with indicative requirements

The requirement can thus be seen as a workpiece frame:

1. commands and data from the On-board Computer (a) are analyzed with respect to their meaningfulness and viability,

2. they are converted into commands and data (c) to send to the database

3. the database receives and interprets such commands by generating contents (p), which must be coherent with commands (n=a).

There is a difference with respect to the traditional workpiece frame described in [37]: in this case, the domain originating the commands is not, strictly speaking, a biddable one. However, from the point of view of the monitor machine, the truck can issue any command at any time; moreover, transmission and sensor errors occur quite frequently. Overall, the behavior of the on board computer perceived by the monitor machine is quite close to a biddable one.

The requirements described above refer only to the data provided by the on board device of the truck, since the only responsibility of the monitor system is to store the data that it receives and recognizes as correct. However, we are also interested in the correspondence of such data to real world phenomena. We are interested in storing a representation of phenomena of the real world, hence we need to extend the problem diagram by including the description of such phenomena and of the domains that control those. Figure 5.6 reports the resulting problem diagram that introduces a model of the truck and its devices. The requirements specify the properties of the reported data (a) with respect to the real situation, given by the state of the truck and load (k) and the manual inputs from the driver (i).

5.3.3 Tracking and Display

The tracking and display requirements state that data referring to speed and position have to be displayed on a map, so that the operator can be visually informed about the position, speed and direction of every truck. The problem corresponds to a display frame.

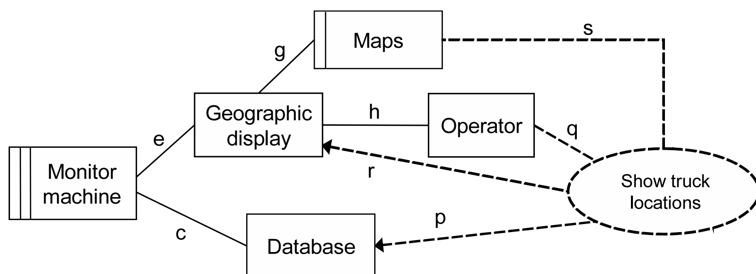

Figure 5.7: The Display Problem diagram for the "Tracking and display" requirement

Commands are sent to the Geographic Display according to the data from the database. This is the only responsibility of the machine, since the geographic display is actually an existing machine that is able to interpret the coordinate and movement data and perform a suitable visualization.

The problem is more complex than the usual display frame, since we explicitly represented that the display uses the Maps to interpret the commands and to perform the visualization used by the Operator.

5.3.4 Alarm detection and notification

The requirement states that the state must be examined and alarm conditions must be evaluated accordingly, whenever new data from the trucks arrive. In practice, alarms are functions of the data stored in the database. This can be seen as a transformation problem: alarm data are derived from the state data.

The Problem diagram is shown in Figure 5.8, where the alarm rules that determine alarm conditions are modeled explicitly as a configurable domain (the editing of the Alarm rules is a workpiece problem). Note that here the database plays both the role of data source and destination.

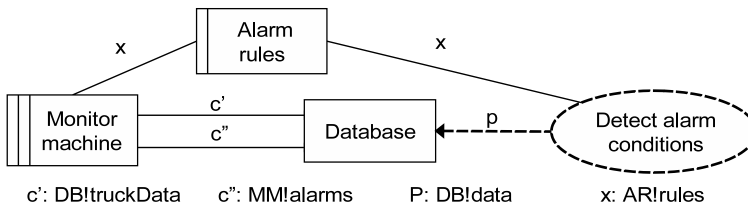

Figure 5.8: The Problem diagram for the "Alarm detection" requirement with configurable alarm rules

Alarm detection is only the first part of the requirement. Alarms once detected must be immediately notified to the operator. The notification has to be both textual (a message is displayed on the operator's console) and graphical (the involved truck is highlighted on the geographic display). The complete problem diagram for the requirement is shown in Figure 5.9.

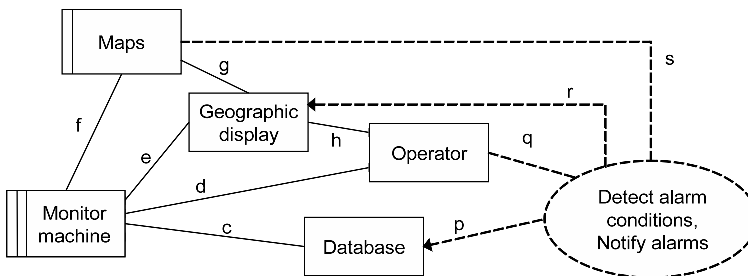

Figure 5.9: The problem diagram for the "Alarm detection and notification" requirement

The problem can be represented as the composition of three problem frames:

1. The alarm identification and storage problem is a Transformation problem, as discussed above.

2. The alarm textual notification is a Display problem. The requirements are very simple: the data describing the alarm have to be shown as they are, the only elaboration consisting in using a background colour corresponding to the severity of the alarm. In Figure 5.9 the domains involved are, besides the

machine, the Database and the Operator. The latter domain is supposed to include a simple display (a causal domain) that is able to visualize textual data.

3. The alarm graphical notification is a distinct and independent Display problem, involving the database as observed element and the geographic display as the display domain. In Figure 5.9 the domains involved are, besides the machine, the Database and the Geographic display.

5.3.5 Map Annotation

Another system requirement states that the operator can annotate the existing maps by defining dangerous zones or road intervals (e.g., corresponding to galleries). In practice the user is allowed to annotate the maps by defining areas and specifying their properties. This is a workpiece problem (with the Operator being a biddable domain, and the Maps a lexical domain); the diagram is reported in Figure 5.10.

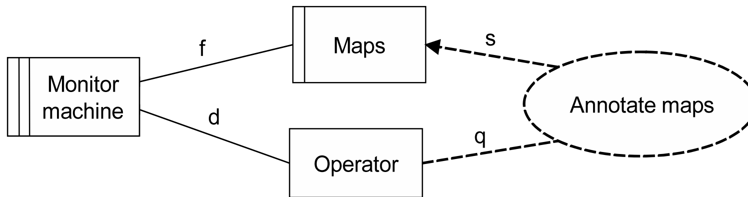

Figure 5.10: The workpiece Problem diagram for the “Map editing” requirement

However, the annotation of maps requires a precise and sophisticated feedback from the user. It must be possible for the user to see on the map the defined area and its boundaries and properties, in order to verify that the area has been properly defined. Therefore, the representation of the requirement is not complete without the specification of the feedback provided by the geographic display, which can be modeled as a display problem (the diagram is given in Figure 5.11, where the Maps domain plays the role of data supplier, and the Geographic Display is the data displayer).

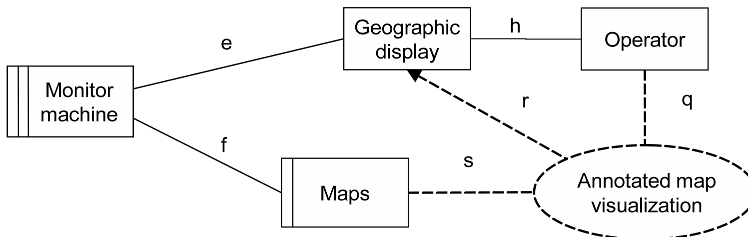

Figure 5.11: The problem diagram for the “Map editing visual feedback” requirement

The complete diagram for the map editing requirement, including the geographic visualization feedback, is given in Figure 5.12.

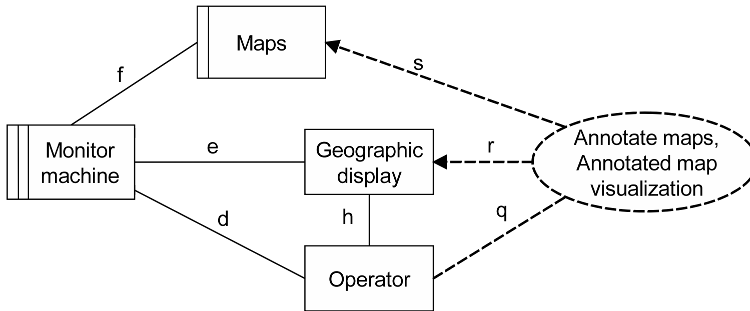

Figure 5.12: The problem diagram for the "Map editing visual feedback" requirement

5.4 Lessons learned

In general, the problem frames approach proved to be suitable for modeling most of the requirements of the monitor system.

However, considering the possibility of using problem frames in an industrial context, methodological guidelines are needed. In fact, the current definition of problem frames and the available documentation sometimes leave the analyst with doubts about the correct way of representing problems via problem frames. For example when considering the user needs (i.e., the requested functionality) rather than the machine responsibility (i.e., what the system is supposed to do) results in different problem frames. In the monitor system the dilemma may arise when considering the map editing requirement: the user needs the annotated maps visualized, thus it seems to be a display problem, but the machine has to convert the editing actions into commands for the display, thus it could be regarded as a transformation problem. Good documentation and guidelines could solve these types of doubts and make the adoption of problem frames easier.

A second issue is that in some cases no problem frame seems to match the given requirements, even relatively simple ones. Consider for instance the requirement which states that the operator can query the database about the situation of the trucks. The problem diagram is reported in Figure 5.13.

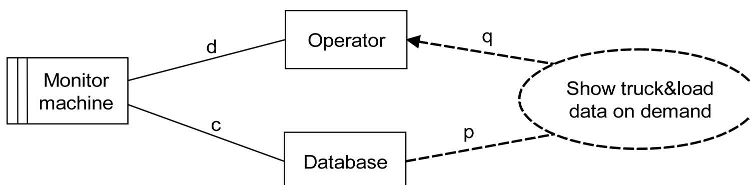

Figure 5.13: The problem diagram for the "answer queries" requirement

Although the diagram in Figure 5.13 is quite simple, it is difficult to find a problem frame -among those proposed in [37]- that matches with this problem. This is probably due to the fact that requirements involve the usage of a database, which is a lexical domain.

Considering the PFs proposed in the literature, the Simple Information Answer [36][35] models a similar problem, but with a relevant difference: the object of the enquiry is the real world, while we want to read from a lexical domain. Since we have to retrieve information from a database, which is a lexical domain, we propose a new problem frame, which we name Data Repository Enquiry, to stress that the source of information is a lexical domain. The problem frame is represented in Figure 5.14.

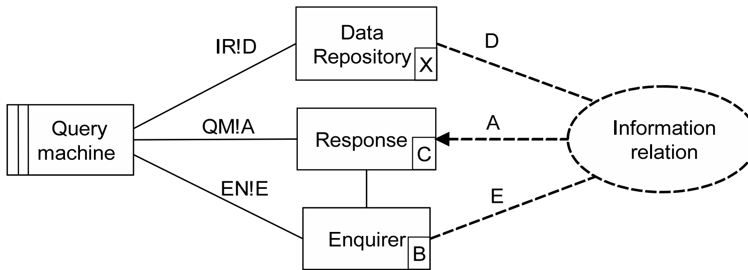

Figure 5.14: Problem Frame Diagram: Data repository enquiry

The information relation requirement states what answer (A) the Query machine has to produce, when it receives query E and the contents of the repository is D.

Note that in both the Simple Information Answer and the Data Repository Enquiry frames, the actual source of the information is the Enquirer; nevertheless, the causal Response domain is introduced because the Enquirer is a biddable domain, which cannot be constrained. Instead, the Response is constrained, as shown in Figure 5.14 and discussed above.

During the requirements modeling activity, it was observed that often the problems are composed of sequences, such that a problem triggers the following one. For instance, the data storage in the database triggers the alarm evaluation, which in turn triggers the alarm visualization; the annotation of maps triggers the visualization of annotated maps, etc. This is another effect of the presence of a database: in fact the access to the database divides the problem into subproblems. Consider for instance the detection of alarms: if the requirement was just to detect alarm on the basis of data received from trucks, then it would be a simple display problem. Instead in the monitor system we have three subproblems: receive the data and store them into the database, then retrieve the data from the database and detect alarm conditions, finally show currently active alarms. Note that the data retrieved from the database are generally a superset of those just stored (for instance, an alarm could involve several trucks, e.g., when they are closed to each other). Therefore, the database plays a primary role in the problem.

Finally, we noted that some problem diagrams provide suggestions for the design and implementation phases. The most outstanding case is given by the

problem diagram for the “Map editing visual feedback” requirement shown in Figure 5.12. It is possible to see that the problem fits quite well in the Model-View-Control pattern: the Map is the Model, the View is given by the phenomena (labelled 'h' in Figure 5.12) shared between the Display and the Operator, the Control is expressed by the rules that connect the commands (phenomena 'd' in Figure 5.12) to the effects on the Map (phenomena 'f' in Figure 5.12) and on the Display (phenomena 'e' in Figure 5.12). Therefore, we can conclude that a careful analysis of the problem diagrams can not only provide indications for the design activities, but even indicate precise design patterns.

Chapter 6

A SysML & Problem Frames based approach

Altough the nature of SysML diagrams suggests a sequence of modeling phases and imposes constraints on the modeling activities, SysML is not provided with a specific modeling methodology. This lack is particularly relevant since SysML offers linguistic means to describe and allocate requirements, but does not provide users with any hint on how to define and structure requirements nor on how to classify the problems they represent. The guidelines based on the Reference model for requirements specification by Gunter et al. [26] proposed in Chapter 3 start addressing such issue. The aim of this chapter is to extend those guidelines by defining a requirements analysis approach based on Problem Frames [37, 33].

Problem Frames (PFs) [37, 33] drive developers to understand and describe the problem to be solved. The PFs approach has the potential to dramatically improve the early lifecycle phases of software projects. Nevertheless, PFs have some limitations that hinder their application in industrial software development processes. In particular, they are not provided with adequate linguistic support. The modeler has to choose a suitable language to predicate about phenomena. Moreover, the PFs approach causes a sort of "impedance mismatch" with respect to the languages employed in the subsequent development phases. Let us consider the quite common practice of UML-based development: in such case, PFs should be "translated" into UML. This is an error-prone activity that makes traceability harder and requirements more difficult to read for developers not familiar with PFs.

Therefore, it appears convenient to integrate modeling languages with the PFs approach. In fact, in [43] it is explored the integration of UML with the PFs concepts. The result was that UML-based development can be equipped with a sound and rigorous method for requirements analysis and representation. From the PFs point of view, the method is equipped with the linguistic support provided by UML that is reasonably expressive and easy to use. Nevertheless, the experimental integration of UML and PFs revealed that UML suffers from several limitations that hinder its usage in combination with PFs. In fact, UML is too much design-oriented, and does not support modeling at the correct level

of abstraction (PFs generally require a level of abstraction higher than the one adopted by UML). For instance, UML components are the best option for modeling problem domains, but UML imposes that all relations between components are interfaces, while at the requirement level it would be preferable to express more abstract relations (specifying interfaces implies design choices, e.g., what component takes the initiative of "calling" the interface provided by the other component).

Moreover, UML defines the OCL (Object Constraint Language) [52] to specify properties that cannot be represented in UML diagrams. However, OCL suffers from several limitations that do not allow specifying many properties and requirements as needed. For instance, when dealing with time-dependent systems, OCL does not allow referencing different time instants in a single OCL formula (except for attribute values before and after method execution).

SysML promises to perform better than UML with respect to the aforementioned issues. Therefore, the purpose is to integrate PFs concepts with SysML and verify how well the linguistic constructs of SysML support PFs-based requirements modeling. Although some researchers have studied how to extend PFs based methods to the design phases, these topics are out of the scope of this chapter. Here we address exclusively the requirements modeling phase.

The chapter is organized as follows: Section 6.1 introduces a running example used to illustrate the approach. Section 6.3 presents the combined approach of SysML and PFs with the help of the proposed example. Section 6.4 accounts for related work.

6.1 The Sluice Gate example

In order to illustrate the applicability of PFs concepts in SysML modeling, the well known problem of controlling a sluice gate [37] (Figure 6.1) is used as a running example.

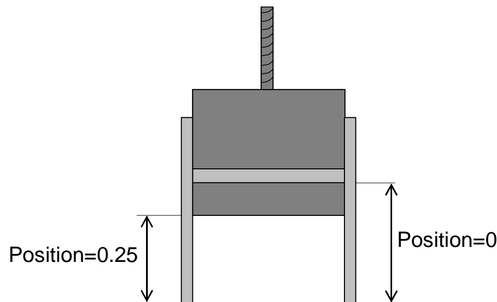

Figure 6.1: The sluice gate.

A small sluice, with a rising and a falling gate, is used in a simple irrigation system. A computer system is needed to raise and lower the sluice gate in response to the commands of an operator. The gate is opened and closed by rotating

vertical screws. The screws are driven by a small motor, which can be controlled by clockwise, anticlockwise, on and off pulses. There are sensors both at the top and the bottom of the gate travel: when the top sensor is active the gate is fully open, when the bottom sensor is active, it is fully shut. The connection to the computer consists of four pulse lines for motor control, two status lines for the gate sensors, and a status line for each class of operator commands.

The position of the gate is defined as the percentage of space occupied by the gate: when it is open Position=0, when it is closed Position=1. Finally, the top and the bottom sensors are active when Position becomes less than 0.05 and greater than 0.95, respectively.

The PFs diagram for the sluice gate problem as given in [37] is reported in Figure 6.2. According to Jackson's classification, the PF in Figure 6.2 is a com-

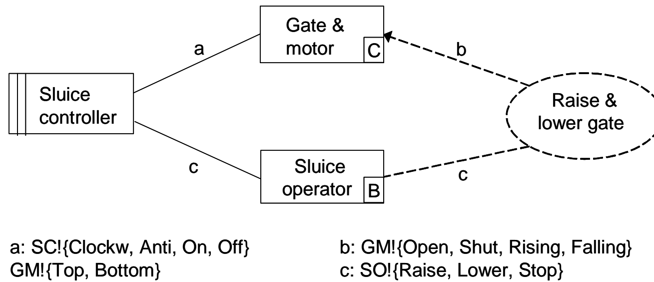

Figure 6.2: The sluice gate commanded behavior frame.

manded behavior frame, since it identifies a problem where "*there is some part of the physical world whose behavior is to be controlled in accordance with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly*" [37].

The example consists of three domains: the Sluice Controller, which is the machine that will be developed to satisfy the requirements; the Gate & motor, which is the domain to be controlled (it is a causal domain since its properties include predictable causal relationship among its causal phenomena); the Sluice Operator, which is a biddable domain indicating a user without a positive predictable behavior (that is, the user can issue commands but cannot be constrained to act in any way).

The next step consists of addressing *frame concerns*, which identify the descriptions that have to be provided with every problem frame. More specifically, addressing frame concerns means making requirements, domain and specification descriptions fit together properly. The combination of these descriptions must result in a 'correctness argument', that is, they must provide evidence that the proposed machine will ensure that the requirements will be satisfied in the problem domain [37]. In the case of the commanded behavior frame, we have to assure that only sensible and viable commands are executed. For instance, the machine should not try to open an already open gate.

Requirements can be expressed as effects on the problem domain caused directly by the user's commands or by other events, such as reaching the completely open (closed) position. According to Jackson, these effects can be expressed in a

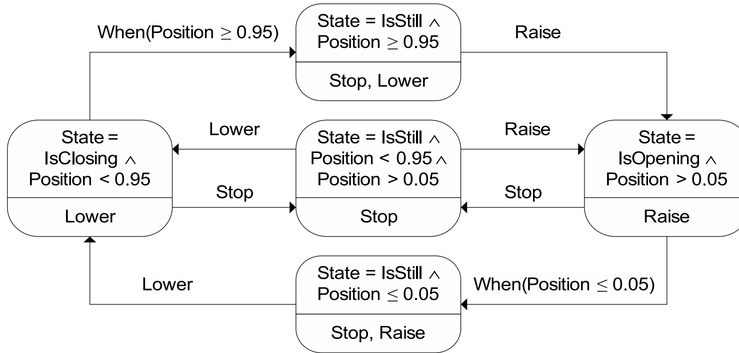

Figure 6.3: Requirements: reaction to commands and events.

rather straightforward way by means of state machines.

Note that in Figure 6.3 some transitions conditions are expressed on Position (clause `when`) rather than on the sensor state. In fact, requirements generally address the real state of the problem domain, rather than the representation of such state that is made available to the machine. Figures 6.3 and 6.4 use the same notation used in Jackson's book: states are divided in two regions, the upper one contains the state invariant, i.e., the condition that holds when the state machine is in that state; the lower one indicates the events that are ignored in the state.

The requirements reported in Figure 6.3 (taken from [37]) do not consider that a Raise command could be issued when the gate is closing. We adopt an extended version of these requirements, according to which in the aforementioned case the gate should switch to the opening state. This requires a "good" behavior of the machine, since just switching the working direction would break the motor (or bring the domain in an unknown state as specified in Figure 6.4).

Also the behavior of the problem domain can be represented by means of a state machine, showing the states of Gate & motor, and specifying the reactions to external commands, as well as the evolution in time of the domain. Such state machine (taken from [37]) is reported in Figure 6.4. Note that state 5 is indicated as "unknown"; in fact it corresponds to an undesirable situation, since it probably involves breaking the motor and/or the gate (e.g., by trying to lower the gate beyond the closed position). Since we are describing the problem domain, we have to explicitly indicate this possibility, but, knowing that it should be never reached, we do not need to explore it in detail.

According to the PFs methodology, the machine specifications combined with the problem domain specifications, must satisfy the requirements. We could specify the behavior of the machine by means of another state machine or a set of logic clauses. Here we skip this phase for space reasons.

6.2 Representing PFs concepts

In order to support the PFs approach, a modeling language has to satisfy a set of requirements concerning the representation of the relevant concepts according to the definition of the PFs technique [37] and semantics[27].

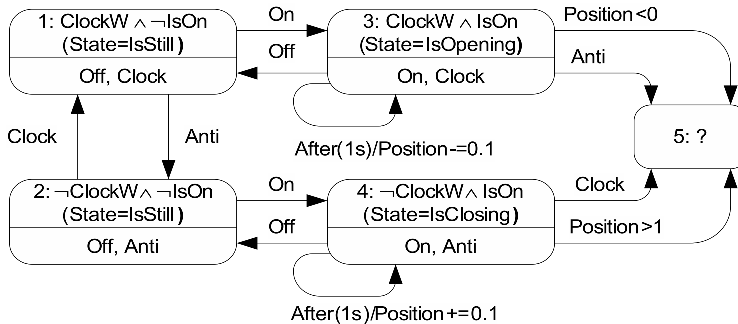

Figure 6.4: Problem domain behavior.

The fundamental components of problem diagrams and problem frame diagrams are domains and phenomena. Therefore, the language has to be able to represent domains and their characteristics according to the domain type (biddable, causal, lexical), as well as the connections and relations between domains.

Domains are mainly characterized in terms of phenomena, which have to be properly represented. For instance, phenomena of biddable domains are events, while those of causal domains are states and events (generalized as causal phenomena) and those of lexical domains are symbolic. For every phenomenon it must be clearly specified by which domain it is controlled and by which domains it is visible. As far as shared phenomena are concerned, it is important that sharing is represented without suggesting any communication policy, such as sending a message or polling for an event occurrence or state change.

Notice that, the same phenomenon may exist in two forms: the real one, which is referenced by the user requirements (e.g., when the heart rate drops below a given threshold, the alarm must be activated), and the sensed one, which is viewed by the machine, and is used in the machine specifications (e.g., when the patient sensor indicates a low heart rate, the machine activates the alarm). The modeling language has to be able to represent both of them, making clear which is the real world phenomenon, and which is the one viewed at the machine interface.

Descriptions can be *indicative*, describing what is given and cannot be changed when developing the solution, or *optative*, describing what has to be achieved by means of the computer-based solution. Both descriptions have to be represented, making clear their differences, possibly by providing different representations.

In PFs, designation of phenomena grounds the problem's vocabulary in the physical world, i.e., it provides names to describe the environment, the system and their artefacts. Therefore a suitable vocabulary must be supported by the modeling language in order to name phenomena.

The modeling language must also support the frame concern. Addressing frame concerns adequately means making requirements, domain and specification descriptions fit together properly and the combination of these descriptions must result in a correctness argument. Thus, the language has to support both the requirements specification and the representation of the correctness argument.

Finally, since complex problems may not fit into a single PF, the language has to support their composition.

6.3 A PFs & SysML based methodology

This section briefly introduces how SysML supports PFs concepts, then it presents a modeling approach based on the integration of PFs and SysML.

6.3.1 SysML support for Problem Frames

In this section the representation of PFs concepts via SysML is addressed.

First of all, problem domains can be represented by means of SysML `bdds` and `ibds`. In fact, blocks can be used to represent domains, and stereotypes can be used in order to denote the different domains type. Ports, interfaces and signals can be used to represent the phenomena shared between domains. Moreover, SysML allows the modeler to distinguish events, calls, and continuous flows of information, so that the resulting description faithfully represents the nature of the real world phenomena.

SysML also supports the distinction of indicative and optative descriptions. We suggest to explicitely separate the description into distinct diagrams. If the modeler considers preferable to keep both kinds of descriptions in a single diagram, he/she should introduce stereotypes. For instance, user requirements can be labeled `«UserRequirement»` while system requirements (intrinsic properties of problem domains) `«PDRequirement»`.

The dynamic properties of the domains can be specified by means of state machine diagrams. When a more declarative style is preferred, "constraints" can be defined and attached to the involved elements in "parametric diagrams".

Interactions between domains (e.g., between the machine and the environment) can be modelled by means of activity diagrams.

Finally, SysML, having been conceived to model potentially complex systems, provides (de)composition mechanisms for all diagrams and elements, thus making possible to structure the system descriptions at different levels of aggregation and abstraction. These mechanisms can be applied in a straightforward way to problem frames as well.

6.3.2 Approach

Our approach inherits the conceptual activities from PFs increasing its effectiveness by means of a suitable notation. Moreover, it supports the integration of formal notations for the definition of properties. This allows one to anticipate the verification activities at the requirements engineering time, reducing the risk of expensive design mistakes.

The methodology is structured into the following steps:

- Problem analysis: the problem context, i.e., the *world* in PFs terminology, is decomposed into *domains*, and *shared phenomena* are identified. *Domains* and *Shared phenomena* are represented by means of SysML Blocks and are defined using a `bdd` showing the entities of the problem context.

- Problem definition: the domain blocks are instantiated and connected one to another using an `ibd`.

- Requirements definition: user requirements and properties associated with domains are defined by means of `req` diagrams.
- Domains refinement: domains are refined using SysML diagrams such as `bdd` and `ibd` to support domains decomposition into simpler structures, and `stm`, `act`, `par` and `seq` diagrams to define behaviors.
- Requirements refinement: user requirements are refined by means of `par` and `stm` diagrams, which provide a more rigorous definition of the properties informally introduced in `req` diagrams

Notice that such activities are not completely independent one from another and the methodology should not be considered an ordered sequence of steps. Exploiting SysML notation, the involved diagrams provide different views of the same model and thus our methodology exploits a sort of model-centric requirements analysis approach.

The rest of the section discusses in more detail the above steps using the Sluice Gate example.

Problem analysis The domains involved in the problem have to be identified and described, so that requirements can be unambiguously associated with domains.

Domains are represented by means of SysML Blocks, i.e, constructs sufficiently expressive and flexible to represent both concrete physical entities and logical domains. Blocks which can be annotated with stereotypes that specify their nature (e.g., Causal, Machine, Biddable). Shared phenomena are represented by means of SysML Ports through which data can flow and Blocks. Standard Ports describe *event* based communication, while Atomic Ports allow one to access the internal states of Domains. Information is typed by means of Interfaces, Signals or Enumerations. More specifically, *Interfaces* describe events by means of operations. *Signals* define events characterized by complex data structures. *Enumerations* represent the states of a domain. Domains and Shared phenomena are represented by means of a `bdd` diagram describing the entities involved in the problem context. The resulting diagram contains 1) a *system* block representing the *world*, which is composed of domain blocks, and 2) blocks representing the types of the shared information. Domain blocks, in turn, define the ports representing the communication channels through which phenomena can be shared.

The domains of the Sluice Gate Controller problem are specified in the (`bdd`) of Figure 6.5. The whole system is defined by means of the `SluiceGateSystem` block and its components. The `SluiceOperator` block describes an operator capable of sending specific commands. Notice that such element is biddable, in fact we cannot constrain the behavior of the operator. The `Gate&Motor` block represents the Causal Domain. Both the gate and the motor are given elements, i.e., they are object of an indicative description. Here we indicate their characteristics that will possibly be involved in the interactions with the controller. The `SluiceController` represents the Machine Domain: the properties of its interaction with the environment must be defined in order to satisfy the user requirements.

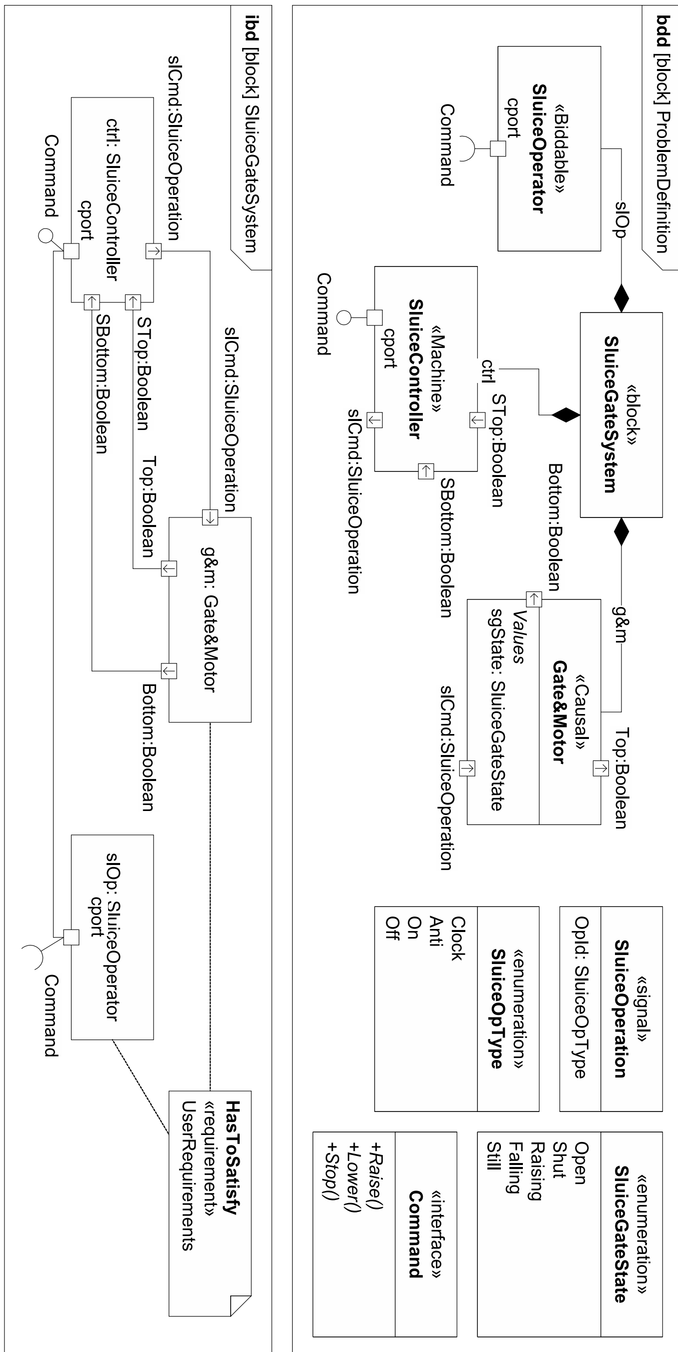

Figure 6.5: bdd, ibd: Representation of the Sluice Gate Control problem.

All the aforementioned domains specify interaction points through which phenomena can be shared. As an example, the `SluiceOperator` requires the `Command` interface to allow the operator to issue commands by invoking the operations provided by the interface. The interface is actually provided by the `Controller` block, which is able to execute the operations defined therein. Similarly, the `Gate&Motor` block defines two atomic output ports that propagate the events generated by the sensors according to the position of the gate. Moreover, the `Controller` defines two dedicated input ports to receive notification by external sensors.

Notice that shared phenomena are specified by defining the types associated with ports and interfaces: for instance, the `SluiceOperation` signal specifies the nature of the commands sent from the Machine to the `Gate&Motor`.

Problem definition PFs Problem Diagrams are represented by means of `ibd` diagrams that define the internal organization of the *system* block. Such diagrams define how the domains are instantiated and connected one to another.

The `ibd` of Figure 6.5 describes the context of the SluiceGate problem showing how the involved domains share information. Notice that it is topologically similar to the original problem diagram shown in Figure 6.2. For instance, the `SluiceOperator` and the `SluiceController` share phenomena that are directly generated and controlled by the `SluiceOperator` and received by the `SluiceController`. `Gate&Motor` and `SluiceOperator` domains have to satisfy the requirements described by the `UserRequirements` requirement diagram.

Requirements definition Requirements are categorizable in two distinct classes: 1) *User requirements*, which specify the user expectations against the solution of the problem, and 2) *System requirements*, which specify properties associated with domains. User requirements express properties that usually involves different domains, specifically they allow one to *User requirements* are defined using a `req` diagram; they are specified by means of informal textual descriptions structured in a sort of hypertext; decomposition and derivation relationships allow one to organize and refine requirements till reaching the desired structure and granularity degree.

Figure 6.6 reports the `req` diagram of the Sluice Gate problem. In order to satisfy user requirements, it is necessary to know the behavior of the causal parts of the problem domain. SysML (implicitly) drives the modeler to describe in a possibly unique requirements diagram both the problem domain *and* the user requirements (i.e., both the indicative and optative descriptions).

The behavior of the causal domain `Gate&Motor` is constrained according to the phenomena generated by the biddable `SluiceOperator` domain: for instance, User requirements specify when commands have to be considered useless, and the expected results in such cases. In Figure 6.6 the top-level block specifies the user requirements in a very abstract way, while the following refinements describe the effects of the commands that the operator can issue.

Once defined, *User requirements* have to be allocated to problem domains by annotating the correspondent blocks of the `ibd`. For instance, as shown in Figure 6.5, `Gate&Motor` and `SluiceOperator` are the involved domains.

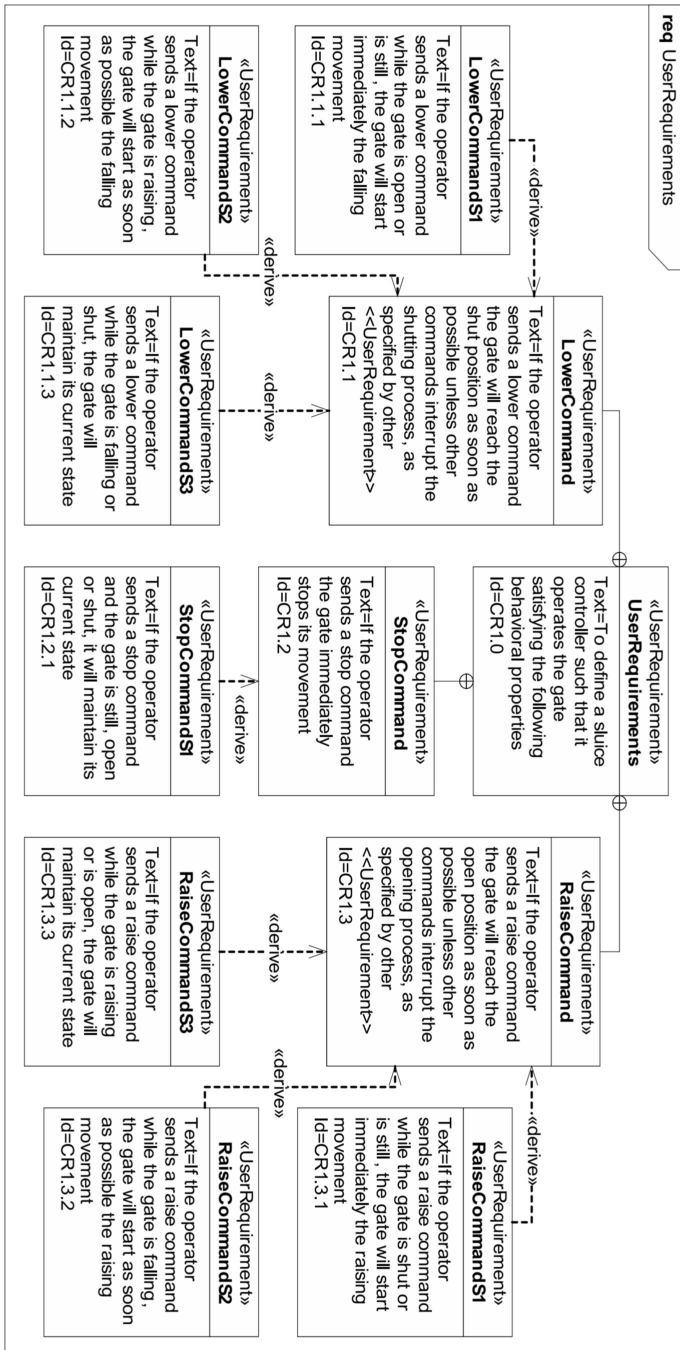

Figure 6.6: `req`: User Requirements decomposition.

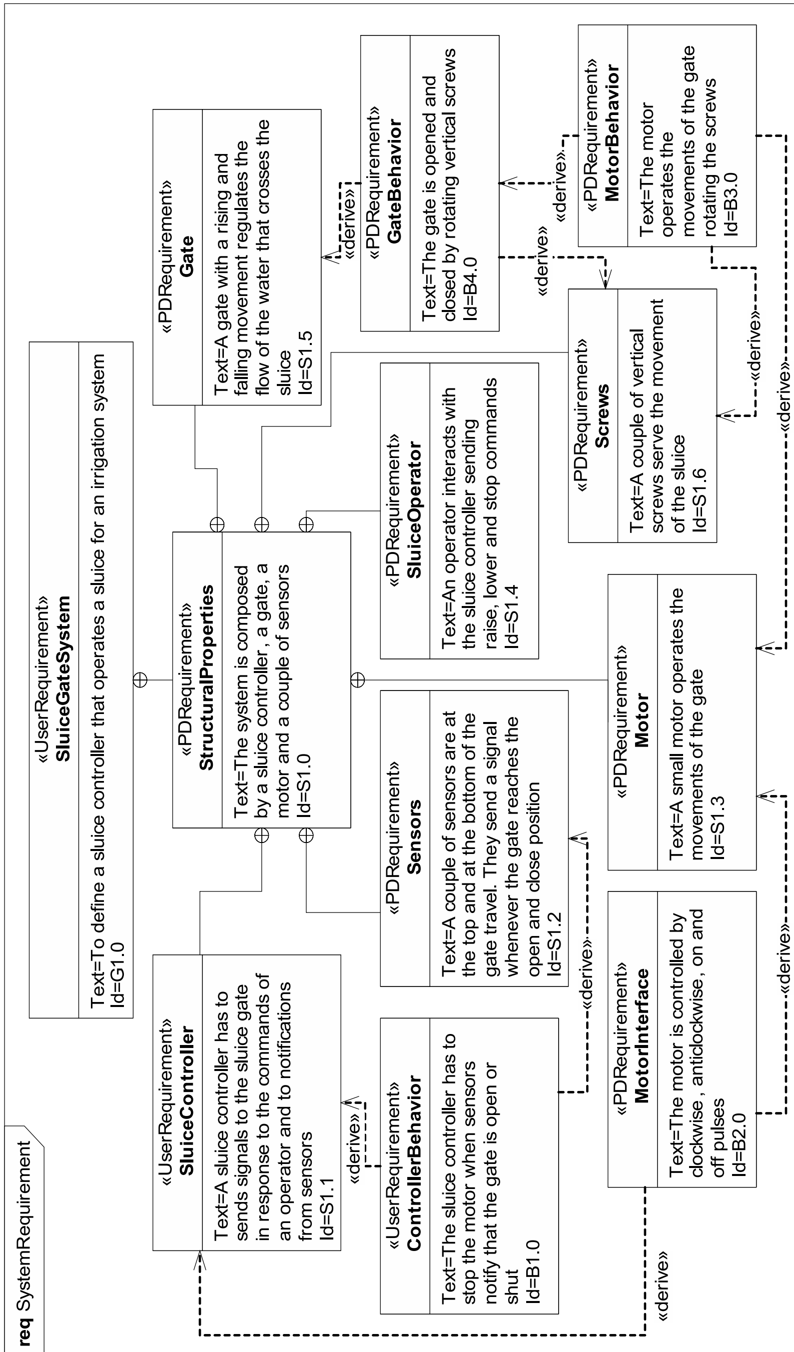

Figure 6.7: `req`: System Requirements decomposition.

System requirements express properties concerning domains. As in the previous case, they can be defined and structured using `req` diagrams. SysML provides different diagrams supporting the definition of a system in a more precise and detailed way. In this way it is possible to informally describe the *world* whenever the construction of the system requires the definition of properties of the causal domain. However, this is not mandatory, since `req` diagrams can be replaced by other diagrams that could better support the environment description as explained in what follows.

Here we do not consider such case, since it is supposed that the `Gate&Motor` domain is given and cannot be modified. Therefore, the "system requirements" in Figure 6.7 are not actual requirements, but just an informal description of the `Gate&Motor` domain.

Refining domains descriptions Domains represent the constituent parts of the *world*. They are introduced in `bdd` and `ibd` diagrams that describe the structural aspects of the problem context at an high level of abstraction. Moreover, domains properties are possibly informally described in the `req` diagram that defines the *System requirements*. Domains descriptions are now refined into more precise and detailed specifications.

System requirements are characterized by static and dynamic aspects, and thus depending on their nature they have to be refined exploiting different diagrams. Structural properties are defined by means of `bdd` and `ibd` diagrams, while behaviors are described by means of `act`, `stm` and `par` diagrams. SysML, having been conceived to model potentially complex systems, provides (de)composition mechanisms for all diagrams and elements, thus making it possible to structure the system descriptions at different levels of aggregation and abstraction.

`bdd` and `ibd` are used to decompose domains into simpler structures. For instance, Figure 6.8 shows the internal structure of the `Gate&Motor`. Domains decomposition follows the same criteria applied for the problem definition (i.e., sub-domains are defined by means of Blocks in a `bdd`, and instantiated and used in a `ibd`, where the connections are shown).

`stm` are used to specify in an operational style the behavior associated with a single Block. For instance, the `stm` of Figure 6.9 specifies the reactions to external commands and the internal evolution of the `Gate&Motor` domain.

`act` are used whenever it is necessary to specify a complex behavior involving several domains. Notice that it is possible to use `act` diagrams in addition to `stm`. Also these diagrams feature an operational style and support the composition of simpler activities. In our example, the coordinated behavior of the Gate and Motor pair is specified by means of an activity diagram, as shown in Figure 6.10.

Refining requirements The `req` diagrams defining the *User requirements* can be used by the analyst to define and organize requirements in an informal top-down approach. However in order to apply automatic verification techniques such properties need to be expressed in a formal way.

Our approach does not impose the usage of a specific formal language for refining purposes. The modeler is free to adopt the language that he/she considers most suitable to the problem. The methodology simply defines *how* formal

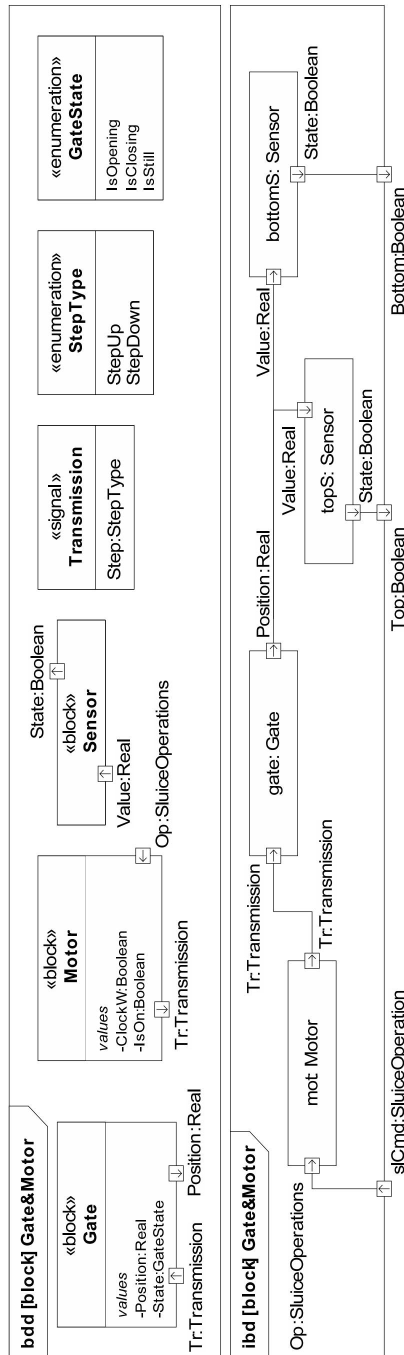

Figure 6.8: `bdd`, `ibd`: Decomposition of the `Gate&Motor` domain.

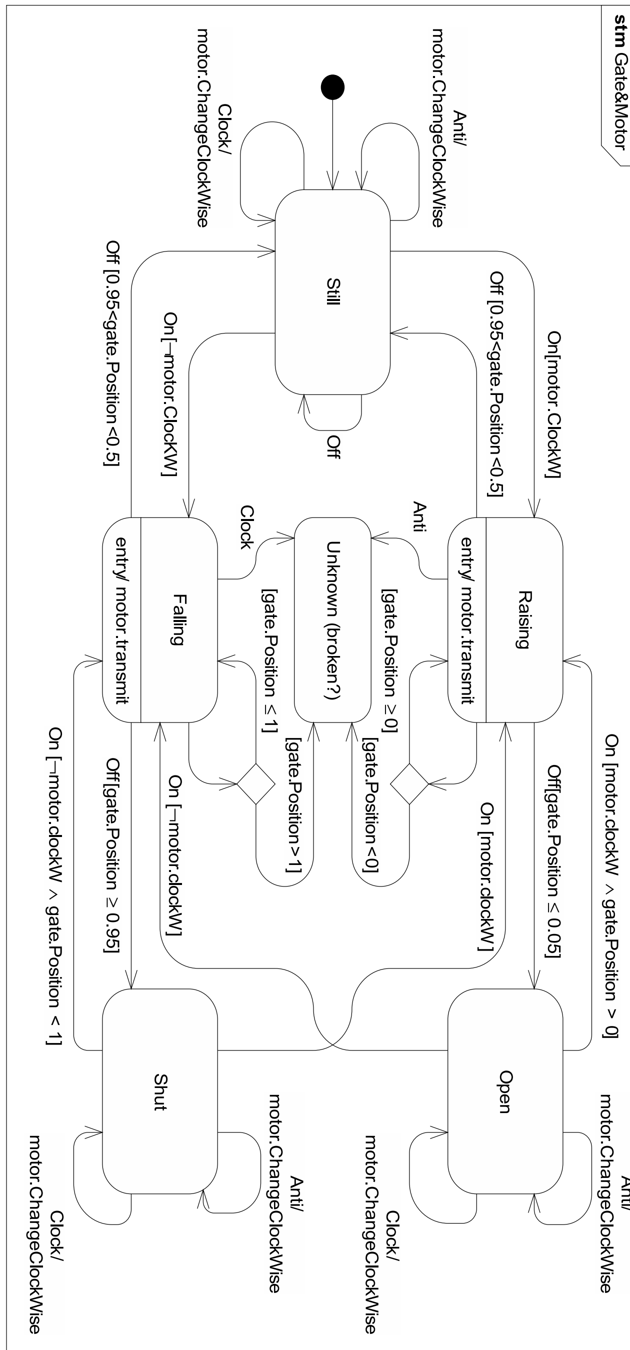

Figure 6.9: `stm`: The dynamics of the Gate and Motor

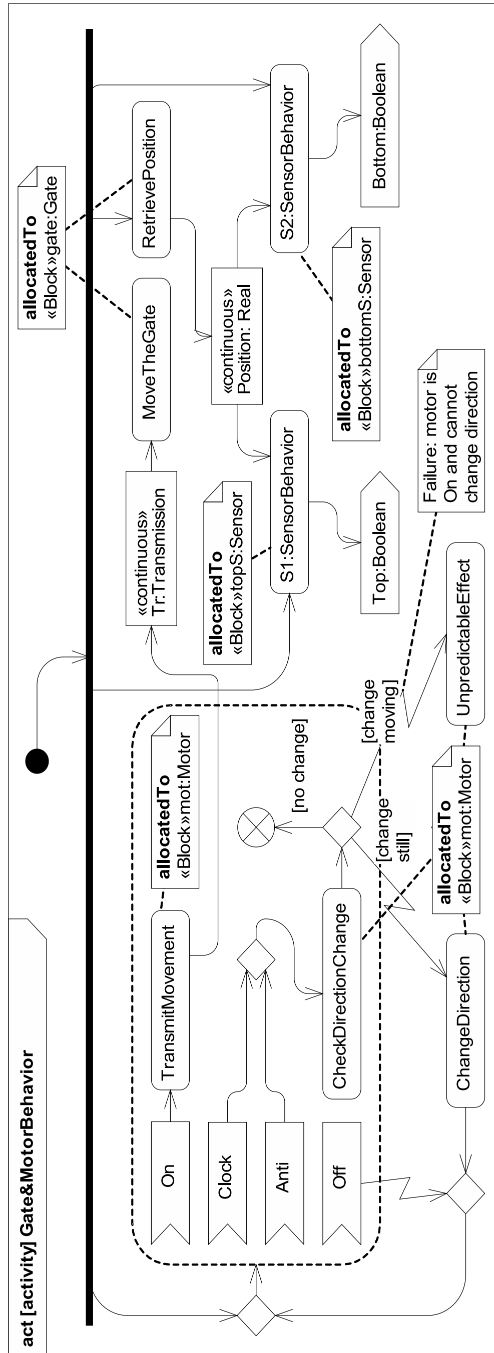

Figure 6.10: `act`: The activities associated with the components of the problem domain

properties have to be specified and integrated in the model.

`bdd` have to formally define properties with `constraint` blocks. Constraints are expressed by composition of parameters. `par` diagrams allocate constraints to one or more (sub)domains. Notice that, the same approach can be used also for refining domain descriptions. Figure 6.11 shows a formalization of a *User requirement* (the `Lower` command) using TRIO, a first order temporal logic for the specification of real-time systems [23].

`stm` represent an effective alternative to the specification of constraints. They are particularly suitable to the definition of properties concerning the evolution of blocks. Their usage is widely supported by different analysis techniques that support formal verifications [10]. A `stm` equivalent to the state machine by Jackson reported in Figure 6.3 could define the *User requirements* concerning the states of the `Gate` block.

6.4 Related work

The proposed approach for integrating SysML and PFs can be seen from two perspectives: 1) providing SysML with a rigorous conceptual framework and a set of methodological guidelines, and 2) providing PFs with a suitable notation.

In the former case, there are no proposals concerning how to use the language. In fact, the literature reports just a few experiences in using SysML. For instance, some whitepapers are available from Artisan, describing systems like a house heating system [48] or a waste treatment plant [28]. However, these papers describe the nature and the role of SysML diagrams, rather than suggesting methodological guidelines.

A SysML modeling approach for real-time systems is proposed in [14]. An experimental application of SysML in the real-time domain showed that SysML supports well the definition/usage dichotomy (e.g., concerning the `bdd`/`ibd` and `bdd`/`par` pairs), the hierarchical decomposition of behavioral and static elements, and the usage of cross-cutting constructs (namely requirements and allocation mechanisms). In addition, the paper discusses the limits of the language, mostly related to the lack of formal semantics and mechanisms to extend and adapt the language. SysML inherits the extension mechanisms provided by UML. Tagged values, constraints and stereotypes allow the user to extend the expressiveness of the SysML basic profile through the definition of new constructs. However, it is not possible change the semantics of the basic constructs (for details see [14]). However, these limits can be partially overcome by using formal languages for defining constraints and properties. This possibility leads to the development of formal methods applicable to SysML models (e.g., analysis methods and tools to verify the correctness and the consistency of a model).

In order to put the usage of SysML in context, it should be noted that the systemic perspective supported by SysML can address two possible situations:

- The required development concerns only the software part of the system, the rest being given. In fact, in our example the problem is just to develop the software machine satisfying the user requirements, while the problem domain is completely fixed.

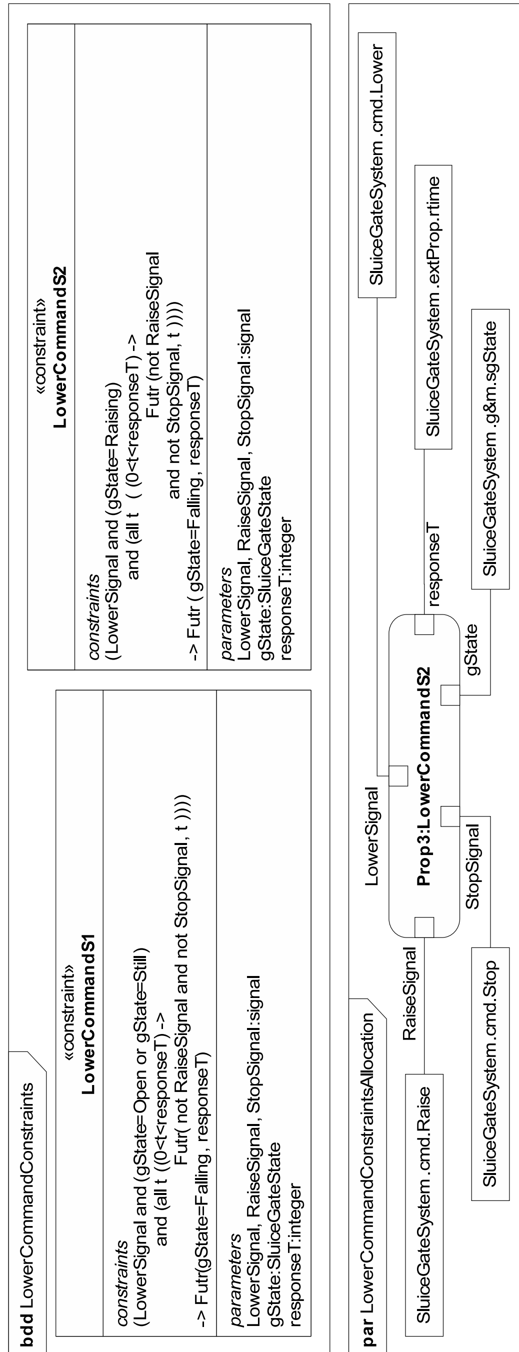

Figure 6.11: `bdd`, `par`: Formal specification of the Lower Command requirement.

- The required development concerns the whole system, i.e., both the non-software part and the software part are subject to definition and implementation.

The problem frames approach applies well to the former case, which is the one usually addressed by software developers. The approach described in this chapter also applies to such a case.

The original proposal of the PFs approach [37] already addressed the methodological issues of requirements engineering. On this basis, several researchers built extensions and refinements of the method. Some focused on the adaptation for specific purposes, like web service requirements modeling [39], or e-business modeling [4]. Others studied the transition from requirements to machine specifications [45, 65]. Jackson and co-authors also elaborated on the PFs approach in several directions, such as defining the semantics of PFs [27], composing PFs [41], and highlighting the role of PFs in software engineering [38]. However, none of the mentioned works concerned the linguistic issues, nor the integration of PFs in any popular model-based development practice.

Chapter 7

A SysML based PFs catalogue

According to the Problem Frames approach, complex problems need to be decomposed into simple problems that can be represented and solved separately.

The problem decomposition can take advantage of problem classes, i.e., given recurrent patterns shared by different problems. Jackson proposes five basic Problem frames [37] that differ by their requirements, domain characteristics and frame concern.

Basic problem frames are introduced with the idea that, at decomposition time, once the appropriate problem frame is identified, the associated analysis method is predefined and easy to apply.

This chapter aims at testing the effectiveness of the previously approach by applying it to the modeling of the catalogue of basic Problem Frames proposed in [37]. Although the intrinsic complexity of the proposed problems is relatively low, such basic frames play a fundamental role, since the whole decomposition process of PFs aims at reducing complex problems to a set of simple problems that should fit these frames.

Each problem frame is analysed by using the the SysML-PFs based approach and illustrated by means of SysML diagrams that show the characteristics and the behavior of the involved domains, the requirements and the machine specification.

7.1 Required behavior: one way traffic lights

This section presents the analysis of the One way traffic lights problem, a simple example proposed by Jackson in [37] to describe the Required behavior frame.

> *When a section of road is being repaired, it's often necessary to enforce one way traffic. Half of the road width is used for traffic and half for the repair work. The traffic is controlled by a pair of simple portable traffic light units.*
>
> *The repairers put one unit at each end of the one-way section, and connect it to small computer that controls the sequence of lights. Each unit has a Stop light and a Go light. The computer controls the lights*

by emitting RPulses and GPulses, to which the units respond by turning the lights on and off. The regime for the lights repeats a fixed cycle of four phases. First, for 50 seconds, both units show Stop; then, for 120 seconds, one unit shows Stop and the other Go; then for 50 seconds both show Stop again; then for 120 seconds the unit that previously showed Go shows Stop, and the other shows Go. Then the cycle is repeated.

The corresponding Required behavior problem frame is reported in Figure 7.1.

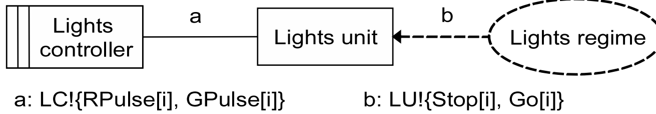

Figure 7.1: The one way traffic lights controlled behavior frame

Problem context definition According to the proposed approach, the first step of the analysis consists in identifying and describing the domains and phenomena involved in the problem, and in defining the architecture of the problem context.

The problem context is defined by means of the block definition diagram reported in Figure 7.2.

The problem is characterized by two distinct domains: *Lights controller* and *Light unit*. The former is a *Machine* domain representing the *Control machine* of the Required behavior frame, while the latter is a *Causal* domain representing the *Controlled domain* of the same basic frame. *Light unit* is characterized by two Boolean attributes named *Stop* and *Go* representing the state of the red and green lamps, respectively.

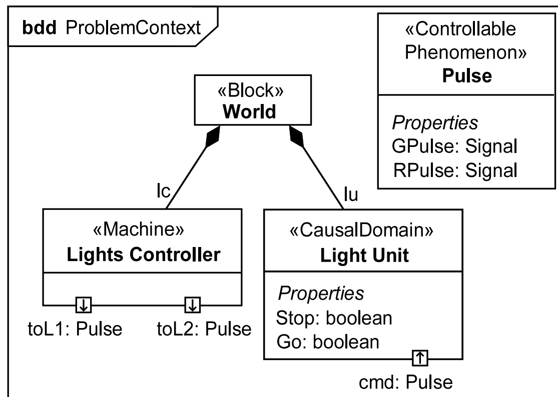

Figure 7.2: The `bdd` representing a Lights Controller and two Light Units

The lights controller manages the light units sending two signals of type *Pulse*: *RPulse* to turn on the red light and *GPulse* for the green light. Pulses are shared

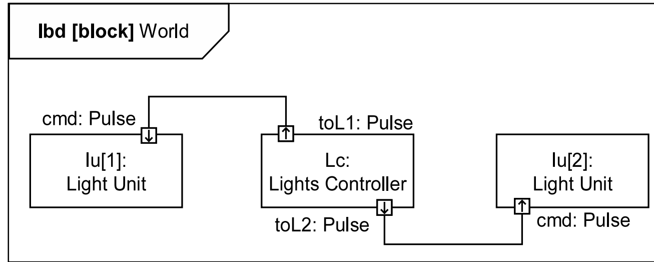

Figure 7.3: The `ibd` representing the structure of the problem domain

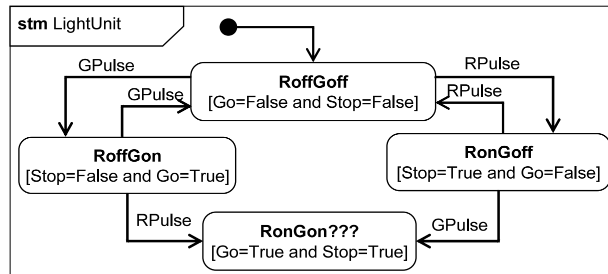

Figure 7.4: The `stm` specifying the behavior of light units

phenomena controlled by the machine: they are represented in the `bdd` by means of stereotyped data flowing through Flow Ports (*toL1*, *toL2* and *cmd*).

The structure of the problem context is described by the `ibd` of Figure 7.3, where the domains introduced in the `bdd` are instantiated and connected among them through flow ports.

Problem domain modeling The next step of the approach consists in defining the behavioral aspects of the problem domains.

The light units behavior is described by the `stm` diagram reported in Figure 7.4. *Light unit* is characterized by four distinct internal states named *RoffGoff*,*RonGoff*,*RonGon???* and *RoffGon* representing all the combinations of the values of *Stop* and *Go*. Notice that although *RonGon???* is an internal state of *Light unit* that can be reached by different transitions, *Light controller* should prevent to reach such state. State transitions depend on the external events *GPulse* and *RPulse* received from the light controller via the flow ports. Note that, as in UML, whenever a state S does not have any outgoing transition labelled with an event X, this means that event X has no effect in state S. For instance, if *GPulse* occurs while the light unit is in state *Go*, it is simply dropped.

Requirements specification The next step of the approach consists in the specification of the requirements.

It is required that the system of lights behaves according to a continuously repeated cycle, which is composed of a sequence of 4 phases regulated by temporal

events as specified by the state machine in Figure 7.5.

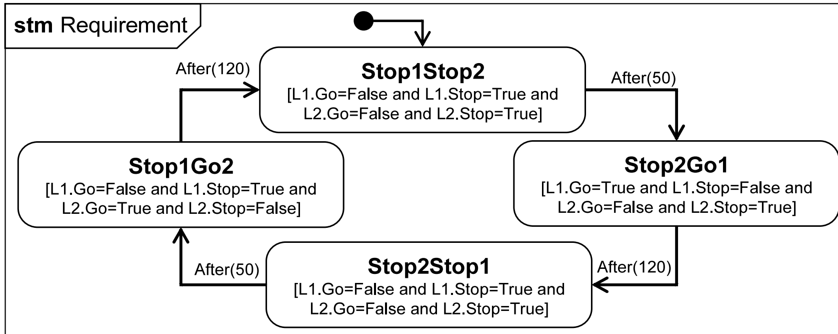

Figure 7.5: The `stm` representing the required behavior of the two light units

Notice that the `stm` refers to the internal state of both lights units involved.

Machine specification The final step of the approach consists in the specification of the machine.

The machine behavior is represented by the action Generate Pulses shown in the `act` diagram of Figure 7.6.

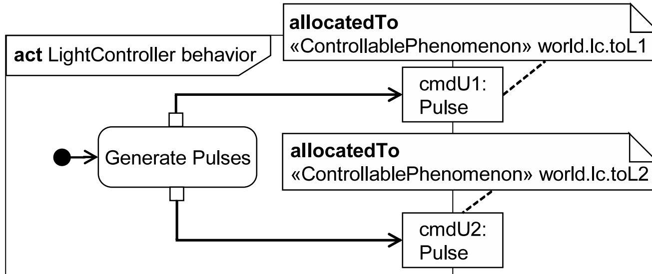

Figure 7.6: The `act` diagram representing the behavior of the lights controller

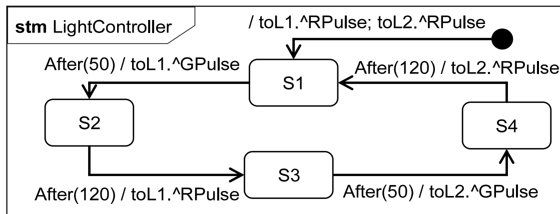

Figure 7.7: The `stm` diagram representing the behavior of the lights controller

Notice that such description is not expressive enough to represent the specification of the behavior and therefore it needs to be refined. For instance, we could

add constraints to this black box representation or we could complement the activity diagram with a state machine diagram, which specifies the pulse generation timing. The `stm` diagram of Figure 7.7 shows the latter.

The transitions generate signals GPulse and RPulse that are sent to the interested domains via the Flow Ports *toL1* and *toL2*.

7.2 Information display: odometer display

This section presents the Odometer display problem, a simple example proposed by Jackson in [37] to illustrate the Information display frame.

> *A microchip computer is required to control a digital electronic speedometer and odometer in a car.*
>
> *One of the car's rear wheels generates pulses as it rotates. The computer can detect these pulses and must use them to set the current speed and total number of miles travelled in the two visible counters on the car fascia. The underlying registers of the counters are shared by the computer and the visible display.*

With respect to the problem proposed by Jackson [37] our case is slightly different, because of the presence of a shared phenomenon that represents the continuous tension generated by the speed sensors allocated to the wheels of the car.

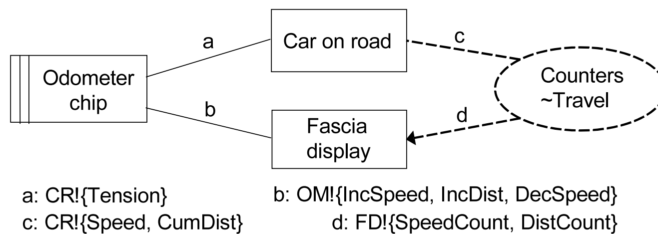

Figure 7.8: The odometer display: an information display problem frame

Problem context definition The system is composed of the machine Display controller and the causal domains *Car on road* and *Fascia display*, as specified by the `bdd` reported in Figure 7.9.

Display Controller represents the *Information machine* of the Information display frame, while *Car on road* and *Fascia display* represent the Causal domain *Real World* and *Display* of the same pattern.

Car on road controls a phenomenon *A* representing the continuous tension generated by the internal speed sensors. *Display Controller* controls a phenomenon *B* representing the commands sent to the display in order to update the current values of both speed and amount of cumulated distance. The constraint in the causal domain *Car on road* specifies that the tension is proportional to the speed.

The problem context structure is defined by the `ibd` of Figure 7.10.

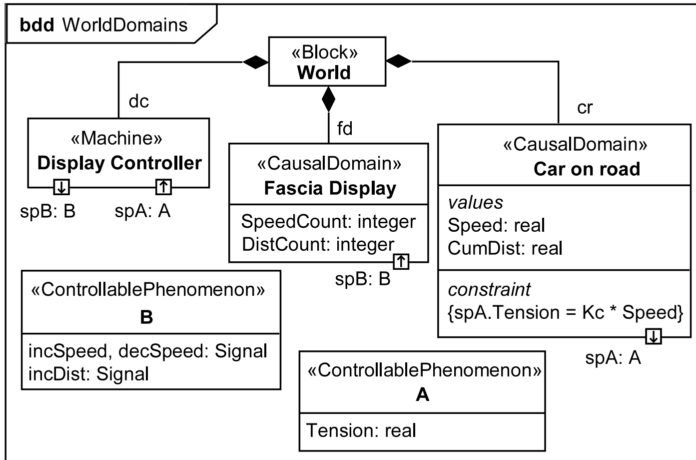

Figure 7.9: The odometer display: problem context

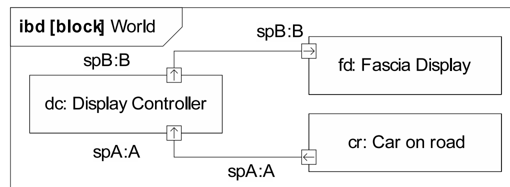

Figure 7.10: The odometer display: the `ibd` representing the structure of the problem context

Problem domain modeling The behavior of the car is described in Figure 7.11 by an `act` diagram.

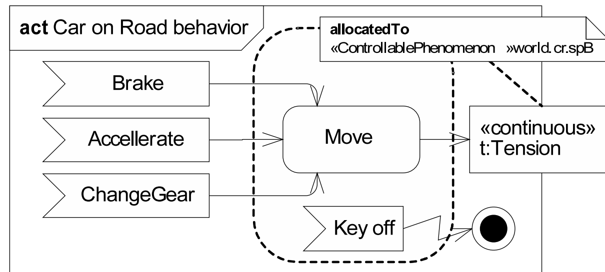

Figure 7.11: The `act` diagram specifying the behavior of the car

Activity *Move* generates a tension that is proportional to the current speed of the car and it is allocated to the output Flow Port *spB* and, through this port, it is made continuously visible to the display controller. The behavior of *Move* is influenced by the state of the car. *Move* determines the current speed, which is transformed into a tension. The behavior of the fascia display is defined by means of the activity diagram in Figure 7.12.

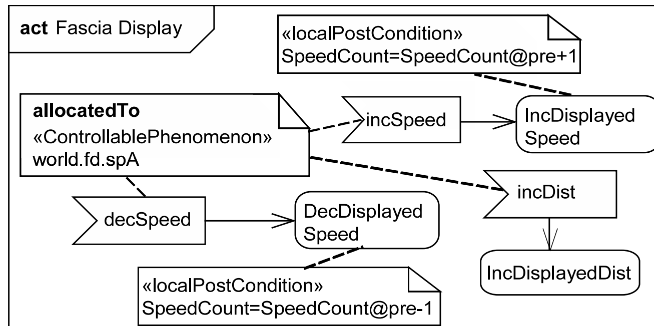

Figure 7.12: The `act` diagram specifying the behavior of the display.

The internal behavior is described by means of two activities that update the displayed speed and the displayed cumulated distance, respectively. Activities are triggered by external signals allocated to the input Flow port *spA* and received by the controller.

Requirements specification Let us now specify the required value of *SpeedCount* as a function of the actual *Speed*. The specification is carried out in two steps: first we define the value to be displayed (*STBD*), then we take into account that a (given, little) delay in displaying the speed is acceptable. Since the behavior of the display is time dependent, we choose to express it by means of TRIO [23] formulas according to the guidelines illustrated in Chapter 3. The following TRIO formulas define *STBD*:

$$UpToNow_e(STBD(X))\ and\ Speed(Y)\ and\ |Y - X| \leq K\ \rightarrow\ STBD(X)$$

$$UpToNow_e(STBD(X))\ and\ Speed(Y)\ and\ |Y - X| > K\ \rightarrow$$

$$STBD(Z)\ and\ |Y - Z| < K$$

The first formula states that if *STBD* has been equal to *X* during an (even very small) interval, and now *speed* equals *Y*, and the distance between *X* and *Y* is less than the acceptable precision, then *STBD* must not change. The formula defines two properties: that the difference between the displayed value and the real value must not be greater than *K*, and that small changes in the actual speed do not cause changes in the displayed value. The second formula states that if the actual speed becomes significantly different from the displayed value, then the latter must be updated with a value that makes such a difference less than or equal to *K*. Now we define the *SpeedCount* as a function of *STBD*: if the need to display value *X* arises at time *t*, then *X* is actually displayed not later than *t+Delay*.

$$STBD(X) \rightarrow Futr(SpeedCount(X), t1)\ and\ t1 \leq Delay$$

$$SpeedCount(X) \rightarrow Past(SD(X), t1)\ and\ t1 \leq Delay$$

The second formula is needed to express that all the displayed values correspond to the sensed speed of the car (i.e., no spurious values are displayed).

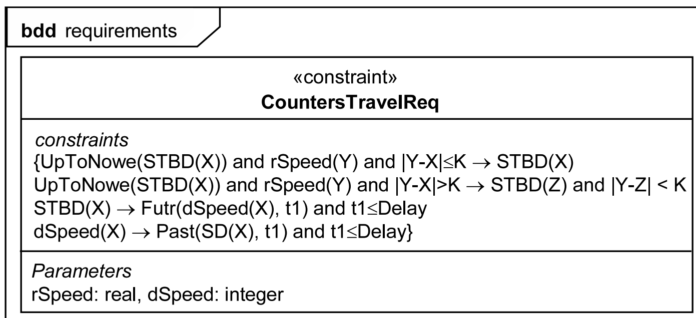

Figure 7.13: The constraint specifying speed reporting

In order to complete the specification of the requirement, we have to specify that the initial values of *Speed*, *SpeedCount*, and *STBD* are all zero. Since this part of the specification is rather trivial, it is not reported here. The requirements reported above are expressed in SysML by means of the constraint reported in Figure 7.13. The specification of the requirements concerning *DistCount* is similar.

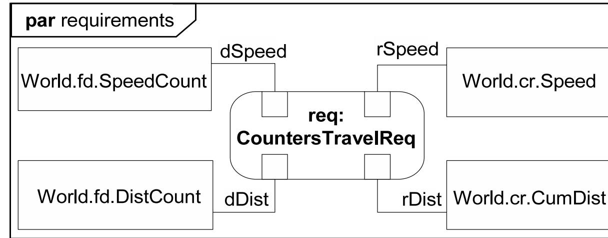

Figure 7.14: The `par` diagram describing the allocation of the requirements CountersTravelReq to the problem domains

Machine specification The machine specification is in three parts. The first defines the speed perceived by the machine (PS): it is proportional to the *Tension* received from the wheel sensors:

$$PS = C_k\ Tension$$

The second part defines how the integer speed to be displayed ($PSTD$) depends on the speed perceived by the machine (PS). The relation between $PSTD$ and PS is very similar to the one between *STBD* and *Speed*, described in the requirements specifications:

$$UpToNow_e(PSTD(X))\ and\ PS(Y)\ and\ |Y - X| \leq K \rightarrow PSTD(X)$$

$$UpToNow_e(PSTD(X))\ and\ PS(Y)\ and\ |Y - X| > K \rightarrow PSTD(X + 1)$$

$$UpToNow_e(PSTD(X))\ and\ PS(Y)\ and\ |X - Y| > K \rightarrow PSTD(X - 1)$$

Finally, we have to specify how the $PSTBD$ values determine the need to issue *IndSpeed* or *DecSpeed* signals. The following formulas state that *IncSpeed* (*DecSpeed*) is issued whenever $PSTBD$ increases (decreases).

$$ForEach(X, IncSpeed\ \leftrightarrow (PSTD(X)\ and\ UpToNow(PSTD(X - 1))))$$

$$ForEach(X, DecSpeed\ \leftrightarrow (PSTD(X)\ and\ UpToNow(PSTD(X + 1))))$$

Note: the specification above is based on the hypothesis that the machine is able to immediately detect when *Y-X* becomes greater (or less) than *K*. In such a case, the new value of $PSTBD$ is always the following (or preceding) integer. If the machine is slower with respect to the acceleration of the car, or if $K>1$, then we should change the specifications accordingly. The specifications reported above can be included in a constraint and parameterized by means of a `par` diagram as we did for the requirements in Figure 7.13 and Figure 7.14.

A fundamental task of requirements specification based on problem frames consists in addressing frame concerns. It is therefore necessary to evaluate how well SysML models built as described above support the frame concern task. The main steps of the process for the odometer display problem are reported in Figure 7.15. Notice that such diagrams illustrate how the different SysML diagrams are used.

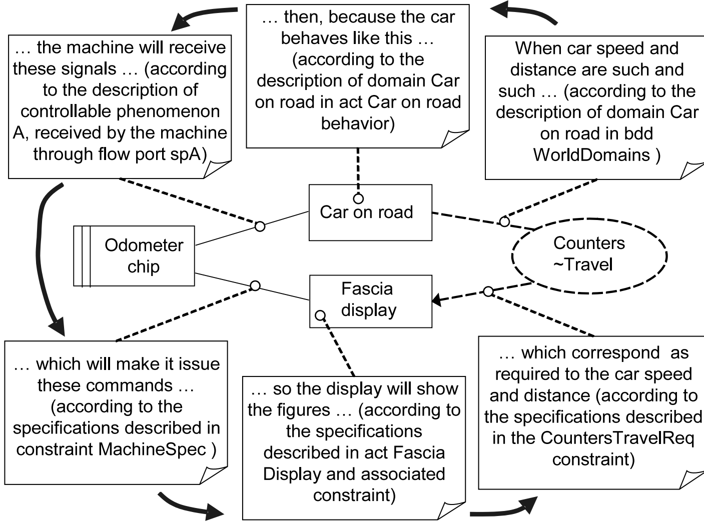

Figure 7.15: Display frame concern

7.3 Commanded behavior: sluice gate control

This section presents the Sluice gate control problem, a simple example proposed by Jackson in [37] to illustrate the Commanded Behavior frame.

> *A small sluice, with a rising and falling gate, is used in a simple irrigation system. A computer system is needed to raise and lower the sluice gate in response to the commands of an operator.*
>
> *The gate is opened and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by clockwise, anticlockwise, on and off pulses. There are sensors at the top and bottom of the gate travel; at the top it's fully open, at the bottom it's fully shut.*

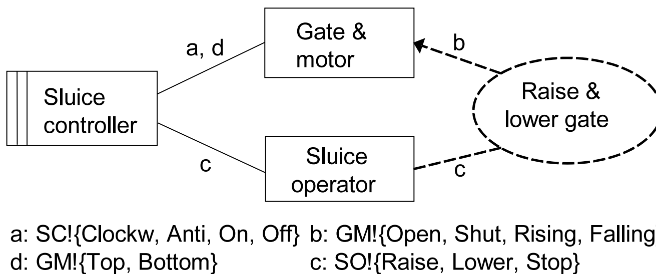

Figure 7.16: The commanded behavior problem frame diagram for the sluice gate controller

Notice that although this example is the same used in the previous chapter to illustrate the PFs-SysML based analysis approach, the problem is described by using different diagrams in order to show how SysML can be easily adapted to different modeling styles and analysis strategies.

The Problem diagram for the Sluice gate control problem is shown in Figure 7.16.

Problem context definition The Problem context is described by the bdd of Figure 7.17. The digram shows the *world* as composed of a *Machine* domain *Sluice controller*, by a *Causal* domain *Gate&motor* and by a *Biddable* domain *Sluice Operator*. Notice that such domains represent instances of *Control machine*, *Controlled domain* and *Operator* of the Commanded behavior frame.

The structure of the problem context is illustrated in Figure 7.18.

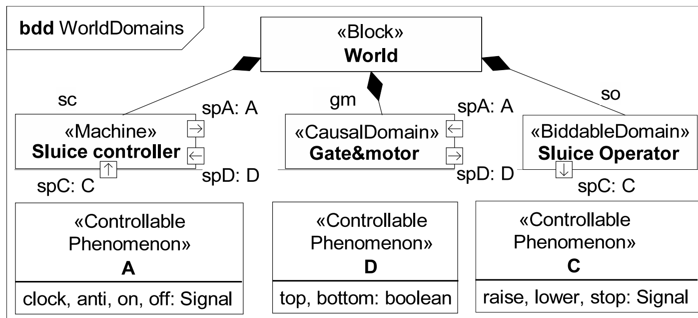

Figure 7.17: The sluice gate controller: context diagram

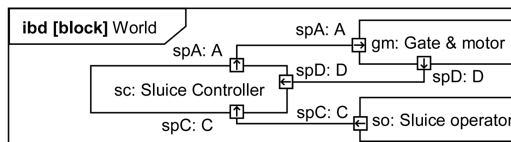

Figure 7.18: The structure of the problem domain

Problem domains modeling *Gate&motor* is in turn composed of other causal domains. The bdd of Figure 7.19 defines the characteristics of such domains. Note that the descriptions of the blocks are equipped with constraints that define some relevant properties of the domains.

For instance, the position of the gate as a function of the *Transmission* value is defined by the constraint in the block *Gate* . Similarly, the transmission is defined as a function of the state of the motor, and the state of the sensors is defined as a function of the gate position.

The structure of the *Gate&motor* domain is described by the ibd in Figure 7.20. Commands from the controller flow through the *spA* port, while the

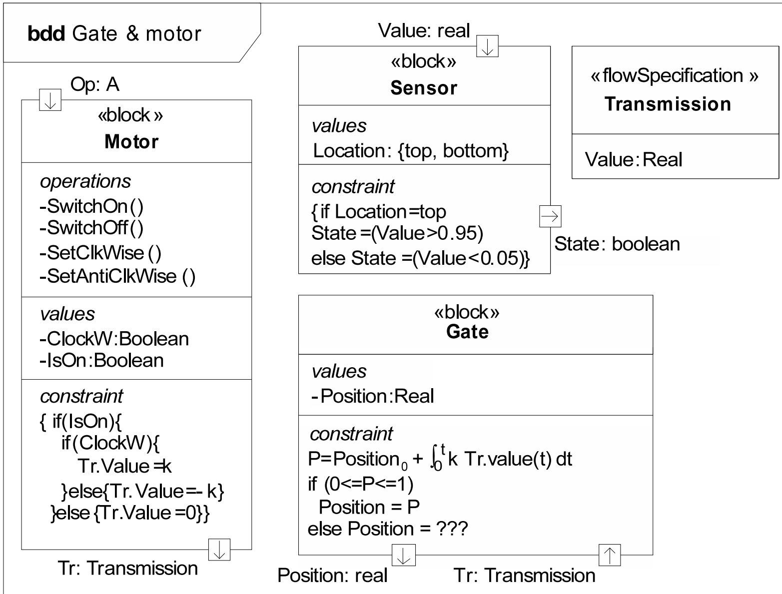

Figure 7.19: Elements of the Gate&Motor domain

state of the sensors that monitor the opening state of the gate is continuously notified via port *spD*.

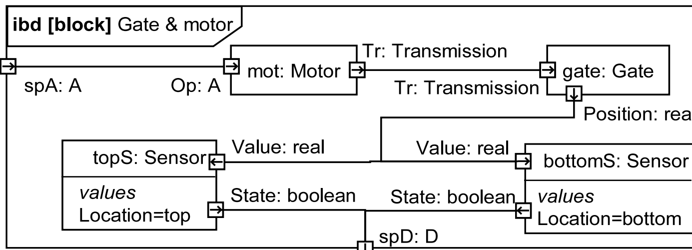

Figure 7.20: The structure of the Gate&Motor domain

Domain behavioral aspects are defined by using `act` and `stm` diagrams. The `act` diagram in Figure 7.21 defines the behavior of the domain *Gate&motor* in terms of activities and flows of information.

The `stm` diagram of Figure 7.23 provides a complementary view of the internal behavior of the domain *Gate&motor*.

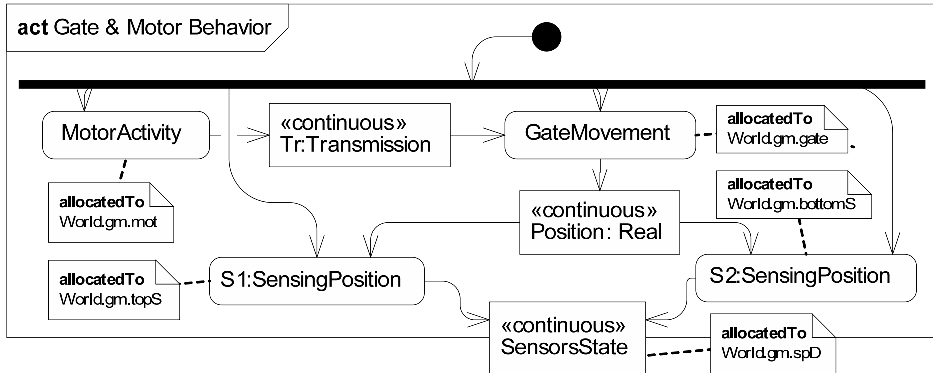

Figure 7.21: The behavior of the gate and motor

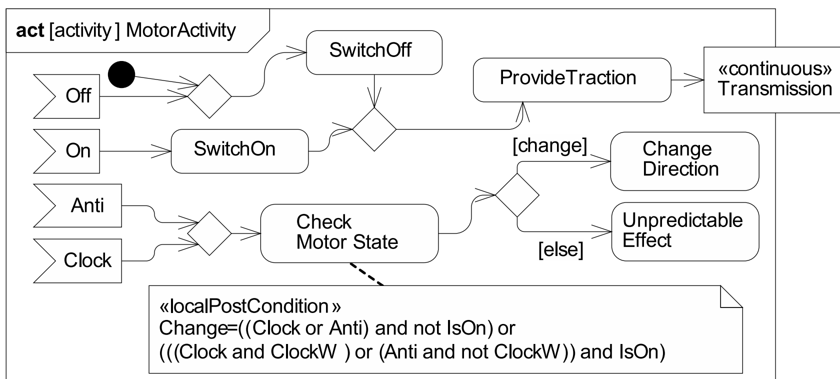

Figure 7.22: The `act` representing the behavior of the motor

Notice that state transitions are activated by signals that come from both internal components (e.g. the motor) and the controller machine. (i.e. allocated to input ports). Transitions in turn trigger the execution of the activities defined in the `act` diagram of Figure 7.21.

The behavior of the motor is described by the `act` diagram in Figure 7.22 and it is characterized by the activities *ProvideTraction* (which generates a continuous flow of data representing the transmission of the motor to the gate), *ChechMotorState* and *ChangeDirection*, and is regulated by the events *On*, *Clock*, *Anti* and *Off* allocated to the *spA* input Flow Port. Note that the existence of unknown states (as defined in [37]) is specified in a `bdd` (Figure 7.19) for the *Gate*, in an `act` diagram (through the *UnpredictableEffect* activity in Figure 7.22) for the motor.

Requirements specification User requirements are specified by means of the `stm` diagram in Figure 7.24. The `stm` describes the expectations of the user in terms of which states the gate has to reach as a consequence of the commands issued by the sluice operator.

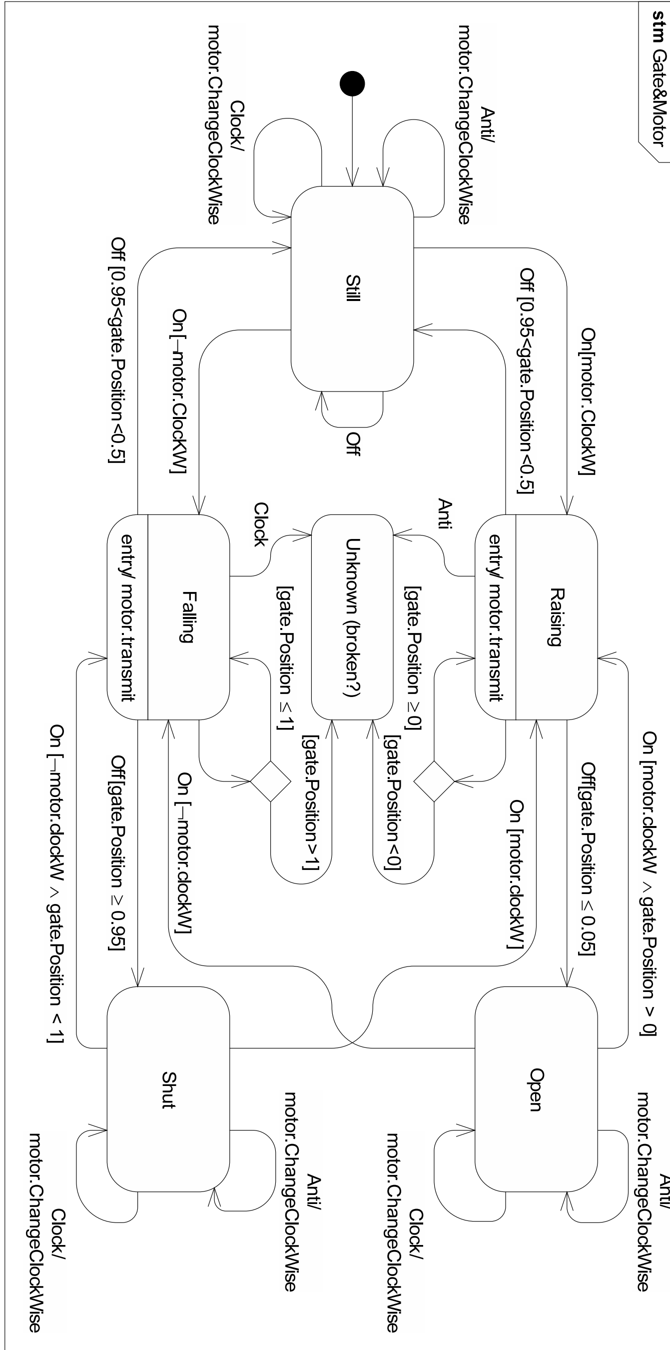

Figure 7.23: The behavior of the gate and motor by means of a `stm` diagram

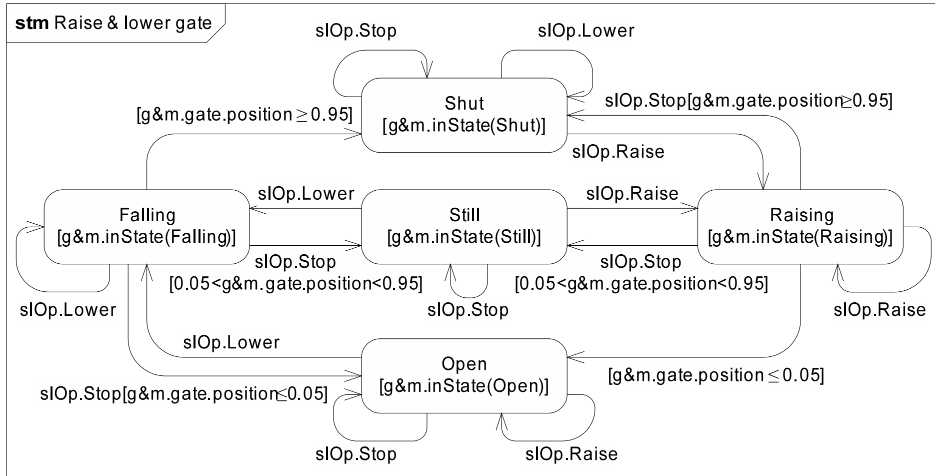

Figure 7.24: The state machine representing user requirements

Machine specification Finally the specification of the machine *Sluice controller* is proposed in the `stm` diagram shown in Figure 7.25.

Notice that such specification describes how, starting from the current state of the sensors, the commands sent by the operator are converted into command pulses for the motor that operates the gate.

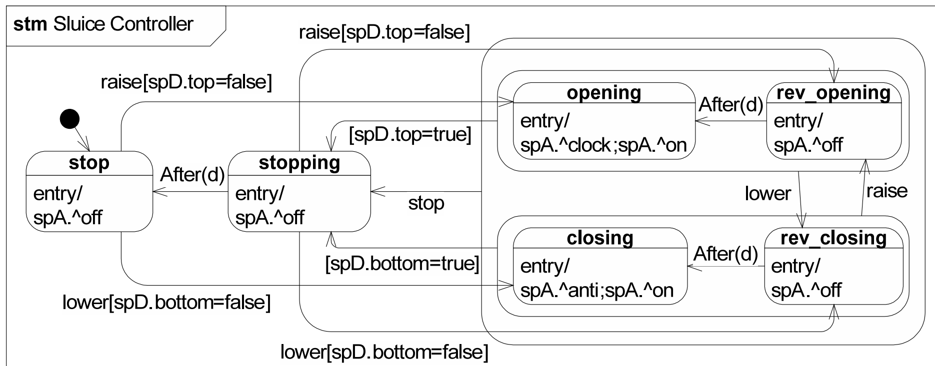

Figure 7.25: The specification of the sluice gate controller

7.4 Simple workpieces: Party plan editing

This section presents the Party plan editing problem, a simple example proposed by Jackson in [37] to illustrate the Simple workpieces frame.

> *Lucy and John need a system to keep track of the many parties they give and the many guests they invite to them. They want a simple*

editor to maintain the information, which they call their party plan. Essentially, the party plan is just a list of parties, a list of guests, and a note of who's invited to each party. The editor will accept command-line text input, in a very old-fashioned DOS or Unix style. To begin with, at least, we are not concerned with presenting or printing the information, just with creating or editing it.

The Problem diagram of the proposed problem is shown in Figure7.26.

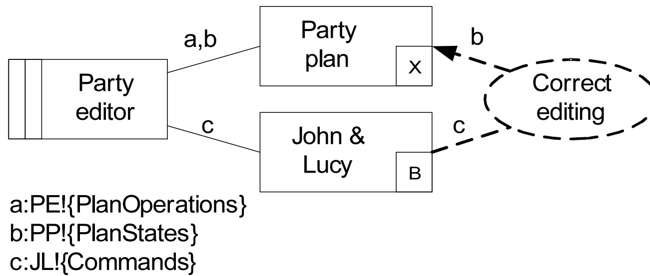

Figure 7.26: The problem diagram for the Party plan editing problem

Problem context definition According to the `bdd` of Figure7.27, the problem context is composed of the machine *Party editor*, the lexical domain *Party plan* and the biddable domain John and Lucy. *Party editor* has to operate the data repository *Party plan* according to the commands issued by *John and Lucy*. Notice that *Party editor* is an instance of *Editing tool* of the Simple Workpiece problem, while *Party plan* and *John and Lucy* are instances of $Workpiece$ and $User$, respectively.

The Problem diagram of the proposed problem is shown in Figure 7.27.

Shared phenomena are defined by means of interfaces and data that can flow through Flow Ports.

A is an interface with a set of operations that update the current state of *PartyPlan*. It introduces some operations that allow the editor to add a new party, a new guest, or to specify that a certain guest is invited to a party. It also provides operations that allow the editor to remove guests, parties and invitations from the party plan.

B is an interface that has been defined to access the internal state of the party plan. More specifically, it provides operations that check whether a certain party or guest is stored in the party plan, and an operation that verifies whether a guest is invited to a party.

C represents commands that can be issued by *John and Lucy* to *Party editor*. Commands are executed into a specific context. Three different contexts, i.e., the party context, the guest context and the plan context are defined for the interpretation of commands. The context can be set by means of dedicated commands.

The commands are strings compliant with the following format:

- `EG gname` set the current context to context of the guest named `gname`

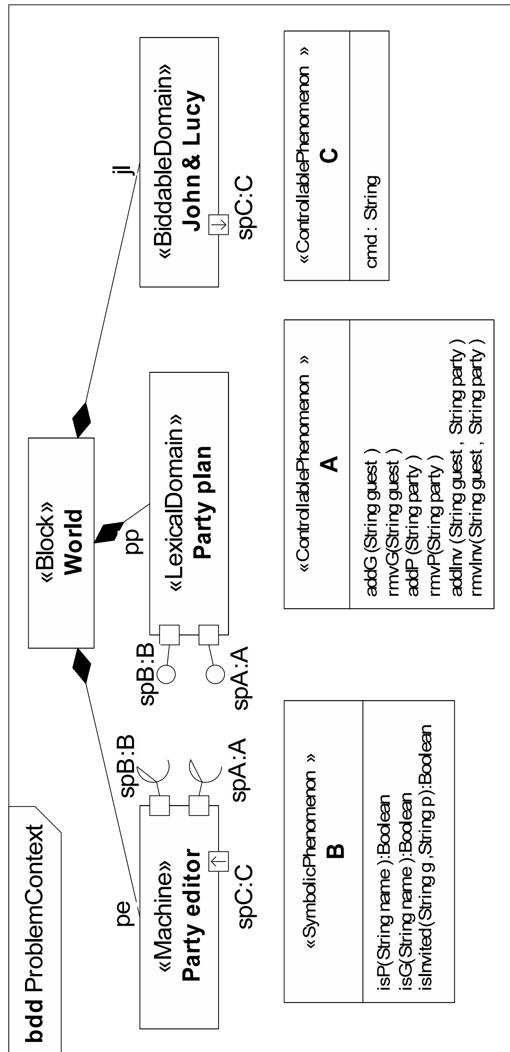

Figure 7.27: The Party plan editing problem: context diagram

- `EP pname` set the current context to the context of the party named `pname`
- `E` set the current context to the context of the party plan
- `G gname` adds a guest named `gname` to the plan or *Party editor* to particular party
- `XG gname` remove the guest named `gname` from the plan or from a particular party
- `pname` adds a party named `pname` to the plan or to the set of parties to which a particular guest is invited
- `XP pname` removes the party named pname from the plan or from the set of parties to which a particular guest is invited

Problem domain modeling *Party plan* is characterized by an internal structure described by means of the `bdd` diagrams shown in Figure 7.28.

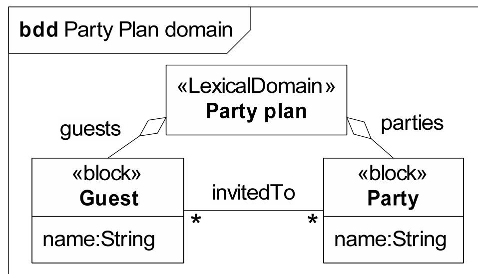

Figure 7.28: The internal structure of the problem domain Party Plan

Party plan stores a collection of guests and a collection of parties, and specifies for each guest a list of parties to which he/she is invited, and for each party it specifies the set of invited guests.

The behavioral aspects of the domain are represented by the operations, defined by the interfaces A and B, implemented by the domain block.

The specification of the behavior is provided in a declarative style by means of OCL statements. For each operation pre and post conditions are defined to describe under which constraints each operation can be invoked and what are the effects of their execution. Constraints involve operations of the interfaces A and B. In the following, some of the most interesting constraints are illustrated.

In order to add a new Party to Party plan, it is necessary that such a party has not to be already stored.

```
context PartyPlan::addP(String party)
  pre:  self.parties->forAll(p|p.name<>party)
  post: self.parties->exists(p|p.name=party)
```

In order to specify that a certain guest is invited to a party both the guest and the party need to be already stored.

```
context PartyPlan::addInv(String guest, String party)
  pre:  self.parties->exists(p|p.name=party) and
          self.guests->exists(g|
    g.name=guest and g.invitedTo->forAll(ip|
              ip.name<>party))
  post: self.guests->exists(g|
          g.name=guest and g.invitedTo.exists(ip|
    ip.name=party))
```

Other properties shown in Figure 7.29 constraint the operations behavior of the interface B that allow one to access the internal state of *Party Plan*.

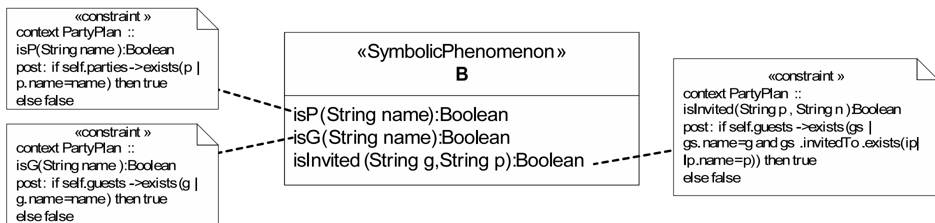

Figure 7.29: Representation of the shared phenomena B

Requirements specification The requirements specify the effects of the commands issued by the operator on the internal state of *Party Plan*. The requirements are defined by means of the `stm` diagram shown in Figure 7.30.

Notice that requirements predicate on three different aspects: 1) they establish which commands issued by *John and Lucy* are not correct and therefore must be rejected, 2) they define the commands context, and 3) they define the effect of the commands on the internal state of *Party Plan*.

The admissible commands are those specified by the transitions of the `stm` reported in Figure 7.30. All other commands are syntactically incorrect and are rejected.

The context of the commands is defined by using two variables named h and q that store the current guest and the party context (the name of the guest and of the party), respectively.

The commands effect is defined by using the operations defined in the interface B, as they allow one to access the state of *Party Plan*.

Machine specification The machine specification is described in the `stm` diagram of Figure 7.31. Notice that the diagram is similar to the one used for the requirements specification, except for the operations invoked to update the state of *Party Plan*.

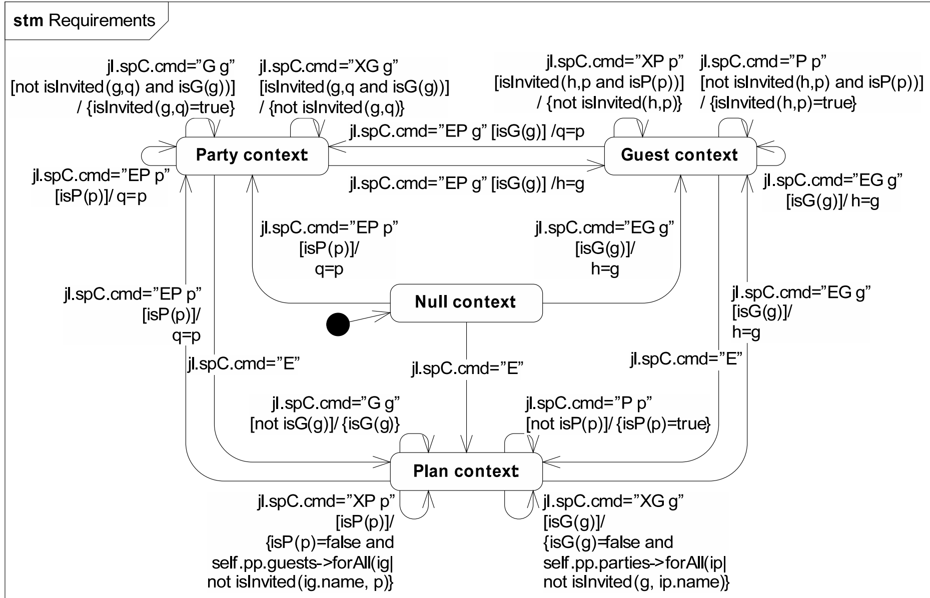

Figure 7.30: The state machine representing the user requirements

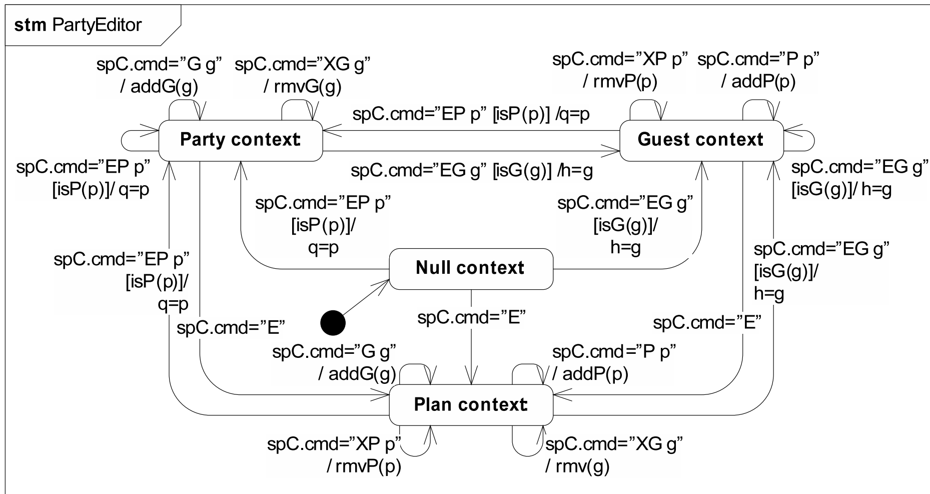

Figure 7.31: The state machine representing the Party editor machine

7.5 Transformation: Mailfiles analysis

This section presents the Mailfiles analysis problem, a simple example proposed by Jackson in [37] to illustrate the Transformation frame.

> *Fred has decided to write a program to analyse some patterns in his email. He is interested in the average number of messages he receives and sends in a week, the average and maximum message length, and similar things. After some thought he has worked out that he wants a report that contains a line for each of his correspondents. The line shows the correspondent's name, how many days the report covers, the number of messages received from the correspondent and their maximum and average lengths, and the same information for the messages sent to the correspondent by Fred.*

The Problem diagram for the proposed problem is shown in Figure 7.32.

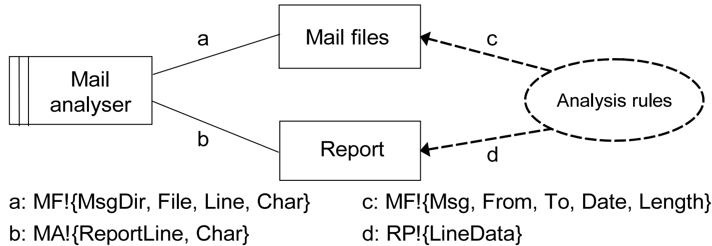

Figure 7.32: The mail file analysis transformation frame

Problem context definition The problem context is composed of the *Machine* domain *Mail analyser*, and of two lexical domains respectively named *Mail files* and *Report*. Notice that *Mail analyser* represents an instance of *Transformmachine* of the Transformation frame, while *Mailfile* and *Report* represent instances of the domain *Inputs* and *Outputs*.

The problem context definition exploits the `bdd` proposed in Figure 7.33.

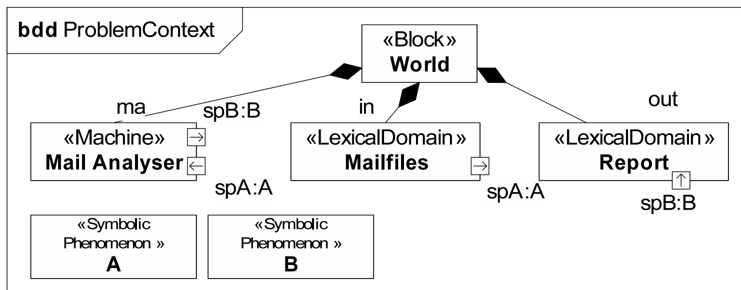

Figure 7.33: The Mail file analyser: context diagram

Mail files is provided to *Mail analyser* through *spA*, and *Report* is generated by the machine and sent to *Mail files* through the port *spB*.

Problem domain modeling The internal structure of the lexical domains *Mail files* and *Report* is reported in the bdds of Figure 7.34 and Figure 7.35.

Figure 7.34: The Mail files domain

Figure 7.35: The Report domain

Requirements specification Differently from the Odometer display case, where the use of the external language was constrained by the temporal aspects of the problem, here the requirements can be expressed by means of simply static properties. Since the user requirements involve only lexical domains, which are described

by means of bdds, it is convenient to specify such requirements using a constraint language like OCL [52]. Let us consider the requirement fragment that states that for every mail file there is one (and just one) line on the report, and that at the beginning of the line the name of the file (corresponding to the name of the sender) is reported. Such requirement is defined by means of the constraint blocks reported in Figure 7.36.

```
«constraint»
Report properties
---
constraints
{rep.dataLine->foreach(dl | mf.message->exists (msg |
    msg.Header.From.name = dl.Name)) and
mf.message->foreach(msg | rep.dataLine->one(dl |
    msg.Header.From.name = dl.Name))}
---
parameters
rep: Report, mf: Mailfiles
```

Figure 7.36: The rules of the analysis specified as a constraint

Machine specification The specification of the machine *Mail Analyser* is also defined by means of an OCL statement (see Figure 7.37).

```
«constraint»
Report properties
---
constraints
{rep.reportLine->foreach(rl | MF.Directory.file->exists(f |
  f.name = rl.substring(1, f.name.size))) and
MF.Directory.file->foreach(f | rep.reportLine->one(rl |
  f.name = rl.substring(1, f.name.size)))}
---
parameters
rep: B, MF: A
```

Figure 7.37: The specification of the analyser

Note that the specification of the machine is very similar to the definition of requirements. This is not surprising, since both specifications concern the relation between inputs and outputs. The requirements are expressed in terms of phenomena that are relevant to the user (message, sender, recipient, etc.) while the specification of the machine involves physical elements (files, lines, etc.).

7.6 Final remarks

The basic problem frames proposed by Jackson play a fundamental role in the decomposition approach of Problem Frames. Although such frames are relatively simple, Jackson presents basic problem frames as the final objective of the decomposition approach. Moreover, Jackson associates each basic frame with a correctness argument template that is used to determine whether a proposed solution specification stands in the correct relationship to the world description and

the requirements. Such template, also referred to as frame concern, represents the specification pattern that the analyst has to apply when he/she identifies that a certain problem matches the characteristics of one of the proposed basic frames.

The basic problem frames proposed by Jackson are only a starting point. Currently, as presented in Chapter 5, problem frames research is oriented to the identification and validation of further frames.

The frames of Jackson are illustrated by means of simple examples of different application domains. The examples proposed by Jackson cover the usage of the main concepts of the Problem Frames approach. All types of domains and phenomena are used in the different problems.

This chapter reported the definition of the catalogue proposed by Jackson using SysML. This experience represents a first step towards the validation of the combined approach based on SysML and Problem Frames.

SysML was able to represent both structural and behavioral aspects of problem domains, requirements and machine specifications. Further considerations on the expressiveness of SysML and on the effectiveness of the combined approach will be provided in Chapter 9.

Chapter 8

Case study

The main goal of this chapter is to show the application of the previously illustrated requirements analysis and specification methodology in order to further exemplify the joined application of SysML and Problem Frames. Moreover, the chapter aims at validating the proposed approach with a case study of industrial complexity.

This chapter presents the specification of the requirements of a traffic intersection light system. The intersection controller manages the traffic lights for cars and pedestrian traffic at a four way intersection in a European town.

The controller reacts to events such as the pressing of a button at a crosswalk or vehicles transit and operates four vehicle and pedestrian traffic lights that are positioned next to the intersection.

The overall system is a real time safety critical application: failure to operate in a proper way can result in serious and even fatal accidents.

The requirements are presented by using the combined usage of Problem Frames and SysML according to the approach introduced in the previous chapters.

The problem was originally presented by Pacelli et al. in [58] and revised by Laplante in [42]. Here the requirements of the problem are furthermore adapted. The goal of Pacelli et al. in [58] was the development of an actuated traffic control process, the aim of Laplante [42] was the design of the software of the intersection controller, while the goals in this chapter concern the analysis of the requirements and the specification of the intersection controller.

8.1 System description

Intelligent traffic management systems help to make better use of existing transport routes in towns and cities where the volume of traffic is constantly rising. The main requirement for a traffic controller is to maintain the highest reasonable level of efficiency under different traffic conditions depending on the characteristics of the intersection.

The goal of the chapter is the specification of the requirements of the traffic controller that operates the intersection reported in Figure 8.1.

The intersection is composed of two approaches named NS and EW, respectively. Each of the approach is partitioned into two distinct semi-approaches: the approach NS is composed of the semi-approaches N and S, while the approach EW of the semi-approaches E and W.

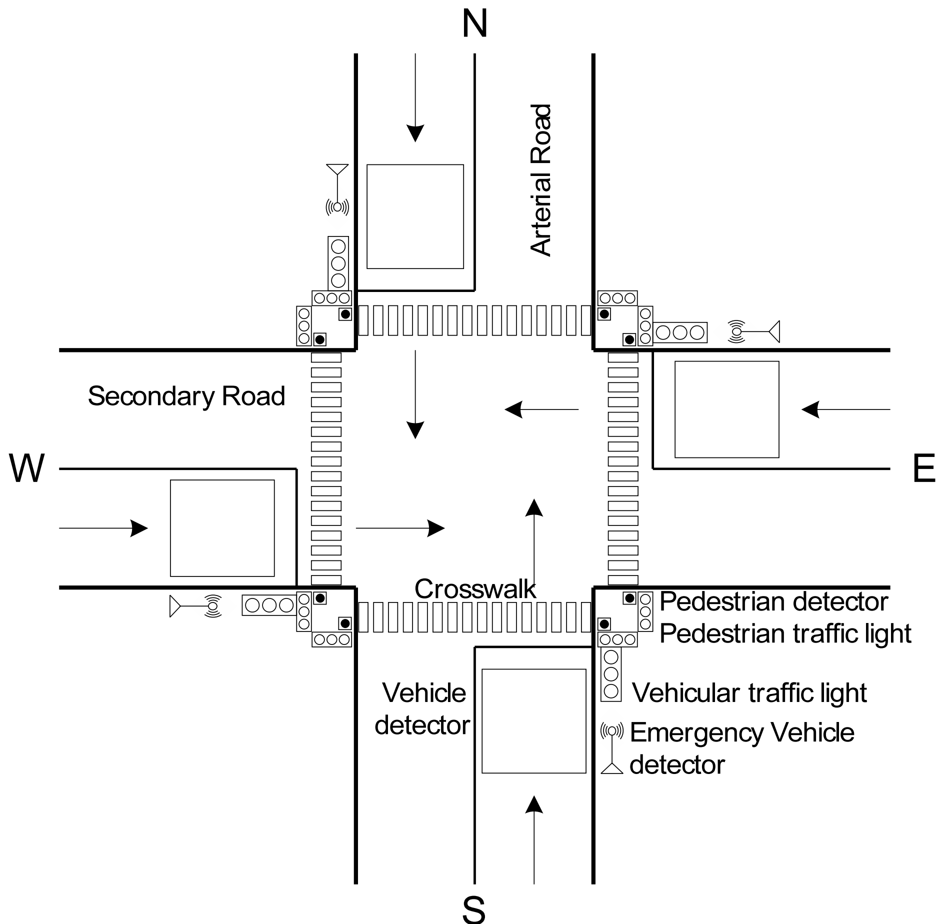

Figure 8.1: Intersection topography

Each of the approaches can be covered in two different directions and is equipped with a vehicle traffic light and a vehicle presence detector positioned immediately before the intersection. For the sake of simplicity consider that the traffic lights do not provide directional signals, hence the involved vehicles are not allowed to curve, changing from the NS to the EW direction or viceversa.

The approaches are crossed by pedestrian crosswalks equipped with two pedestrian presence detectors (to be activated manually by pressing buttons) and two pedestrian traffic lights.

The system is also characterized by detector (one for approach) that reveal the requests to cross the intersection of emergency vehicles like ambulances, firemen

trucks or police cars. For safety reasons such emergency vehicles need to find the green light when they reach the intersection in order to spend as little time as possible to cross the intersection.

The system is also equipped with a console that allows an operator to manually set the state of the traffic lights and to configure the behavior of the controller. The controller is the core component of the system. It has to coordinate the state of the different traffic standards according to traffic conditions and to the requests of the pedestrians and of emergency vehicles.

The controller receives inputs from the vehicle presence detectors, the pedestrian presence detectors, the manual console and the emergency vehicle detectors, and once analysed the actual state of the system it has to send commands to change the state of the vehicle and pedestrian traffic standards.

The intersection controller supports the following operating modes: fixed cycle, semi actuated, fully actuated, locally controlled, and emergency preempted mode.

In the fixed cycle mode, all the operations are pretimed and preset in the controller. The controller continuously issues a predefined sequence of commands regardless of traffic conditions and pedestrian requests. Such commands cause the modification of the states of the traffic lights disposed along the approaches. The duration of the states is predefined and preset in the controller. The combination of the expected states of the traffic lights after a group of commands issued by the controller is named phase and represents a state of the intersection. The system evolves by iterating a sequence of predefined phases.

The fixed cycle pedestrian actuated mode is a little variant to the previous operating mode that considers the requests of pedestrians. In this case the sequence of the phases is the same of the fixed cycle mode, but the duration of the states in which pedestrians have to wait is reduced according to the pedestrian requests. More specifically, when a pedestrian has to cross a certain approach and the corresponding traffic light forbids the passage, he/she presses the request button in order to anticipate the go signal.

In the semi actuated mode it is assumed that the approaches at the intersection are characterized by different priorities. As described in Figure 8.1, the semi-approaches N and S are parts of the same arterial road, while E and W are parts of a minor traffic route. Accordingly, the minor approach EW receives a green light only when traffic is present. Motor vehicle detectors determine when vehicles are on the minor street and the controller provides a variable amount of green time depending on the status of the road. When no traffic is on the minor street or the maximum allowable service time has been reached, the green indication is returned to the main street.

In the fully actuated mode, all phases are controlled by means of detection mechanisms. Vehicle presence detectors are distributed along all the approaches. The duration of the phases is determined according to the actual traffic condition. This mode is more efficient than the previous ones when traffic conditions vary frequently over time. Notice that the duration of lights states may be affected by pedestrian requests both in fully and semi actuated mode.

In the locally controlled mode, the evolution of the intersection is determined by an operator that sends commands by means of a dedicated console. The operator determines the duration of the green and red lights. The requests of the

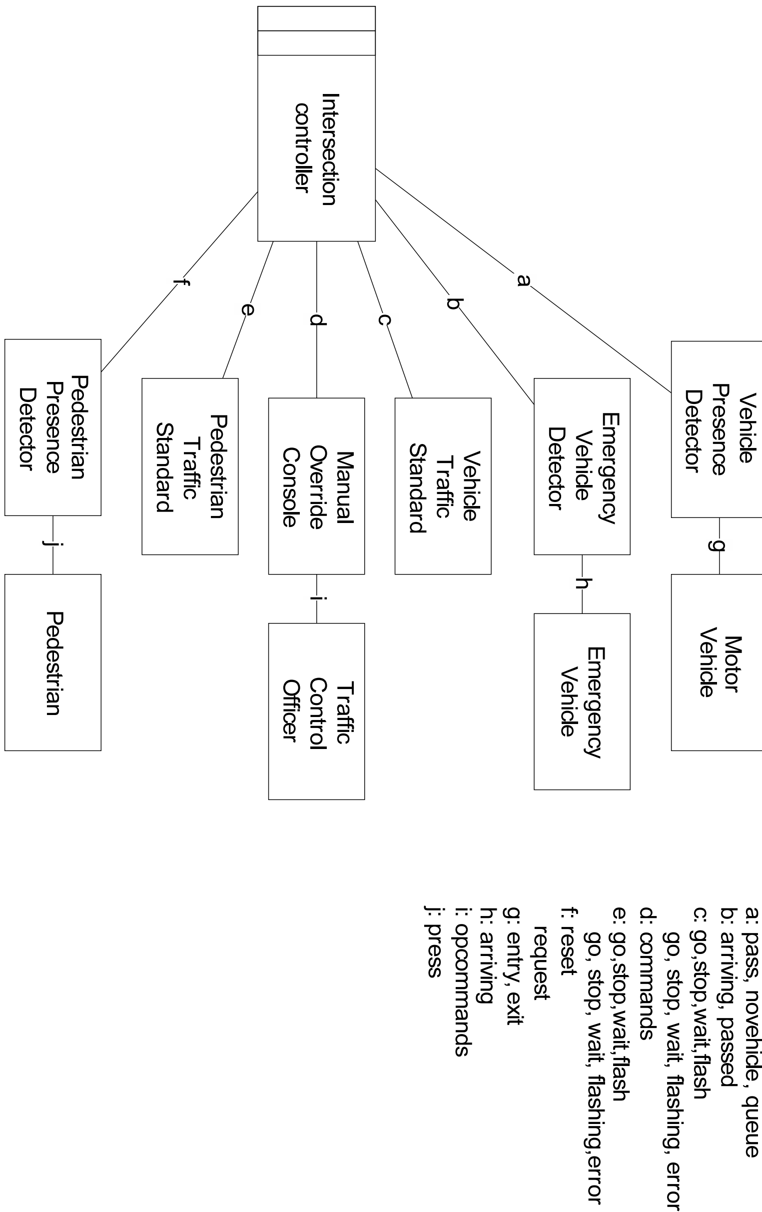

Figure 8.2: The Context diagram for the intersection controller problem

pedestrians, as well as the signals sent by the motor vehicle detectors, are not considered since it is supposed that the operator can see the waiting cars. This operating mode is activated whenever an operator sends a command by using the dedicated console and lasts until an explicit command requires to change the operating mode.

In the preempted mode the normal control cycle (fixed cycle, semi or fully actuated) is suspended and replaced by a specific sequence: the traffic lights of all the semi approaches of the intersection are switched to red with the exception of the ones of the approach where the vehicle that triggered the preemption sequence is arriving. The normal traffic light cycle resumes after the transponder reveals that the emergency vehicle crossed the intersection.

8.2 The context of the problem

The Context diagram of the intersection controller system is shown in Figure 8.2. The diagram defines the domains of the problem and the shared phenomena among such domains.

According to the combined PFs-SysML approach, the context of the problem is also described by means of the Block Definition Diagram (`bdd`) of Figure 8.3, which introduces the domains and the phenomena, and the Internal Block Diagram (`ibd`) of Figure 8.4 that shows the architecture of the problem context.

The problem context is characterized by a machine domain *Intersection controller*, by multiple causal problem domains named *Vehicle Presence Detector*, *Emergency Vehicle Detector*, *Vehicle Traffic Standard*, *Manual Override Console*, *Pedestrian Traffic Standard*, *Pedestrian Presence Detector* and by the biddable domains named *Motor Vehicle*, *Pedestrian*, *Traffic Control Officer* and *Emergency Vehicle*.

The causal domain *Vehicle Presence Detector* represents a occupancy loop disposed along the border of a rectangular area of a tenth of squared meters and positioned on each semi-approach immediately before the intersection. Such devices are used for indicating the presence and the passage of vehicles.

The vehicle presence detector recognizes when a motor vehicle enters or exits its area and generates signals that specify 1) whether a motor vehicle passed the loop, 2) whether a vehicle has been staying motionless on the device for some period or 3) whether the device has not received signals for some period.

The domain *Vehicle Presence Detector* is represented in the `bdd` of Figure 8.3 by means of the homonymous stereotyped SysML Block *«CausalDomain»*.

The input signals *Entry* and *Exit* are shared phenomena controlled by *Motor Vehicle* and observed by *Vehicular Presence Detector*. They are represented in SysML by means of two boolean fields named *entry* and *exit* of the *«Controllable-Phenomenon»* Signal *VSignal*.

Motor Vehicle is represented by the stereotyped Block *«Biddable Domain»*. The sharing of the phenomena *Entry* and *Exit* is defined by means of two Flow Ports of type *VSignal*. The output Flow Port *vso* of the Block *Motor Vehicle* is used to represent that the domain controls the phenomena, while the input port *vsi* of the Block *Vehicle Presence Detector* is used to represent that the domain observes the phenomena.

The output signals *NoVehicle*, *Queue*, and *Passed* are shared phenomena between the domain *Vehicular Presence Detector* and the machine *Intersection Controller*. They are generated by the former domain and sent to the latter in order to notify the corresponding state of the detector. Such phenomena are represented by means of boolean attributes of Signal *VehicleRequest*. The sharing of the phenomena is represented by means a couple of FlowPorts of type *VehicleRequest* of Blocks *Vehicle Presence Detector* and *Vehicle*.

The domain *Pedestrian Presence Detector* represents the button positioned next to the pedestrian traffic light at each end of a crosswalk. This device gives pedestrians who wish to cross the street the ability to alert the intersection controller system of their presence. The notification is represented by means of a shared phenomenon, named *Press*, that is controlled by *Pedestrian* and observed by *Pedestrian Presence Detector*. The former domain is represented in SysML by the *«BiddableDomain»* Block *Pedestrian*, while the latter by the *«CausalDomain»* Block *Pedestrian Presence Detector*. Such phenomenon is represented by means of the signal *PSignal* and by a couple of Flow Ports of the Blocks *Pedestrian Presence Detector* and *Pedestrian*.

The notification to the controller is represented by means of the shared phenomenon *Pressed* controlled by *Pedestrian Presence Detector* and observed by Intersection Controller. Such phenomenon is represented in the `bdd` by the signal *PedestrianRequest* and by a couple of Flow Ports of the Blocks *Pedestrian presence Detector* and *Intersection Controller*. At the end of each phase the controller sends a signal to the detector to reset its state. This is done in order to prevent the detector to keep an already processed request across different phases.

The causal domain *Emergency Vehicle Transponder* is a particular device capable to recognize when emergency vehicles are going to reach the intersection. Such devices generally operate by using infrared signals, optical signals like visible strobe lights and radio signals.

Each emergency vehicle is equipped with an emitter, a device which emits radio pulses or visible flashes of light or infrared pulses at a specified frequency. Receiver devices positioned next to intersection along each of the semi approaches recognize the signal and preempt the normal cycle of traffic lights. Once the emergency vehicle crossed the intersection and the receiving device no longer senses the remote triggering device, normal operations resume.

Notice that we are not interesting in describing the usage of a particular communication technology, we simply focus on the communication mechanisms used by emergency vehicles and detectors. Hence, infrared, optical or radio signals can be represented by a phenomenon named *arriving*, which notifies that a certain emergency vehicle is moving towards the intersection. Such phenomenon is controlled by the domain *Emergency Vehicle* and observed by *Emergency Vehicle Transponder*. The phenomenon is represented by means of the *«Controllable Phenomenon»* Signal *EVSignal*, while the involved domains by means of the *«CausalDomain»* Block *Emergency Vehicle Detector* and the *«BiddableDomain»* Block *Emergency Vehicle*. Similarly to the previous illustrated cases, phenomena sharing is modeled by means of a couple of Flow Ports.

bdd ProblemContext

OCmd
vdE,vdW, vdN,vdS:VehicleRequest
pdE,pdW, pdN,pdS:PedestrianRequest

tsc:TrafficStdCmd
«CausalDomain» «block» **Pedestrian Traffic Standard**
tss: TrafficStdState
8 pts

pr:PedestrianRequest
«CausalDomain» «block» **Pedestrian Presence Detector**
ps:PSignal
8 ppd
dc:DetectorCmd

edEW, edNS: EVehicleRequest
«Machine» «block» **Intersection Controller**
pdcE,pdcW, pdcN,pdcS: DetectorCmd
ptssEW,ptssNS, vtssEW,vtssNS: TrafficStdState
1 ic
ptscEW,ptscNS, vtscEW,vtscNS: TrafficStdCmd

tsc:TrafficStdCmd
«CausalDomain» «block» **Vehicle Traffic Standard**
8 vts
tss: TrafficStdState

«CausalDomain» «block» **Vehicle Presence Detector**
vr: VehicleRequest
4 vpd

OCmd
1 mo
«CausalDomain» «block» **Manual Override**

«block» **World**

2 evd
«CausalDomain» «block» **Emergency VehicleDetector**
evr: EVehicleRequest
evs: EVSignal

* ev
«BiddableDomain» «actor» **Emergency Vehicle**
evso:EVSignal

* mv
«BiddableDomain» «actor» **Motor Vehicle**
vso:VSignal

* pd
«BiddableDomain» «actor» **Pedestrian**
pso:PSignal

* tco
«BiddableDomain» «actor» **Traffic ControlOfficer**

«Controllable Phenomenon» «signal» **PedestrianRequest**
detectorId:String

«Controllable Phenomenon» «signal» **VehicleRequest**
detectorId:String
noVehicle:Boolean
passed:Boolean
queue:Boolean

«Controllable Phenomenon» «signal» **EVehicleRequest**
Id:String
arriving:Boolean
passed:Boolean

«Controllable Phenomenon» «signal» **TrafficStdCmd**
value: TrafficStdCmdType

«enumeration» **TrafficStdCmdType**
Go
Stop
Wait
Flashing

«enumeration» **TrafficStdErrorType**
NoError
RoffYoffGoffFoff
RoffYoffGoffFon
RoffYoffGonFoff
RoffYonGoffFoff
RonYoffGoffFoff
RoffYoffGonFon
RoffYonGonFoff
RonYonGoffFoff
RonYoffGonFoff
RonYoffGoffFon
RoffYonGoffFon
RonYonGonFoff
RonYonGoffFon
RonYoffGonFon
RoffYonGonFon
RonYonGonFon

«Controllable Phenomenon» «interface» **OCmd**
setDuration(String appId, LightDuration type, Integer value)
changeMode(ModeType modeId)
nextPhase()
off()
setStartWorkingTime(Time t)
setEndWorking(Time t)

«enumeration» **LightDuration**
GoD
StopD
WaitD
FlashingD

«Controllable Phenomenon» «dataType» **TrafficStdState**
value: TrafficStdStateType
errorCode: TrafficStdErrorType

«enumeration» **TrafficStdStateType**
Go
Stop
Wait
Flashing
Error

«enumeration» **ModeType**
Fc
Fcp
Sa
Fa
M
Ep

«Controllable Phenomenon» «signal» **DetectorCmd**
id:String
detectorId:String
reset:Boolean

«Controllable Phenomenon» «signal» **PSignal**

«Controllable Phenomenon» «signal» **VSignal**
entry:Boolean
exit:Boolean

«Controllable Phenomenon» «signal» **EVSignal**

Figure 8.3: The bdd that defines domains and phenomena of the intersection controller problem

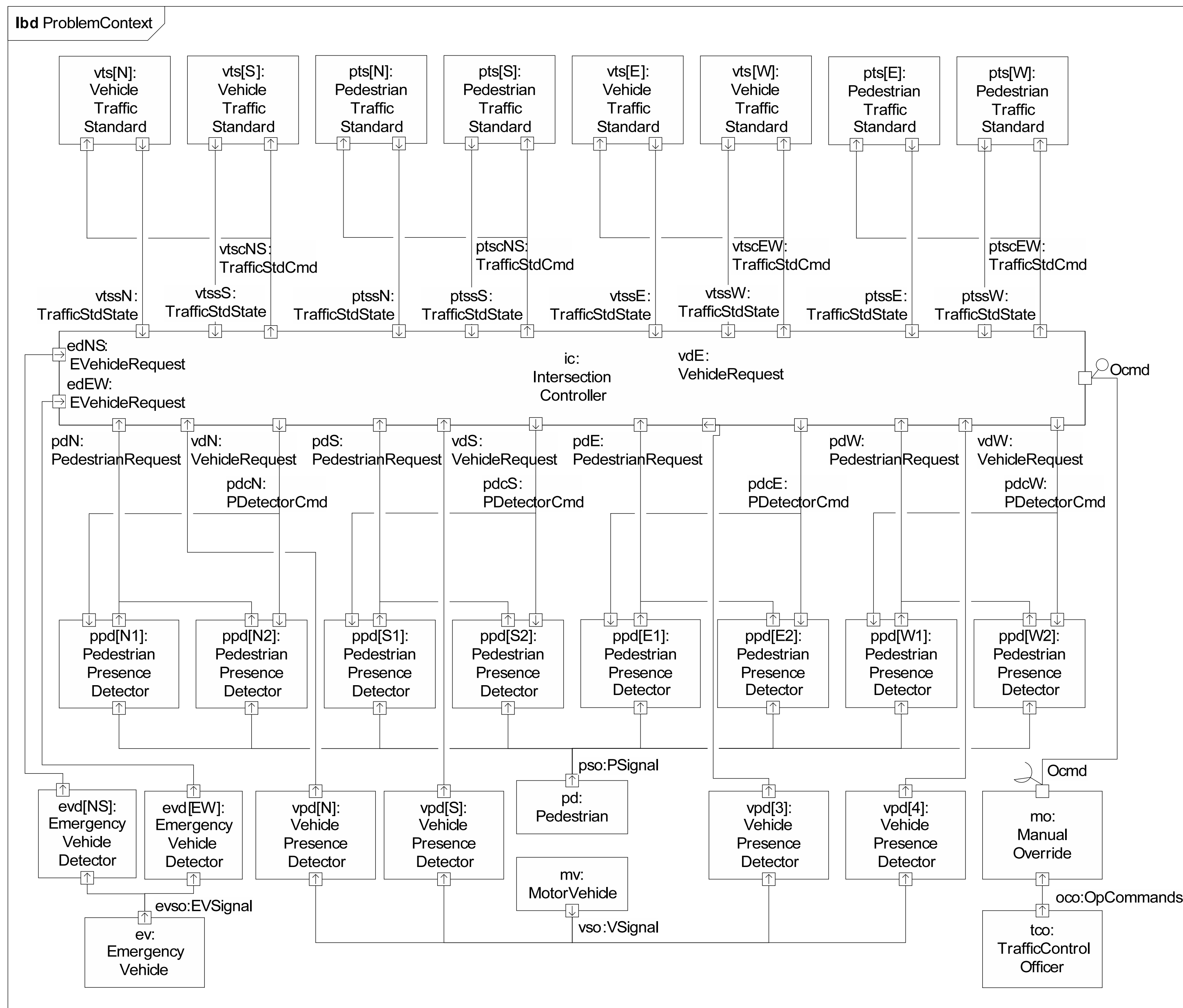

Figure 8.4: The ibd that describes the architecture of the context of the intersection controller problem

Emergency Vehicle Transponder analyses the incoming signals and notifies the intersection controller whether an emergency vehicle is reaching or has crossed the intersection. The notification is expressed by means of the shared phenomena *arriving* and *passed*, both controlled by *Emergency Vehicle Transponder* and observed by *Intersection Controller*. Such phenomena are represented by means of the *«ControllablePhenomenon»* Signal *EVehicleRequest* and by two conjugated ports between *IntersectionController* and *Emergency Vehicle Detector*

The causal domain *Vehicle Traffic Standard* represents the vehicular traffic lights positioned at the end of each semi approach next to the intersection, while *Pedestrian traffic standards* represents the traffic lights that regulate the access to each crosswalk. Such domains are represented by the *«CausalDomain»* Blocks *Vehicle Traffic Standard* and *Pedestrian Traffic Standard*, respectively.

The traffic lights (of both the types) are equipped with three fixed red, yellow and green lamps and one flashing yellow lamp. They change their state by turning on and off the lamps according to the commands issued by the intersection controller. Such commands are represented by means of the shared phenomena *Go*, *Stop*, *Wait*, *Flash*, controlled by *Intersection Controller* and observed by *Vehicle Traffic Standard* and *Pedestrian Traffic Standard*. The phenomena are represented in the `bdd` of Figure 8.3 by means of the *«ControllablePhenomenon»* Signal *TrafficStdCmd*. Such Block is characterized by the attribute *value* of of type *TrafficStdCmdType*, which in turn is an enumeration of values *Go*, *Stop*, *Wait* and *Flash*.

The intersection controller is also able to check the current state of the traffic lights, i.e., for each traffic light it can check which lamps are on and which are off. This allows the controller to understand whether there are problems like burn out lamps or incorrect reaction to previously issued commands. The state of the traffic lights is represented by means of the shared phenomena *Go*, *Stop*, *Wait*, *Flashing* and *Error*. They are represented in SysML by the *«ControllablePhenomenon»* Signal *TrafficStdState*. Such Block is characterized by the attribute *value* and *errorCode*. The former represents the current state of the traffic light, and is of type *TrafficStdStateType*, that is an enumeration of the values *Go*, *Stop*, *Wait*, *Flash* and *Error*. While the latter of type *TrafficStandardErrorType* is used to represent an error condition. More specifically, *TrafficStandardErrorType* is an enumeration of all possible combinations of the states of the lamps. By using such information the controller is able to find the problem and it can react in a proper way.

The sharing of the phenomena is expressed by means of a couple of conjugated ports of type *TrafficStdCmd*. Notice that *Intersection Controller* defines a port for each of the pedestrian or vehicular traffic light that it has to manage.

Manual Override Console represents a device positioned next to the intersection, which allows a traffic control operator, represented by the domain *TrafficControlOfficer* to monitor the state of the intersection and to manipulate its state. The device is equipped with several buttons and a numerical keyboard. Each button is used to send a specific command to the intersection controller. The console domain is represented by the *«CausalDomain»* Block *Manual Override Console*, while the operator by the *«BiddableDomain»* Block *Traffic Control Officer*.

The commands sent by the operator via the console are represented by means

of the shared phenomena *opcommands* controlled by *Traffic Control Officer* and observed by *Manual Override Console*. The console converts the operator commands into specific commands for the intersection controller. Such commands are shared phenomena *commands* controlled by *Manual Override Console* and observed by *Intersection Controller*. The commands are represented by means of the *«ControllablePhenomenon»* Interface *OCmd*, which defines the operations that an operator may invoke. A detailed enumeration of such operation will be provided in the section that illustrates the intersection controller manual mode. In this case the sharing of the phenomena is represented by means of two Standard Ports built on the same interface *OCmd*. The Blocks *Intersection Controller* implements the interface while *Manual Override* requires it.

8.3 Looking inside the domains

In this section the causal domains introduced in the previous section are further described by considering both structural and behavioral characteristics.

Pedestrian Presence Detector The Block *PedestrianPresenceDetector* is characterized by two input Flow Ports named *ps* and *dc* of type *PSignal* and *DetectorCmd*, respectively. The Block is also characterized by an output port *pr* of type *PedestrianRequest*.

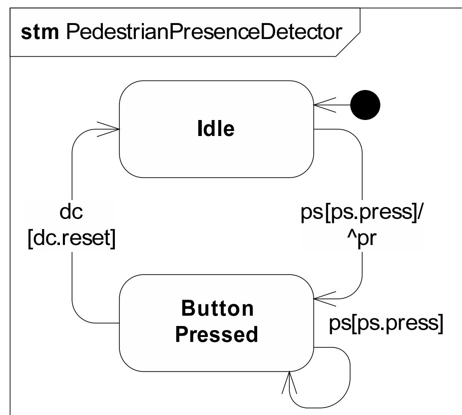

Figure 8.5: The `stm` that describes the behavior of the domain *PedestrianPresenceDetector*

The behavior of the domain is described by the `stm` diagram of Figure 8.5. The state machine is characterized by two states. Initially, the domain is in the state *Idle*, then, as soon as a pedestrian presses the button, a *PSignal* is generated and received by the domain via the port *ps*. This signal triggers the transition

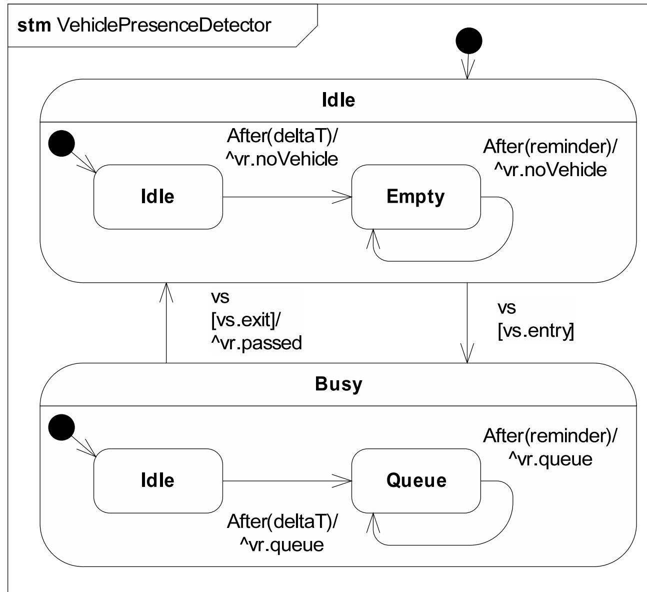

Figure 8.6: The `stm` that describes the behavior of the domain *VehiclePresenceDetector*

to the state *Button Pressed*. Notice that the detector does not report all the pedestrian requests. In case a pedestrian pushes the buttons several times only the first request is taken into account. Hence, in case the state machine receives a signal *PSignal* when it is in the state *Button Pressed*, the signal is ignored.

The state machine returns to the state *Idle* whenever the command *reset* is issued via the port *dc*.

Vehicle Presence Detector The Block *VehiclePresenceDetector* is characterized by an input Flow Port vs of type *VSignal* and an output Flow Port *vr* of type *VehicleRequest*. It is also characterized by two internal integer attributes *deltaT* and *reminder* that represent time intervals whose aim will be clarified later.

The behavior of the domain is described by means of the `stm` diagram shown in Figure 8.6. The state machine is characterized by two composite states named *Idle* and *Busy*. *Idle* in turn is characterized by an internal state machine composed of the states *Idle* and *Empty*. Also the composite state *Busy* is characterized by an internal state machine, which is composed of the states *Idle* and *Queue*.

At starting time the overall machine enters the state *Idle*. As soon as a signal *entry* is received through the port *vs* the machine passes to the state *Busy*. When the machine is in the state *Busy* and a signal *exit* is received via the port *vs*, it returns to the initial state *Idle* and generates a passed signal that is issued via the port *vr*.

Whenever the composite state *Idle* is entered, the internal state machine enters the state *Idle*. Then, in case no signal arrives within *deltaT* time the internal `stm` passes to the state *Empty* by generating the signal *noVehicle* that is externally

propagated through the port *vr*. Then, in case no *VSignal* arrives within *reminder* time the `stm` re-enters the state *Empty* by generating another signal *noVehicle*.

Similarly, whenever the composite state *Busy* is entered, its internal state machine enters the state *Idle*. Then, by following the same criteria of the internal state machine of the composite state *Idle*, signals *queue* can be issued via the port *vr*.

Emergency Vehicle Transponder The Block *EmergencyVehicleDetector* is characterized by an input Flow Port *evs* of type *EVSignal* and by an output Flow Port *evr* of type *EVehicleRequest*. It also characterized by an Integer attribute *maxDelay* representing a time interval whose usage will be specified later.

Figure 8.7 reports a `stm` that describes the behavior of the transponder.

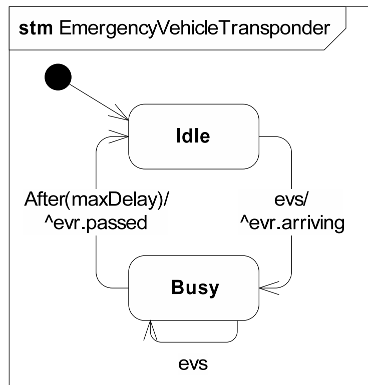

Figure 8.7: The `stm` that describes the behavior of the domain *EmergencyVehicleDetector*

The `stm` is characterized by two internal states named *Idle* and *Busy*. At starting time, the `stm` enters the state *Idle*. The machine remains in the state *Idle* until signal *arriving* is received on the port *EVSignal*. Such signal triggers the transition to the state *Busy* and causes the generation of signal *arriving* on the output port *evr*. Once entered such state, whenever signal *arriving* is received, a transition fires that re-enters the state *Busy*. In case no input signal has received for *maxDelay*, then the state machine returns to the state *Idle* by generating signal *passed* on the output port *evr*.

Traffic Standards The Block *Vehicle Traffic Standard* is characterized by an input port *tsc* of type *TrafficStdCmd* and an output port *tss* of type *TrafficStdState*. The domain is internally composed of several subdomains. More specifically, it is characterized by a controller a sensor, four lamps and four switches.

The internal composition of the domain is shown in the `bdd` of Figure 8.8 and in the `ibd` of Figure 8.9.

The Block *Switch* is characterized by two input Flow Port named *in* and *cmd* of type *PowerSupply* and *LampSwitchCmd* and an output port of type *PowerSupply*.

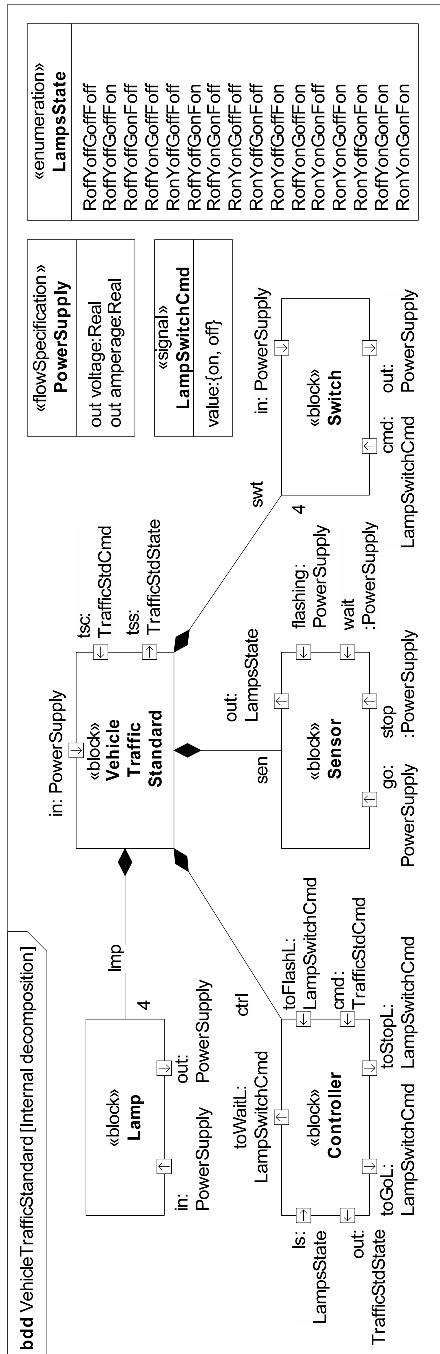

Figure 8.8: The `bdd` that describes the internal components of the Block *VehicleTrafficStandard*

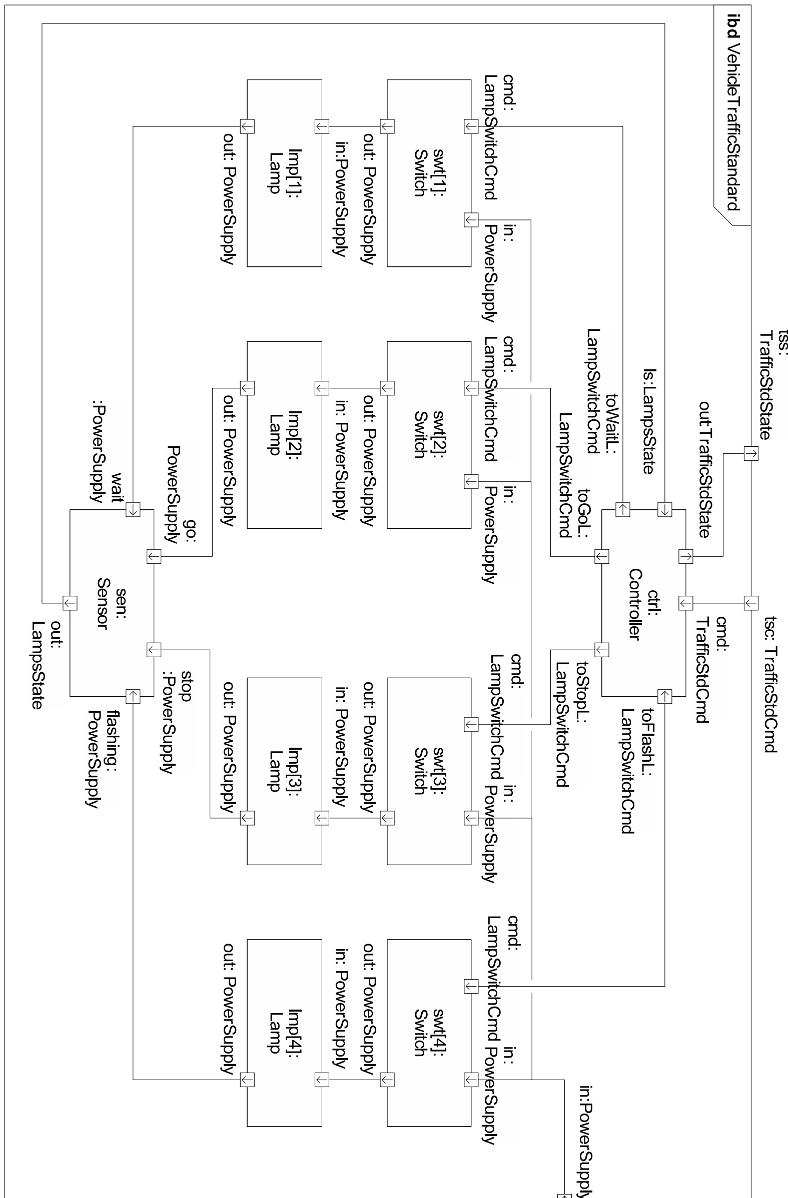

Figure 8.9: The ibd that describes the internal architecture of the Block *Vehicle-TrafficStandard*

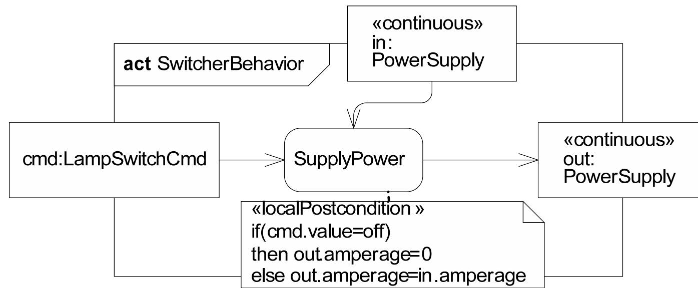

Figure 8.10: The act that describes behavior of the Block *Switch*

LampSwitchCmd is a Signal characterized by an attribute *value*, an enumerative on the values *on* and *off*.

The behavior of the switch is illustrated in the act diagram of Figure 8.10. The activity is characterized by two input and an output parameter nodes named *cmd*, *in* and *out*. Notice that such nodes are allocated to the homonymous ports of the Block *Switch*. Whenever a signal *cmd* of type *LampSwitchCmd* is received, the action *SupplyPower* is invoked. The action receives in input a continuous flow of data of type *PowerSupply*, and once analysed the value of the last command received, it generates a continuous flow of *PowerSupply*. More specifically, in case the value of the last command received is *off* the amperage of *out* is zero, otherwise the amperage is the same as the input.

The Block *Lamp* is characterized by the input port *in* and the output port *out* both of type *PowerSupply*. *PowerSupply* is a FlowSpecification that represents the power supply for the device. It is characterized by two attributes representing the amperage and voltage of the power supply.

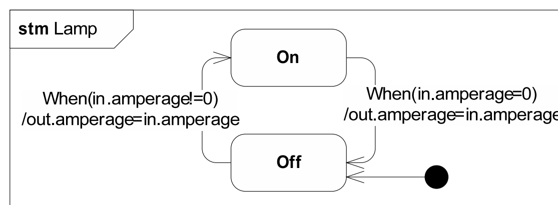

Figure 8.11: The stm that describes the behavior of the Block *Lamp*

The behavior of a lamp is specified by means of the stm diagram reported in Figure 8.11. More specifically, at starting time the lamp is *Off*, then in case the input amperage is different from zero, the stm enters the state *On*. As soon as the amperage equals zero the stm passes to the state *Off*. In case the lamp is in the state *On* a continuous *PowerSupply* with a non-zero amperage is provided via the output port *out*, else no amperage is provided.

The Block *Sensor* is characterized by four input ports of type *PowerSupply* named *go*, *stop*, *wait* and *flashing*, and by an output port *out* of type *LampsState*. *LampsState* is an enumerative type that represents all the possible combination

of states of the four lamps of the traffic light.

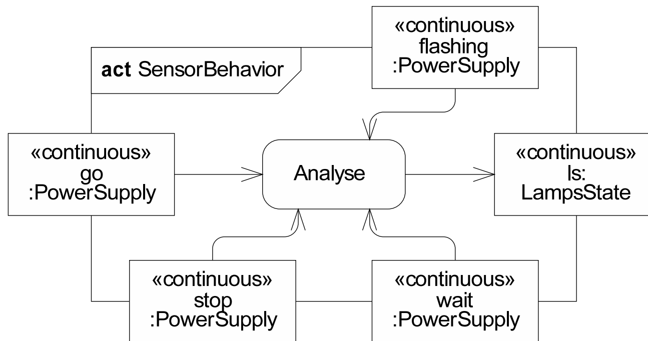

Figure 8.12: The `act` that describes the behavior of the Block *Sensor*

The behavior of the sensor is described by means of the activity diagram shown in Figure 8.12. The activity is characterized by four input parameter nodes named *go*, *stop*, *wait* and *flashing* of type *PowerSupply* and by an output node *ls* of type *LampsState*. Depending of the amperage value of the input parameters, the activity continuously generates a value corresponding to the actual state of all the lamps of the traffic light.

The Block *Controller* represents an internal controller of the traffic light that interprets the commands received on the input port *tsc*, and sends commands to the switches that turn on and off all the lamps and generate a flow of information that describes the current state of the traffic light. The Block is characterized by an input Flow Port *cmd* of type *TrafficStdComd*, an input port *ls* of type *LampsState* and four output ports of type *LampSwitchCmd* named *toGoL*, *toWaitL*, *toStopL* and *toFlashL*.

The behavior of the controller is illustrated by the `act` diagram of Figure 8.13 and by the `stm` diagram of Figure 8.14. The activity shows how the input commands received on the input port *cmd* are converted into commands to turn on and off every lamp of the traffic light.

The activity is characterized by an input parameter node *cmd* of type *TrafficStdCmd* and four output parameter nodes of type *LampSwitchCmd* named *toGoL*, *toWaitL*, *toStopL* and *toFlashL*. Such parameters are allocated to the homonymous ports of the Block *Controller*. Whenever a command is received on the port *cmd* the action *Analyse* is invoked. The computational effects of the action are described by the post condition shown in Figure8.14.

The `stm` diagram shows the internal evolution of the controller depending on the commands received by the input port.

The `stm` is characterized by the states *Green*, *Yellow*, *Red* and *Flashing* and *Error*. At starting time the `stm` enters the state *Flashing*, then depending on the value of the commands received via the port *cmd*, the `stm` may evolve in the other states. Notice that for each state transition, the controller generates four signals of type *LampSwitchCmd*. Three of them are characterized by the value *off* while the fourth one by the value *on*.

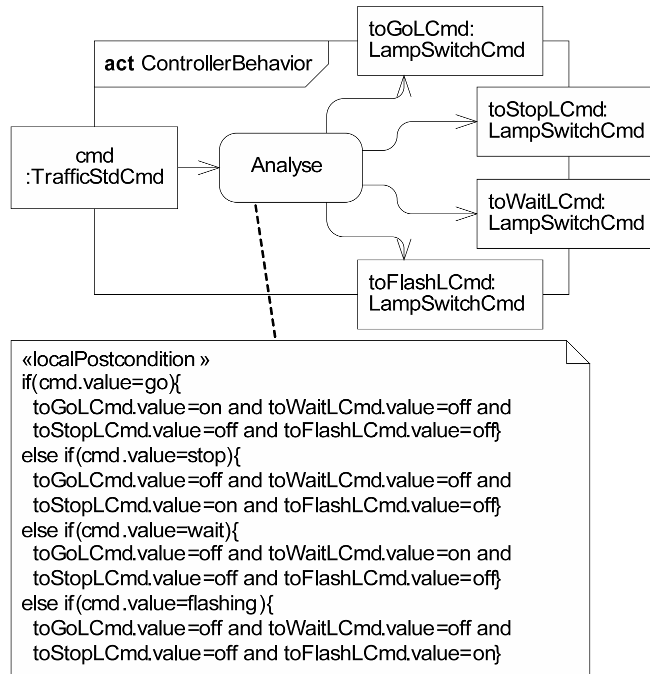

Figure 8.13: The `act` that describes the behavior of the Block *Controller*

The controller, after issuing the commands to the lamps, checks whether the current state of the lamps is the expected one. The control is done by invoking the activity *CheckState* that is illustrated in Figure 8.15.

The activity is characterized by two input object nodes named *ls* of type *LampsState* and *lcmd* of type *TrafficStdStateType*. Notice that such nodes are allocated to the homonymous ports *ls* and *out* and to the attribute *lscmd*, an internal attribute of the Block *Controller* that stores the expected state of the traffic light. When invoked, the activity executes the action *CheckState*. Such action receives in input a continuous flow of information that specifies the current state of the lamps of the traffic light. Then it compares the current states with the expected one specified by last command issued *lcmd* and generates a continuous flow of information by specifying the current state of the traffic light. More specifically, as specified by the post condition of *CheckState* shown in Figure 8.15, in case the states are compatible, the generated output state *out* is characterized by the field *value*, which is set to the last command issued and by the field *errorCode*, which is set to *NoError*. Otherwise *value* is set to *error*, *errorCode* to the actual state of the lamps *ls*, and a signal *Error* is generated. Such signal triggers the transition from the current state of the `stm` *Controller* to the state *Error*. Notice that an error may occur in case a lamp that should be on is burn out or in case the current combination of the lamps state is different from the expected one.

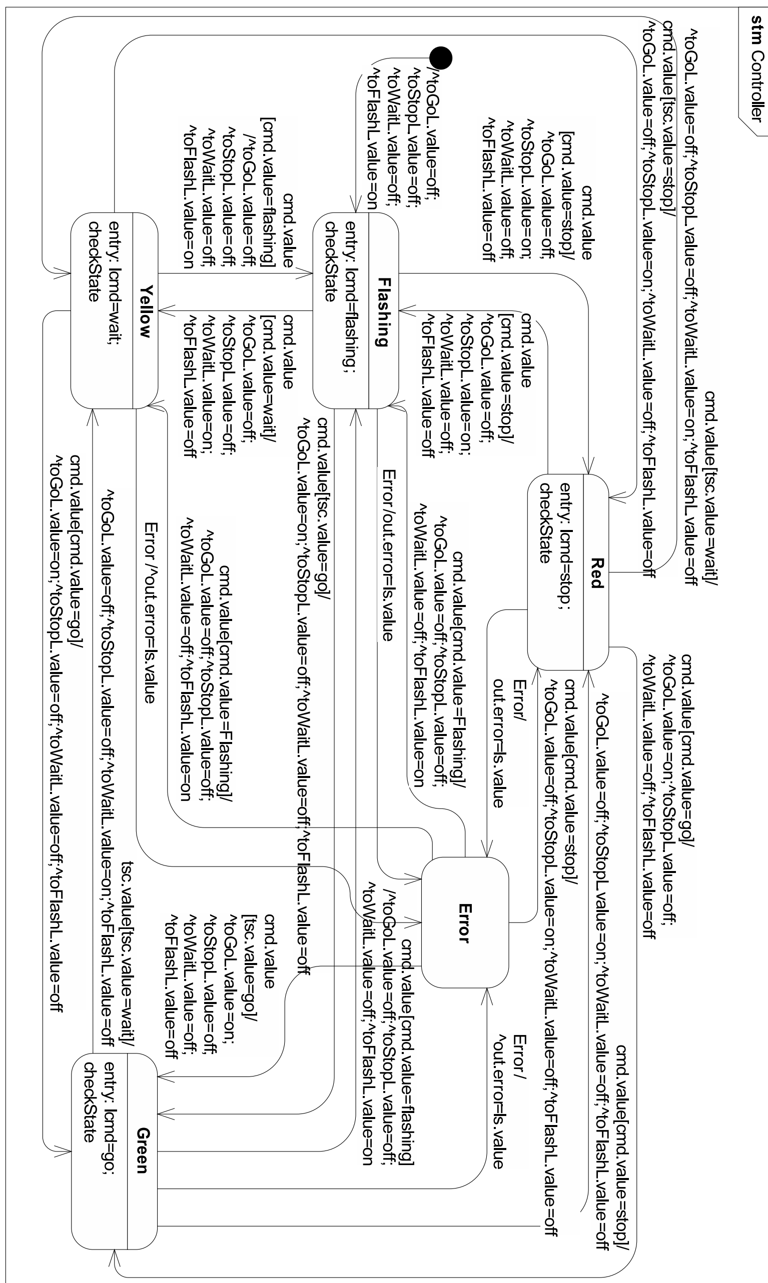

Figure 8.14: The `stm` that describes the behavior of the Block *Controller*

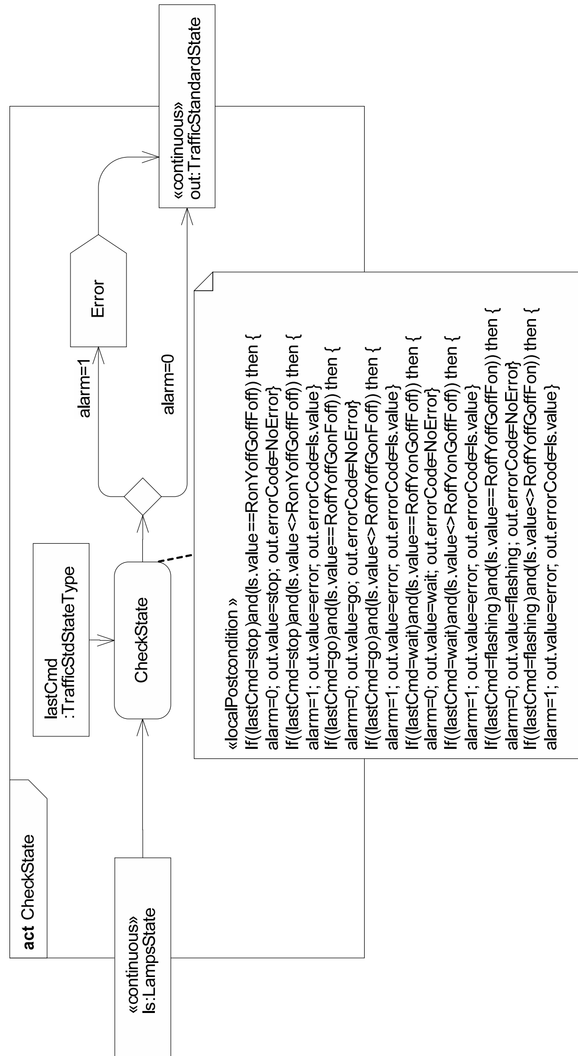

Figure 8.15: The `act` that describes the activity *CheckState*

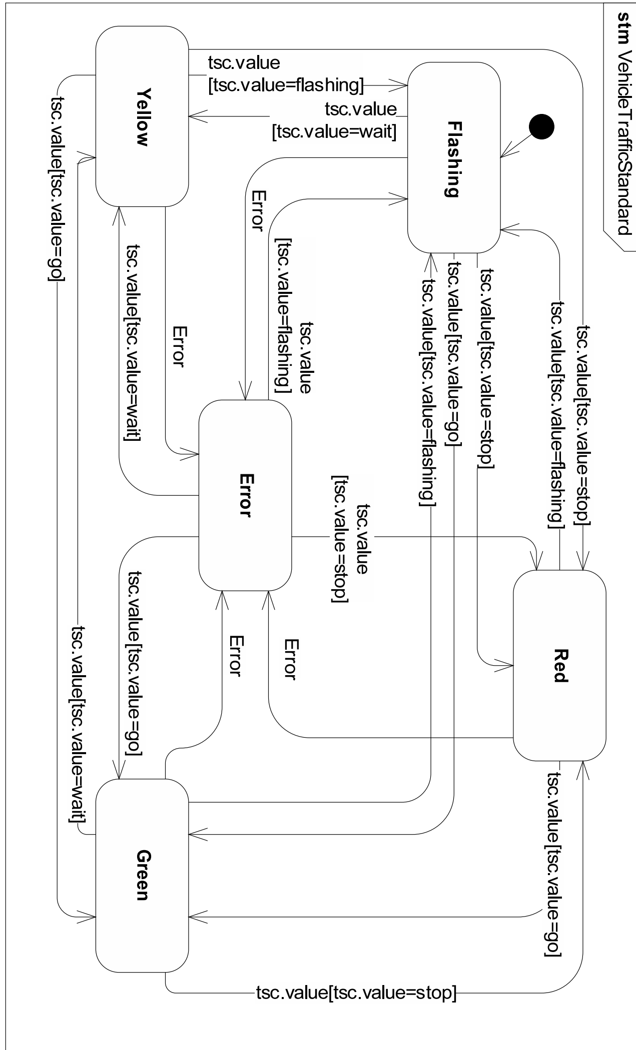

Figure 8.16: The `stm` that describes the behavior of the Block *VehicleTrafficStandard*

The overall behavior of the traffic light Block is described by the `stm` diagram shown in Figure 8.16. Although the `stm` is quite similar to the one that describes the controller it complements such description by specifying the evolution of the traffic light by means of the commands received by the intersection controller.

Notice that the previously introduced diagrams that describe both the structural characteristics and the behavior of the Block *VehicleTrafficStandard* can also be used also for the Block *PedestrianTrafficStandard*. The only difference consists in different allocation of the input and output parameters of the activities and of the signals generated by the internal components. Notice that such diagrams are not reported here since they are not particularly significant.

8.4 The requirements

The first activity towards the analysis of the requirements of the problem consists in sketching out the problem by specifying its Problem diagram. This allows one to explicitely define which domains control the shared phenomena. It also allows one to specify the relationships among the problem domains and the general requirements of the problem by defining which domains are constrained and which are simply referred to, and by specifying the internal phenomena interested by this kind of relationship.

A preliminary high level analysis of the informal description of the problem, allows one to identify the role of the involved problem domains as well as their interactions. The general requirements of the problem concern the state of the traffic lights of the approaches with respect to the operating mode and the traffic conditions. As a consequence, *Vehicle Traffic Standard* and *Pedestrian Traffic Standard* are problem domains constrained by the general requirements of the problem. The traffic conditions are determined by motor vehicles, emergency vehicles, and pedestrian, hence, the problem domains that represent such entities affect the state of the intersection system. As a consequence the problem domains *Motor Vehicle*, *Pedestrian*, and *Emergency Vehicle* are referred to by the general requirements of the problem. Similarly, the domain *Traffic Control Officer* may influence the current state of the intersection, hence also this domain is referred to by the requirements.

The Problem diagram for the intersection controller problem built starting from these preliminary considerations and from the Context diagram of Figure 8.2 is shown in Figure 8.17.

As previously introduced, the controller has to support different operating mode. Each operating mode represents a different requirement for the controller. This section illustrates the requirements of each operating mode. Moreover,a machine specification that addresses the requirements is also introduced.

8.4.1 Fixed cycle operating mode

The fixed cycle operating mode (also known as pretimed or fixed time mode), is the simplest operating mode supported by the intersection controller.

In such a mode, all operating parameters of the signal are preset in the controller, which repeatedly executes the predefined pattern regardless of traffic con-

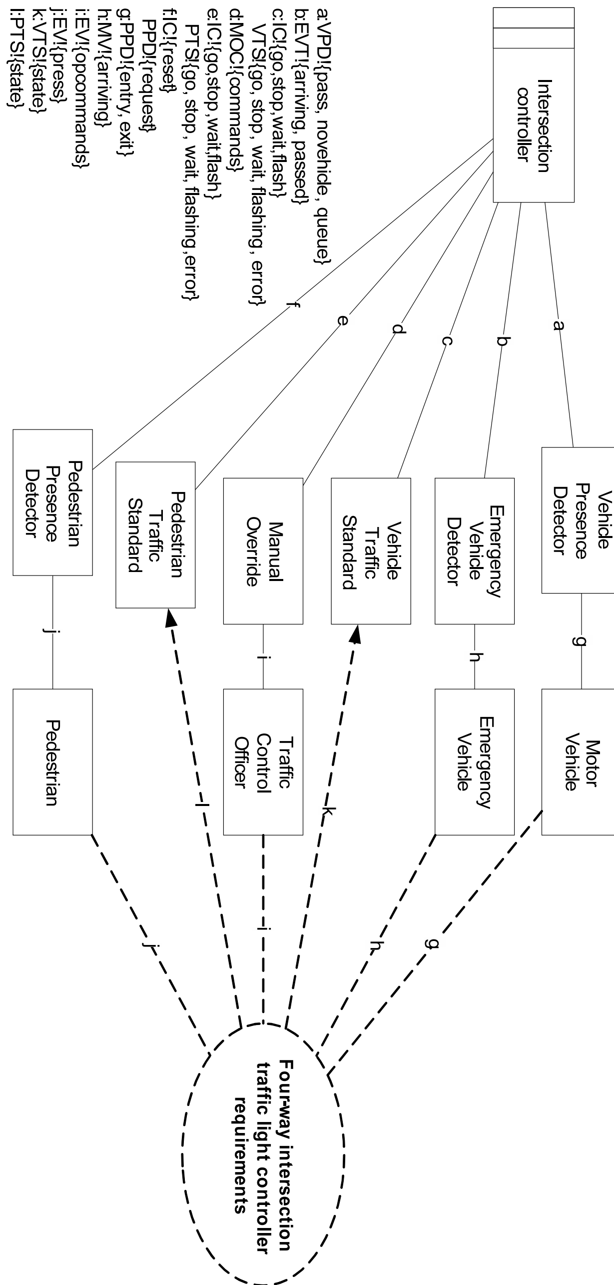

Figure 8.17: The Problem diagram for the intersection controller problem

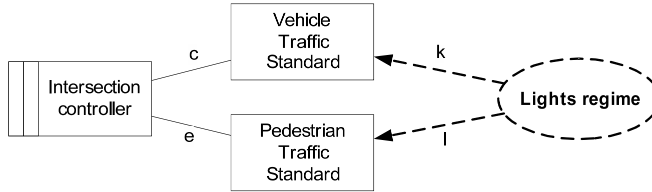

Figure 8.18: The Problem diagram for the fixed cycle operating mode

ditions. Notice that such patterns are defined by traffic engineers and they are based on historical data and experience.

Up to some years ago the fixed cycle mode was the most adopted solution to manage the motor vehicle traffic at road intersections in different countries.

The problem is described by the problem frame diagram shown in Figure 8.18. The problem is characterized by a machine domain, *Intersection Controller*, and by the causal domains *Vehicle Traffic Standard* and *Pedestrian Traffic Standard*. As specified by the problem frame diagram shown in Figure 8.18, such domains share phenomena that represent the commands (go, stop, wait, flash) that the controller sends to the traffic lights.

The requirements of the problem predicate on the sequences of the commands that the intersection controller issues to the traffic lights (both pedestrian and vehicular) and on the scheduling of such commands by defining the duration of the single phases of the intersection system.

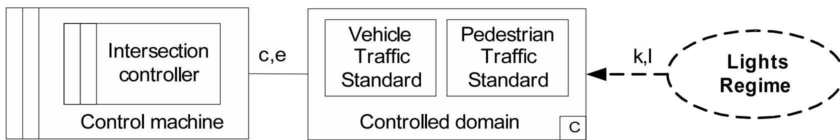

Figure 8.19: The Fixed cycle operating mode problem fits the Required behavior frame

As shown in Figure 8.19, the problem is an instance of the Required behavior frame proposed by Jackson [37]. *Intersection Controller* plays the role of *Machine controller*, while the set composed of *Vehicle Traffic Standard* and *Pedestrian Traffic Standard* the role of *Controlled domain*.

The requirements are described by the `stm` shown in Figure 8.20. At starting time the traffic standards of all the approaches are flashing. At a specific time of the day, specified by the condition *when(now=startWorkingT)*, the transition to the state *Stop-Go Walk-DNWalk* fires. In this state, the current state of the vehicular traffic lights of the approach EW is *Stop*, the one of the approach NS is *Go*, and the current state of the pedestrian traffic lights of the approach EW is *Walk* and the one of the approach NS is *DNWalk*.

now returns the number of milliseconds elapsed from a given date, hence in order to represent that every day at a given time the traffic light has to start working, one should specify a complex time condition. First of all, it is necessary to calculate the number of milliseconds elapsed from the 00.00 of the current

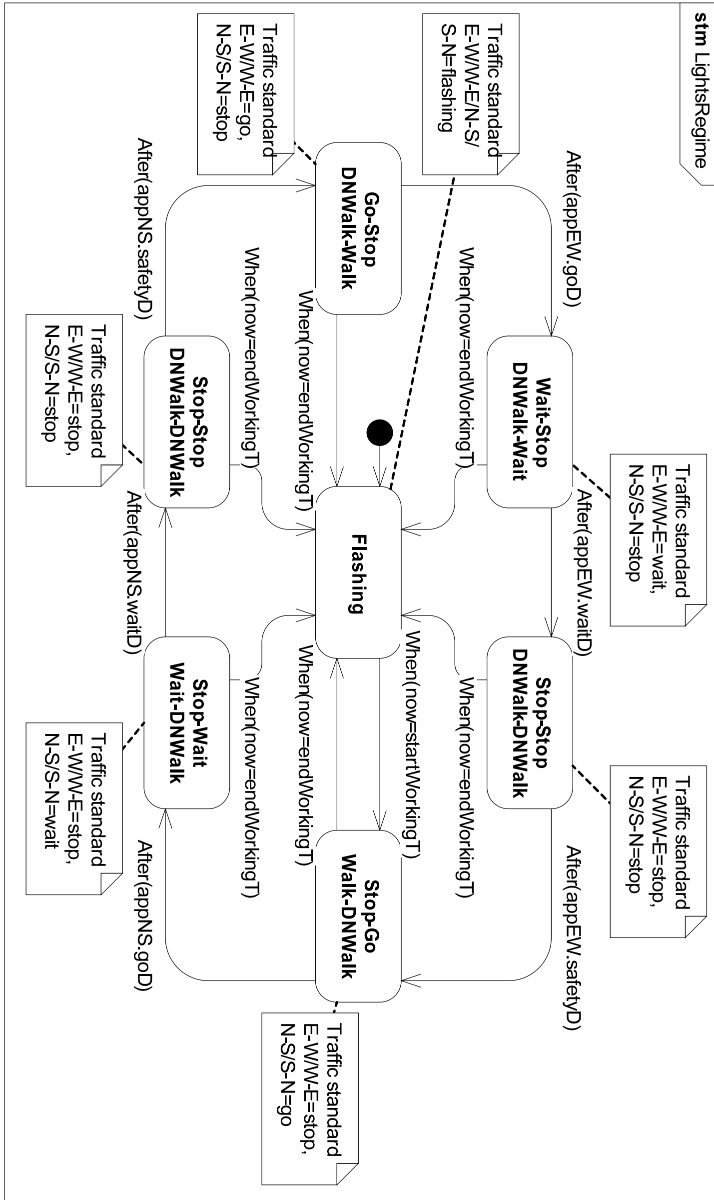

Figure 8.20: The stm that expresses the requirements for the fixed cycle operating mode problem

day. This can be obtained by considering the rest of the integer division of the current time instant *now* and the number of milliseconds in a day: $now\ mod\ 24 * 60 * 60 * 1000 = now\ mod\ 84.600.000$ Then, the trigger time has to be expressed in milliseconds (the number of milliseconds elapsed from the 00.00 a.m). As an example suppose that the traffic light starts to work at 6.05 a.m. Such time expressed in milliseconds becomes $6 * 60 * 60 * 1000 + 5 * 60 * 1000 = 21.900.000$. Hence, the resulting time expression becomes $now\ mod\ 84.600.000 = 21.900.000$

Notice that, similar expressions strictly depend on time conditions that one need to specify. The condition expressed on the transition should become

$$when(f(now) = g(initT))$$

, where f and g are *ad hoc* functions defined by the traffic engineers for specific needs. Notice that, in order to keep our specification as simple as possible, no particular function is specified.

The `stm` may evolve by passing to the state *Stop-Wait Wait-DNWalk* or returning to the state *Flashing*. The first transition fires *appNS.goD* time instants after the entrance of the state *Stop-Go Walk-DNWalk*. The second one fires in case is reached *endWorkingT*. Notice that all the other states of the `stm` can be reached by means of similar conditions. Some considerations must be expressed about the time conditions specified on the `stm`. *AppNS.goD* and *AppEW.goD* represent the duration of the permanency in the state *Go* of the traffic lights of the approach NS and EW, respectively. Similarly, *AppNS.waitD* and *AppEW.waitD* represent the duration of the permanency in the state yellow, and *AppNS.safetyD* and *AppEW.safetyD* represent the duration of the phase where all the traffic lights of all the approaches are in the state *Stop*. Notice that this duration is defined in order to decrement the possibility of accidents caused by motor vehicles that start crossing the intersection when their traffic light is at the end of the permanency in the state *Yellow*. *SafetyT* indicates the time required for a motor vehicle (moving at a given speed) to cross the intersection.

As described by the `stm` diagram, the required behavior of the controller is specular for the approaches. When the traffic lights of one approach are in the state *Green*, the corresponding state of the traffic lights of the other approach is *Red*.

Given an approach, the duration of the red light of its traffic lights is given by the sum of *safetyD* (of the same approach) and *goD* of the opposite approach.

The machine specification describes the behavior that the intersection controller has to follow in order to satisfy the requirements. The specification is provided in the `stm` diagram shown in Figure 8.21.

Notice that the `stm` diagrams of Figure 8.20 and Figure 8.21 are similar. They have the same number of states and furthermore, the conditions that trigger the transitions are the same for both machines. The main difference between such `stms` regards the actions that are performed by the machine when the transitions fire. As an example, when the transition from the state *S1* and *S2*, the controller has to issue ad-hoc commands to the traffic lights of both approaches. It has to send commands to set the vehicular traffic lights of the approach NS to the state *Go*, and those of the approach EW to the state *Stop*. This is done by generating

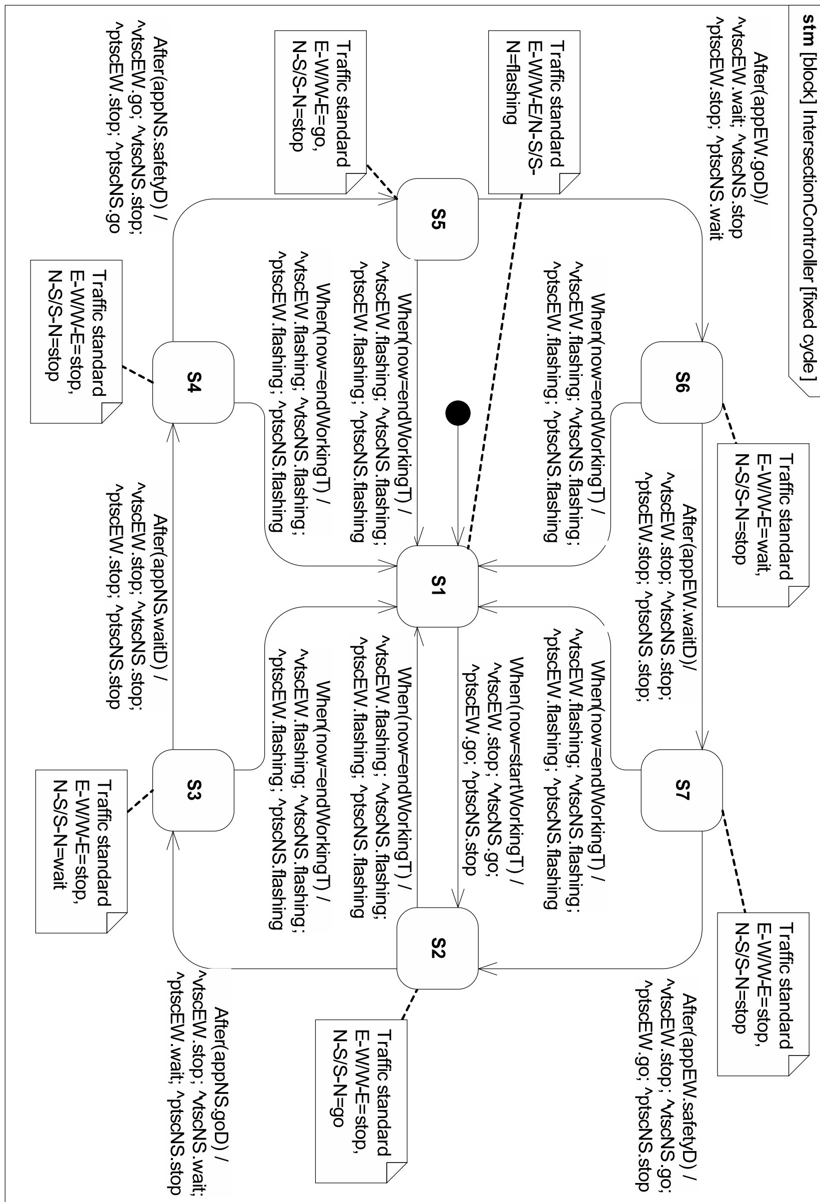

Figure 8.21: The `stm` representing the machine specification for the fixed cycle operating mode problem

signals of type *TrafficStdCmd*, that are issued by means of the ports *vtscEW*, *vtscNS*, *ptscEW* and *ptscNS* of *IntersectionController*.

8.4.2 Fixed cycle pedestrian actuated operating mode

The fixed cycle pedestrian actuated operating mode problem illustrates the homonymous operating mode of the intersection controller. Such operating mode is an extended version of the previously introduced Fixed cycle. The extension consists in considering the requests of the pedestrian in order to anticipate the passing to the state *Go* of the pedestrian traffic light that regulates the crossing of the crosswalks.

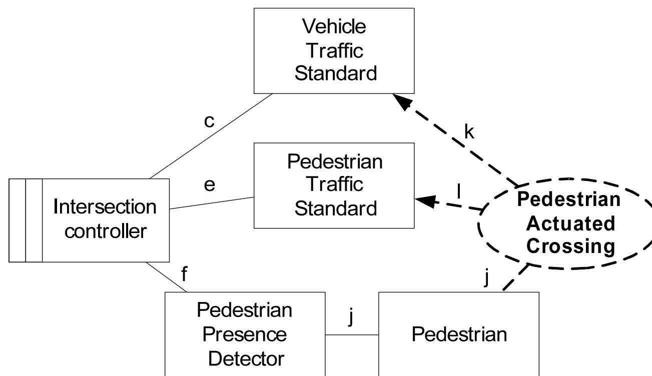

Figure 8.22: The Problem diagram for the fixed cycle pedestrian actuated operating mode

The problem is described by the problem frame diagram shown in Figure 8.22. The problem is characterized by a machine domain Intersection controller, by three causal domains named *Pedestrian Presence Detector*, *Pedestrian Traffic Standard* and *Vehicle Traffic Standard*, and by a biddable domain *Pedestrian*.

The requirements of the problem state that in case a pedestrian is waiting at a crosswalk because of the pedestrian traffic light is set to red and he/she presses the request button, such request has to anticipate the pass to the green phase of the involved traffic lights.

A pedestrian may press the button in each phase of the system, but only the requests issued when the pedestrian is waiting because of the red state of the traffic light at the involved crosswalk, must be considered.

As shown in Figure 8.23, the problem is an instance of the Commanded behavior frame [37]: *Intersection controller* plays the role of *Controller*, while *Pedestrian Presence Detector* and *Pedestrian* represent the domain *Operator* and *Vehicle Traffic Standard* and *Pedestrian Traffic Standard* play the role of *Controlled Domain*.

The requirements are illustrated in the `stm` diagram shown in Figure 8.24. The `stm` diagram is equivalent to the one of Figure 8.20, with the exception of the states *Stop-Go Walk-DNWalk* and *Go-Stop DNWalk-Walk* and of the output transitions of such states.

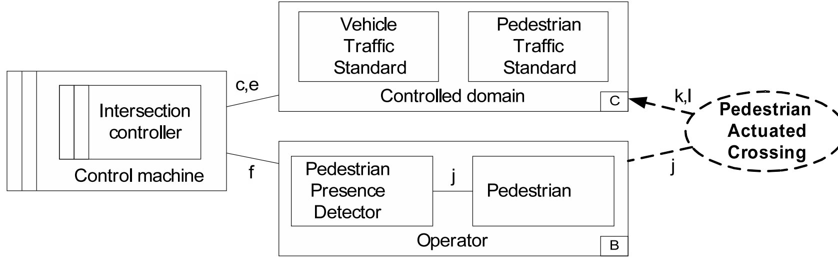

Figure 8.23: The Fixed cycle pedestrian actuated operating mode problem fits the Controlled behavior frame

Such states are composite, i.e., each of them contains a sub-machine that describes a particular behavior. Hence, let us consider the state *Stop-Go Walk-DNWalk*. The internal state machine is composed of two states named *NoPedReq* and *PedReqNS*. The first state represents a situation in which no pedestrian pressed the button. When the composite state is entered, also the internal state *NoPedReq* is entered and a local attributed *initT* is set to the current time. In case a pedestrian sends a request by pressing the button of one of the detectors of the approach, a signal of type *PedestrianRequest* is issued by means of the output ports *pr*. Such signal triggers the transition to the internal state *PedReqNS*. Once entered this state, a local attribute *reqT* is set to the current time. The internal state can be left after a period specified by the expression $(AppNS.goD - (reqT - initT)) * k$ where $k \in \Re$ and $0 < k < 1$. Notice that $AppNS.goD$ represents the duration of the green phase, while $reqT - initT$ the period spent from the entrance of the composite states. Hence the expression means that the waiting time is a percentage k of the time remaining to complete the interval $AppNS.goD$. Notice that in case the internal state is exited, also the composite state is left and the `stm` passes to the state *Stop-Wait Wait-DNWalk*. In case no pedestrian request is received, the composite state is left by means of the same transition used for the fixed cycle `stm`, i.e. the state is abandoned after a period $AppNS.goD$. The other composite state *Go-Stop DNWalk-Walk* is equivalent to the previous one with the exception of the allocation of the signal that indicates the request of the pedestrians, which in this case are allocated to the output ports of the detector positioned on the approach EW.

The machine specification is described by means of the `stm` diagram shown in Figure 8.25. Notice that also in this case the `stm` is quite similar to the one used for describing the requirements and the machine specification of the fixed cycle operating mode problem. The differences consist in the allocation of the signals that trigger the transitions and of the signals that are generated when a transition fires. In fact, in this case, I/O signals are allocated to the input and output ports of the Block *IntersectionController*. Notice that differently from the previous problem the transitions between the states *S7* - *S2* and *S4* - *S5* generate a signal *reset* that is used to reset the state of the pedestrian detectors as described in the previous sections by the `stm` diagram shown in Figure8.5.

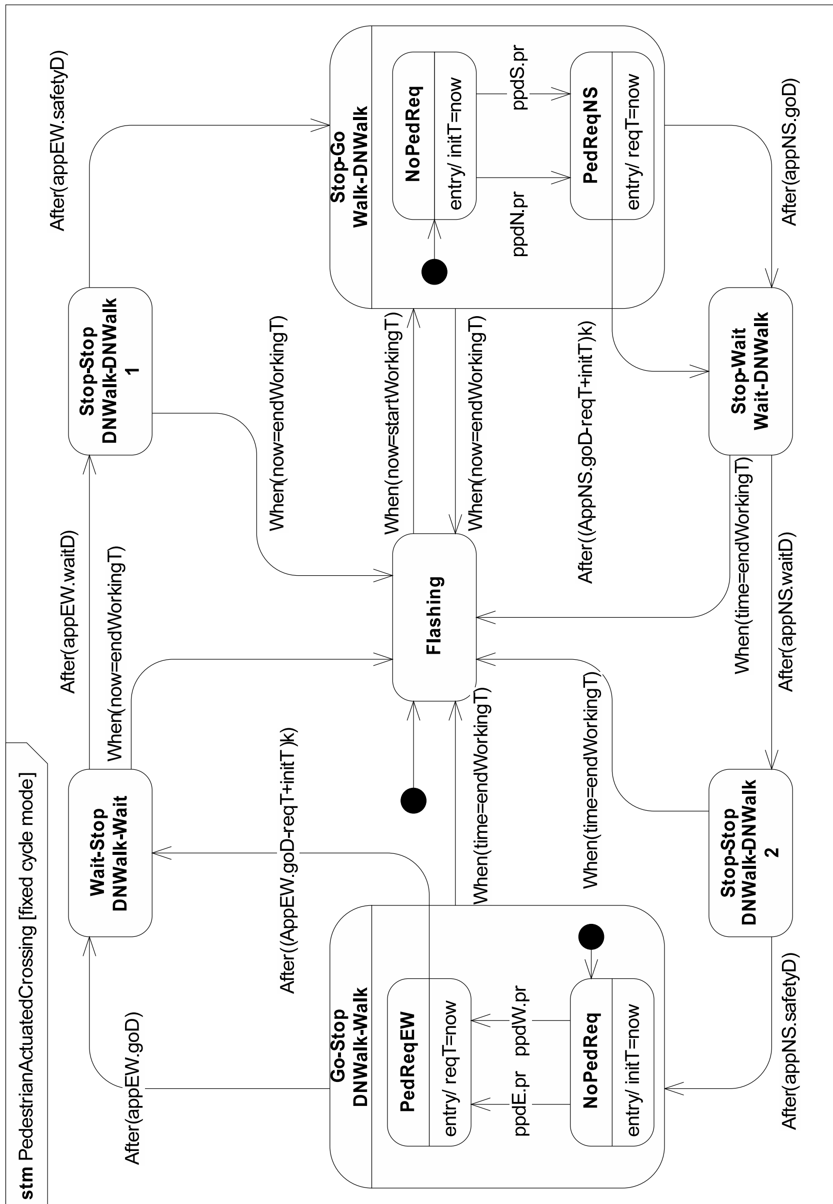

Figure 8.24: The `stm` that expresses the requirements for the fixed cycle pedestrian actuated operating mode problem

Figure 8.25: The stm representing the machine specification for the fixed cycle pedestrian actuated operating mode problem

8.4.3 Semi-actuated operating mode

The semi-actuated operating mode is one of the most applied operating mode for controllers that manage the traffic lights at the intersection of an arterial road and a secondary road.

This operating mode consists in providing the green signal to the road with the lower priority only in case there are motionless vehicles that are waiting for the green signal. Otherwise the green signal is provided to the main road.

In order to check the presence of the vehicles on the secondary street, a vehicle presence detector is placed next to the intersection on each semi-approach of the secondary street. Notice that, also pedestrians may influence the phases of the intersection controller by pressing the request buttons of the detectors that are placed at the end of the crosswalks of both approaches.

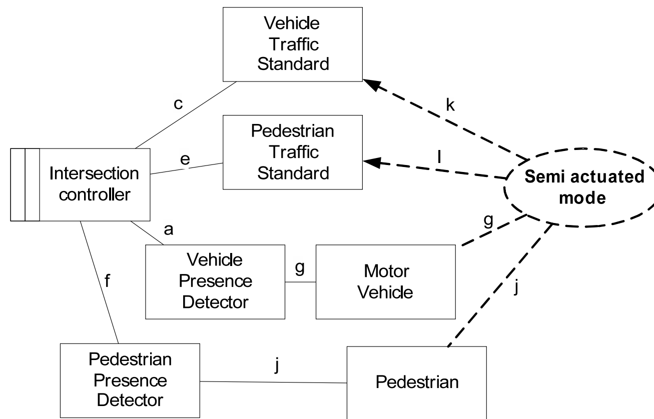

Figure 8.26: The Problem diagram for the semi-actuated operating mode

The semi actuated operating mode problem is illustrated in the problem frame diagram shown in Figure 8.26. The problem is characterized by the machine domain Intersection controller, by the causal domains *Vehicle Traffic Standard*, *Pedestrian Traffic Standard*, *Vehicle Presence Detector* and *Pedestrian Presence Detector*, and by the biddable domains *Vehicle* and *Pedestrian*.

As shown in Figure 8.27, this problem is an instance of the Commanded behavior frame [37]: *Intersection controller* plays the role of *Controller*, while *Pedestrian Presence Detector*, *Pedestrian*, *Vehicle Presence Detector* and *Vehicle* represent the domain *Operator* and *Vehicle Traffic Standard* and *Pedestrian Traffic Standard* represent the domain *Controlled Domain*.

The phenomena shared by *Control machine* and *Operator* are composed of the signals that *Vehicle Presence Detector* and *Pedestrian Presence Detector* issue to *Intersection Controller* in order to notify the presence of a pedestrian or of a vehicle. The phenomena shared by *Control machine* and *Controlled domain* are composed of the set of commands that *Intersection Controller* issues to both the vehicle and pedestrian traffic lights.

The requirements of the problem are illustrated by means of the `stm` diagram shown in Figure 8.28. According to the topology of the intersection shown in

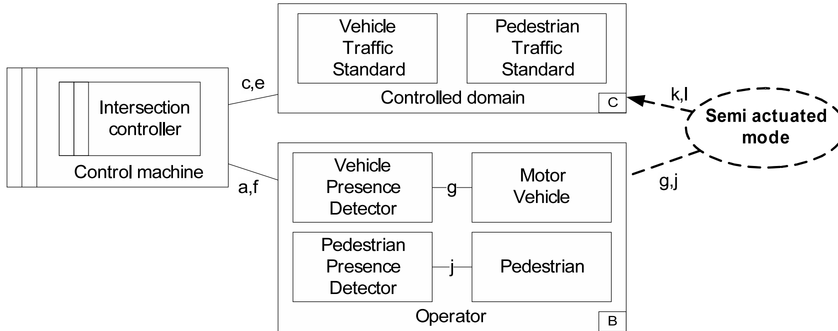

Figure 8.27: The semi actuated operating mode problem fits the Controlled behavior frame

Figure 8.1, the approach EW is the secondary street while the approach NS is the high priority arterial road.

Notice that the expected behavior of the system is quite similar to the one illustrated for the fixed cycle pedestrian actuated problem. Hence, the `stm` is similar to the one that defines the requirements of the other problem. Such `stms` simply differ for the states *Stop-Go Walk-DNWalk* and *Go-Stop DNWalk-Walk* and for some output transitions from such states. Hence, in order to avoid the repetition of the description, only new aspects of this mode are illustrated.

The state *Stop-Go Walk-DNWalk* is a composite state composed of two sub-machines. The first sub-machine, which describes the processing of the pedestrian requests, is equivalent to the one used for the requirements of the previous problem, while the second sub-machine illustrates possible states of the secondary street. Consider that the composite state represents a condition for which the secondary street has the red signal while the main road has the green signal. Hence, once entered the composite state, the first sub-machine enters the state *NoPedReq* while the second one enters the state *NoVehReq*. *NoVehReq* indicates that the presence detector of the secondary street has not notified anything yet to the intersection controller. In case such detectors generate a signals *queue* and the `stm` passes to the state *VehReqEW*. Such state indicates that there is at least a vehicle that has been still for some time on the loop detectors of the semi-approaches of the secondary road EW. For a detailed description of the behavior of the detector see the `stm` of Figure 8.6. The state *VehReqEW* is exited *appEW.ReqVWT* time after the entrance. Such transition is also an output transition for the composite state *Stop-Go Walk-DNWalk*. Notice that such state can be left by means of three different conditions: 1)in case a pedestrian pressed the button of a pedestrian detector, 2)in case there are vehicles on the secondary road, and 3) in case *AppNS.goD* time elapsed from the entrance (where *AppNS.goD* is the duration of the green phase of the approach NS).

The state *Go-Stop DNWalk-Walk* is another composite state composed of two sub-machines. Such states indicate a condition for which the secondary road has the green signal while the arterial road has the red one. As for the other composite state, the first state machine handles the requests of the pedestrian

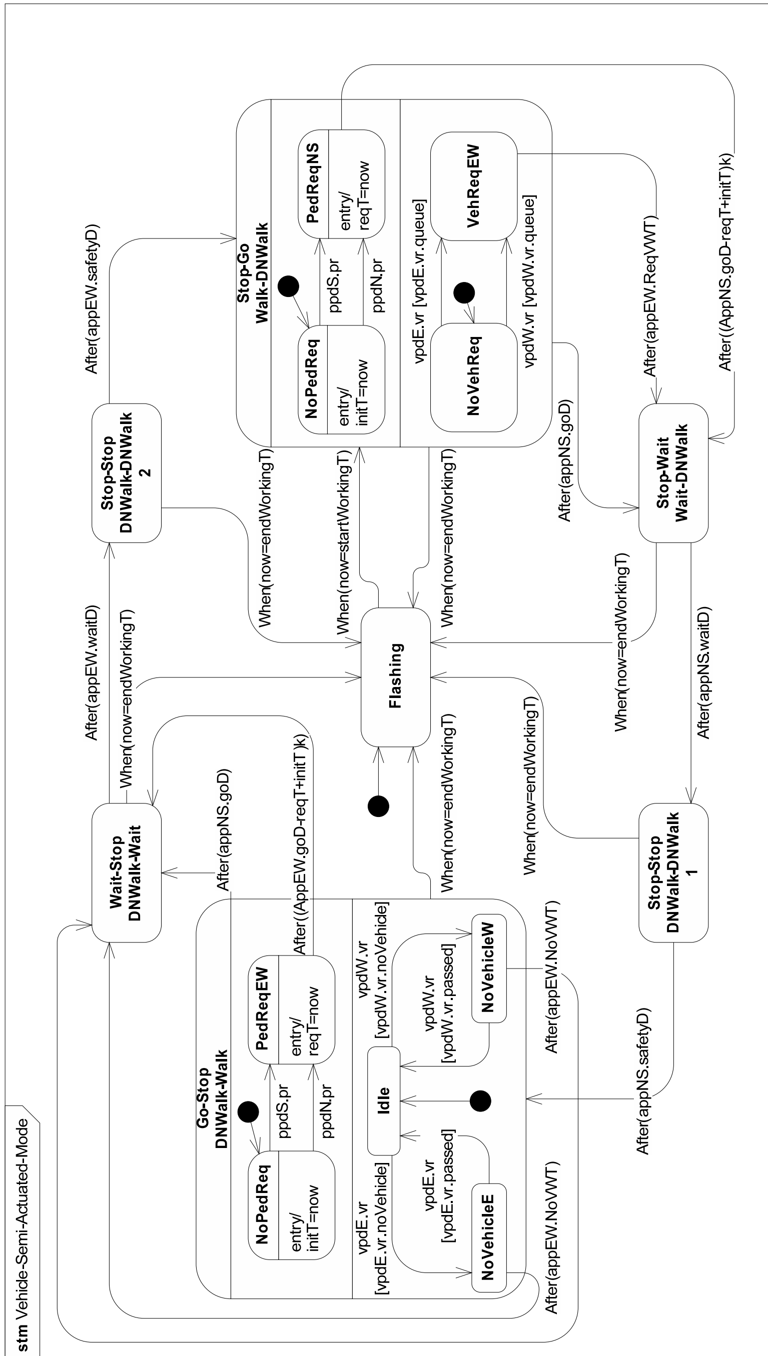

Figure 8.28: The `stm` that expresses the requirements for the semi-actuated operating mode problem

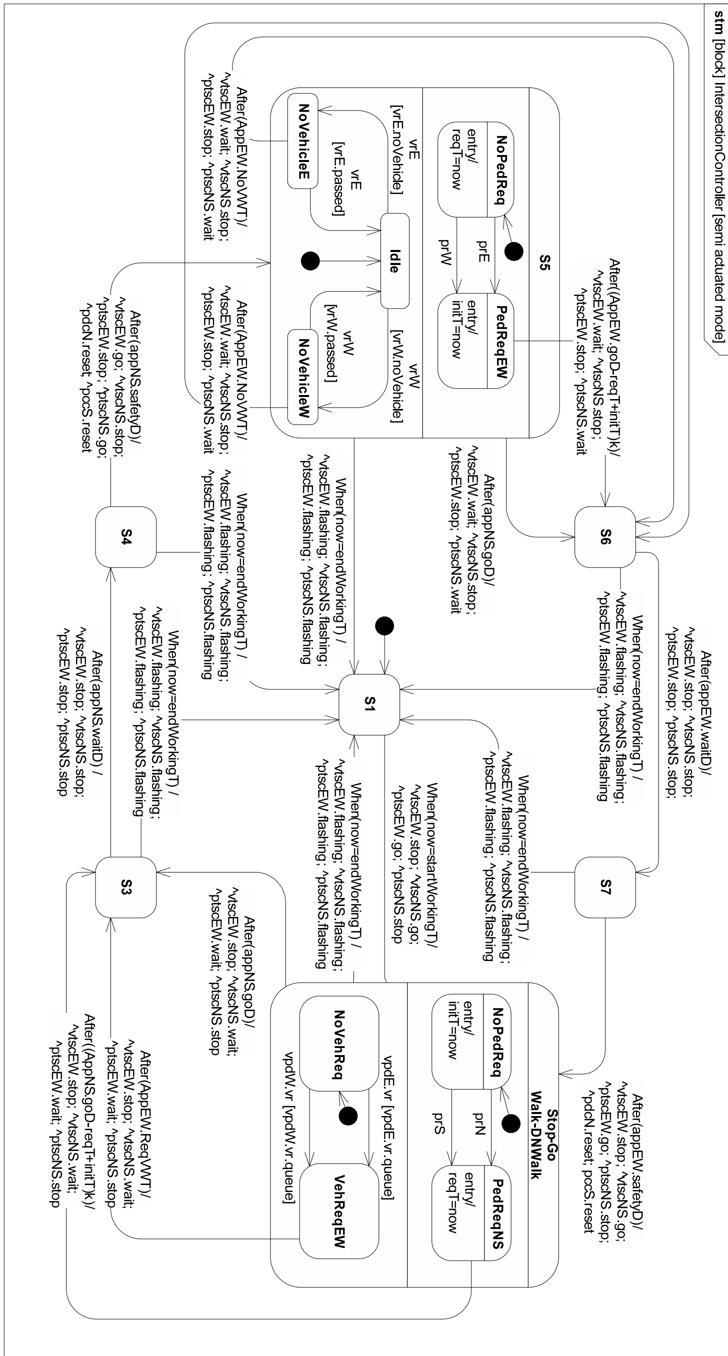

Figure 8.29: The `stm` representing the machine specification for the semi-actuated operating mode problem

and is equivalent to the one illustrated in the requirements of the fixed cycle mode. The second `stm` keeps track of the state of the secondary road. At entry time in the composite state, the state *Idle* of the second state machine is entered. Such internal state indicates that no particular signal has been notified so far by the detector of the secondary street. In case a signal *noVehicle* is generated by one of the detectors of the semi-approaches, the `stm` passes to the state *NoVehicleE* or *NoVehicleW*. Then, in case no other signal is generated for *appEW.NoVWT* time units, such states (and also the composite state) are exited. In case a signal *passed* is generated by one of the detectors the sub-machine returns to the state *Idle*. *Go-Stop DNWalk-Walk* can be exited in case three different conditions are satisfied: 1) in case *appEW.goD* time elapsed from the entrance of the composite state, 2) in case no vehicle has been detected for *appEW.NoVWT* time on one of the semi-approaches, 3) in case a pedestrian of one of the crosswalks of the approach EW presses the button of the presence detector.

Since the rest of the requirements is equivalent to the one of the previous problem, they are not further illustrated.

The specification of the machine are defined by means of the `stm` diagram shown in Figure 8.29. Also in this case such `stm` differs from the one that describes the requirements exclusively for the allocation of the signals (here they are allocated to the I/O ports of *IntersectionController*), and for the commands that are issued whenever a transition fires. As in the previous problem the commands are issued to the traffic lights in order to change their state and to the pedestrian detector to reset possibly pending pedestrian requests. Notice that in case a state has multiple input transitions, the commands that are sent when such transitions fire are the same.

8.4.4 Fully-actuated operating mode

The fully-actuated operating mode is an advanced operating mode that allows a controller to operate the state of the traffic lights depending on the actual traffic conditions. More specifically, depending on the traffic conditions the duration of the different phase of the intersection are automatically regulated.

In this case the vehicular presence detectors are positioned on each semi approach next to the intersection. As in the previous problem, such detectors recognize when there is no vehicle, when a vehicle is on the sensor and when a vehicle passed the sensor.

In the fully actuated mode no priority mechanism is applied, hence, all the streets are equally considered, and the duration of the phases are exclusively calculated starting from the data provided by the sensors. Notice that such data indicate the actual condition of the approaches (i.e., whether there are queues, a regular flow of vehicles or no vehicle along the approaches) and are also used to calculate statistics such as the rate of vehicles that are traveling along an approach. Moreover, consider that also in this operating mode the requests of the pedestrian are considered and may influence the duration of the phases.

The Problem frame diagram that illustrates the problem is shown in Figure 8.30. Notice that such diagram is equivalent to the one that illustrates the semi-actuated mode, hence it is not further illustrated. As shown in Figure 8.31,

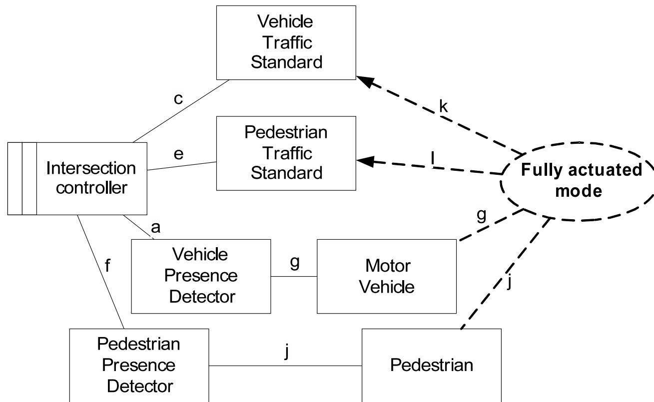

Figure 8.30: The Problem diagram for the fully actuated operating mode

also this problem is an instance of the Commanded behavior frame [37]. Since the involved domains and phenomena are the same that were illustrated in the previously described problem, their description is not repeated here.

The requirements of the problem are described by means of the `stm` diagram shown in Figure 8.32.

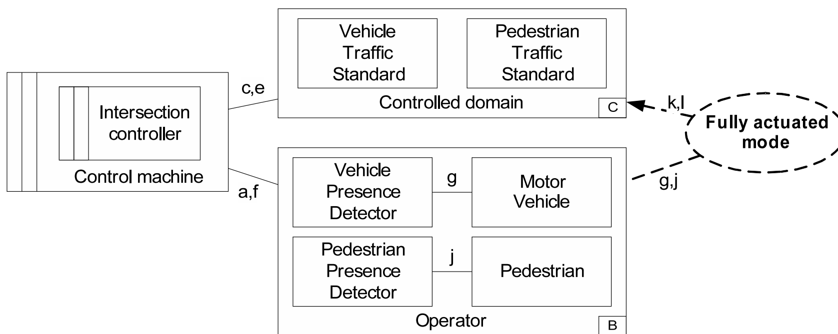

Figure 8.31: The Fully actuated operating mode problem fits the Controlled behavior frame

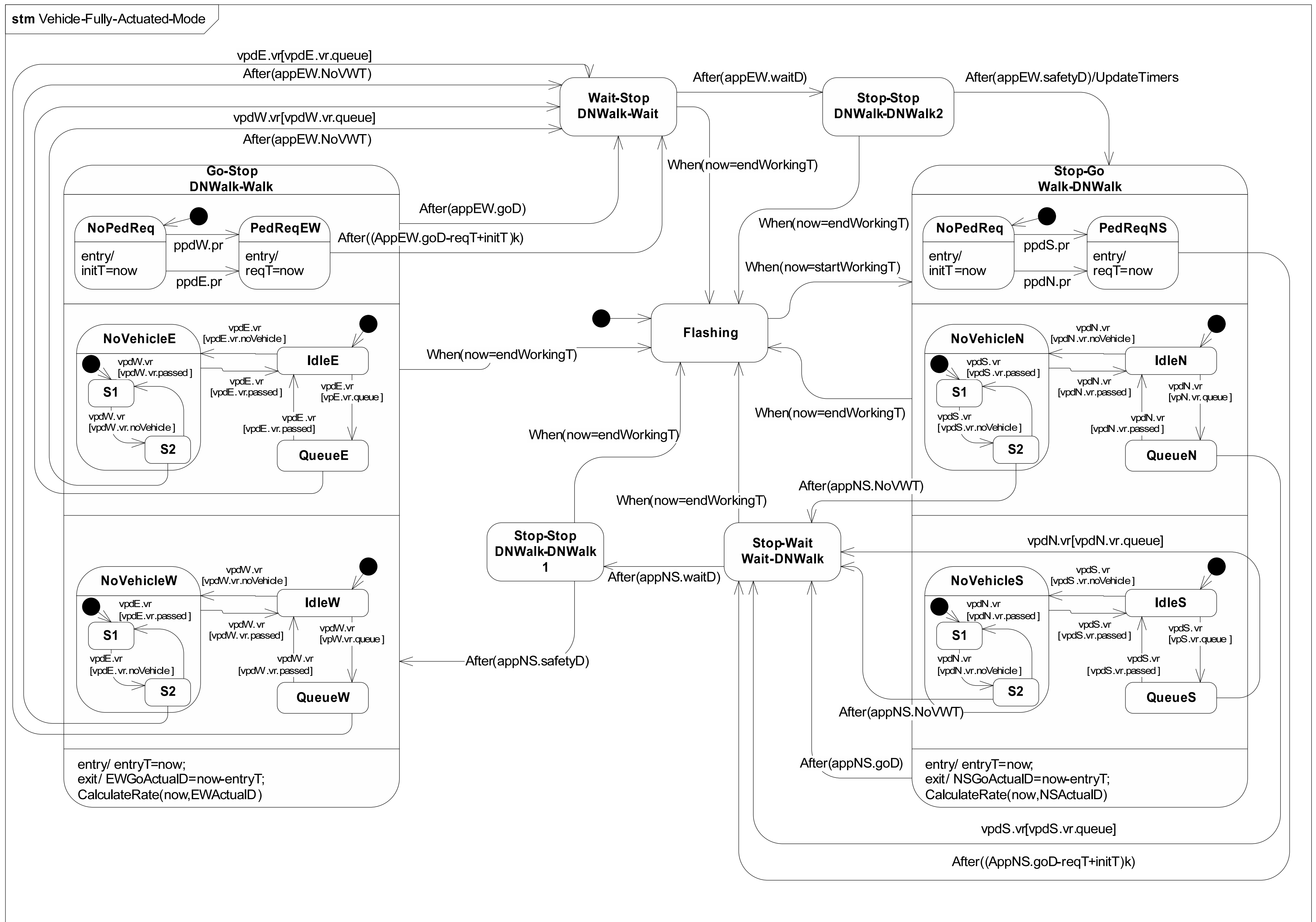

Figure 8.32: The stm that expresses the requirements for the fully-actuated operating mode problem

Notice that the phases of the system are the same of all the other problems, and also the duration of most of the phases is the same. As for the other problems, the differences are exclusively related to the states *Stop-Go Walk-DNWalk*, *Go-Stop DNWalk-Walk* and to their output transitions.

Such states represent the most important phases of the whole cycle, as they are the only phases during which the vehicles can cross the intersection. The analysis of the traffic conditions is centered on such states.

Let us consider the composite state *Stop-Go Walk-DNWalk*. This state represents a phase of the system during which the semi approaches of the road EW have the red signal, while the ones of the street NS have the green. During this phase the traffic along the approach NS may have different behavior. We could have one of the semi approaches that is completely empty, while on the other one there is a regular flow of vehicles. In an alternative scenario both the semi-approaches could be characterized by a queue that forbids the movement of the cars. Notice that a combination of every possible state (queue, no vehicle or regular traffic) for both the semi-approaches can characterize the actual state of the whole road.

The duration of the phase is reduced in case particular conditions are verified. More specifically, 1) in case there is queue or 2) in case there is no vehicle on a semi approach. This requirement is described by means of two sub-machines. The first `stm` describes the actual state of the semi approach N, while the other one describes the state of the semi approach S. Let us consider the first one. It is composed of three states named *NoVehicleN*, *IdleN* and *QueueN*. *IdleN* represents a regular traffic condition, while *QueueN* a queue condition and *NoVehicleN* represents a situation with no vehicle on the semi approach. When entering the composite state, the state *IdleN* is entered. The `stm` passes to the state *QueueN* in case a signal *queue* is generated by the vehicle presence detector, while it passes to the state *NoVehicleN* in case the detector generates the signal *noVehicle*. Notice that once entered such states, whether the detector notifies the transit of a vehicle by generating the signal *passed*, the `stm` returns to the state *IdleN*. *NoVehicleN* is a composite state composed of another state machine. Such a sub-machine is used to keep track of the actual condition of the semi-approach S and is composed of the states *S1* and *S2*. *S1* represents the condition of regular traffic or queue on the semi-approach S, while *S2* describes the condition of no car on the semi approach.

The submachine that describes the condition of the semi approach S is equivalent to the one just presented.

Stop-Go Walk-DNWalk is composed of a third sub-machine that manages the requests of the pedestrians at the crosswalks of the approach NS. This state machine is equivalent to the one introduced by the semi actuated problem, hence it is not further commented.

Stop-Go Walk-DNWalk can be exited by means of a direct output transition (from the composite state) and output transitions of the internal states *PedReqNS*, *QueueN*, *QueueS* and *S2* of the composite state *NoVehicleN* and *NoVehicleS*.

Exiting from the state *PedReqNS* indicates that a pedestrian of the approach NS sent the request to cross the crosswalk. At pressure time, the pedestrian has to wait a percentage k of the time that remains before the passage to the following phase.

Exiting from the states *QueueN* and *QueueS* is triggered by a queue signal generated by the vehicle detector of the opposite semi-approach. This means that in case both the semi approaches are congested by traffic (notice that a queue signal is generated in case there is a vehicle motionless on the detector), the current phase must be stopped and the system has to pass to the following one.

Exiting from the state *S2* indicates the opposite situation, i.e., there has been no vehicle on both the semi approaches for *AppNS.NoVWT* time units. This indicates that in case there is no machine it is useless to keep active the current phase, it is better to stop it and to pass to the following one.

Stop-Go Walk-DNWalk can be exited also by means of a further direct output transition. Such transition fires *AppNS.goD* time units after the entrance of the composite state. Notice that *AppNS.goD* represents the actual duration of the phase and is continuously updated depending on the traffic conditions of the approaches.

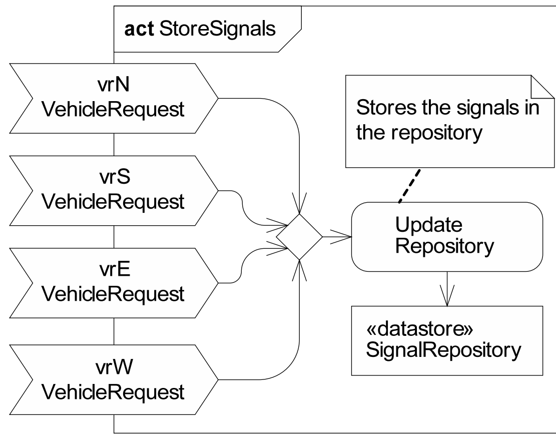

Figure 8.33: The `act` diagram that describes the action *StoreSignals*

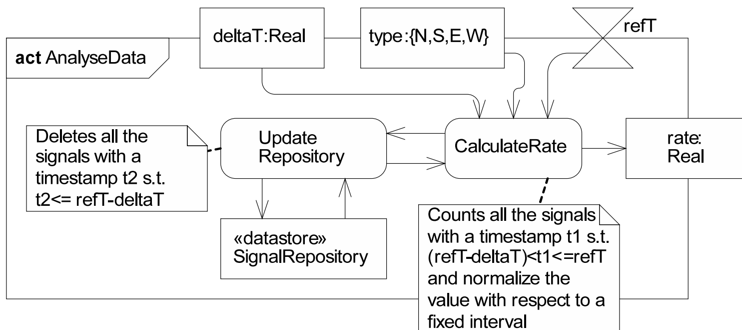

Figure 8.34: The `act` diagram that describes the action *AnalyseData*

The update is performed by means of an articulated process essentially based on the transit rate of the vehicles in both the approaches. First of all, as specified by the activity *StoreSignals* shown in Figure 8.33, whenever a signal of type *VehicleRequest* is generated by one of the vehicle detectors, such signal is stored in the repository *SignalRepository* by means of the action *UpdateRepository*. The transit rate is calculated by the action *AnalyseData* illustrated in the `act` diagram of Figure 8.34. The action receives in input 1) a value *deltaT*, which specifies the length of the period to analyse, 2) a value *refT*, which indicates the reference time of the analysis, and 3) a value *type*, which specifies the allocation of the signals to be considered, i.e., it specifies the interested semi approach. Such values are passed to the action *CalculateRate*, which in turn, by means of the action *UpdateRepository*, 1) counts all the signals with a timestamp t (with $(refT - deltaT) < t \leq refT$) generated by the detector of the specified semi-approach, 2) normalizes the value with respect to a fixed interval and 3) deletes from the repository all the signals with a timestamp *t2* such that $t2 \leq refT - deltaT$.

The input parameters for *AnalyseData* are intrinsic properties of the phase *Stop-Go Walk-DNWalk*. More specifically, *deltaT* is set to the duration of the whole phase, *refT* to the timestamp value at exit time from the state, and *type* to the identifier of the semi approach N or S. The actual duration of the phase is calculated by using two attributes named *entryT* and *NSGoActualD*. The first attribute is set to the current time at the entrance of *Stop-Go Walk-DNWalk*. The second one is set , at exit time, to the difference between the current time and the entrance time (notice that the resulting value represents the actual duration of the phase).

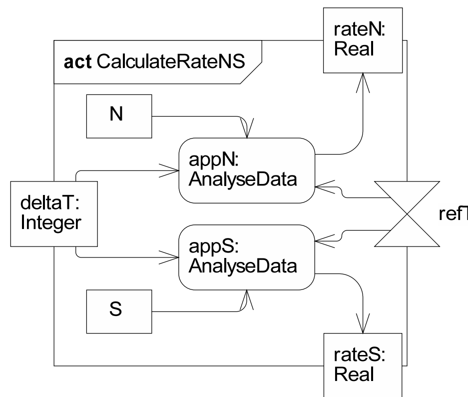

Figure 8.35: The `act` diagram that describes the action *CalculateRate*

Finally, at exit time, the action *CalculateRateNS* is invoked in order to calculate the appropriate rate for each semi approach. Such action is described by the `act` diagram shown in Figure 8.35. The action is characterized by the input nodes *deltaT*, and *refT*. Such nodes are allocated to the attribute *NSActualGoD* and to the current timestamp value at the exit time from the composite state. *CalculateRateNS* invokes the action *AnalyseData* for both the semi approaches N and S. The resulting values *rateS* and *rateN* represent the actual traffic rate

condition for the whole approach NS.

Activities and state machines equivalent to the one used for the phase *Stop-Go Walk-DNWalk* are used to describe the phase *Go-Stop DNWalk-Walk*. Such description is not reported here.

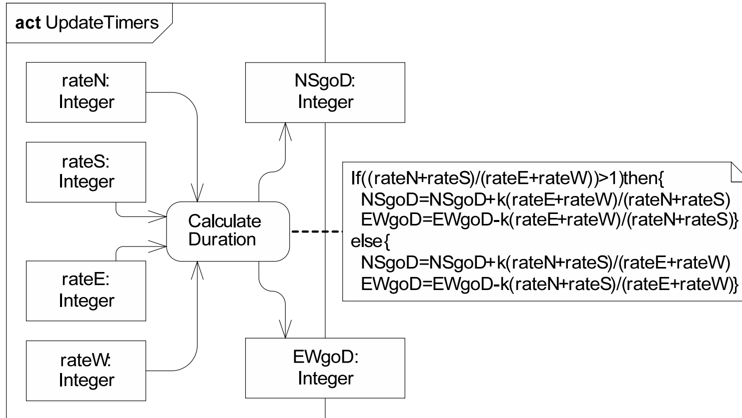

Figure 8.36: The `act` diagram that describes the action *UpdateTimers*

Once the rates of the semi approaches of the the roads are calculated, the actual value of expected duration of the phases *Stop-Go Walk-DNWalk* and *Go-Stop DNWalk-Walk* are updated. This is done immediately before entering the state *Stop-Go Walk-DNWalk* by invoking the action *UpdateTimers*. The action is described by the `act` diagram shown in Figure 8.36. *UpdateTimers* receives in input the rate of all the semi-approaches and invokes *CalculateDuration*, an action whose effects are described by the annotation shown in Figure 8.36. More specifically, the duration of the green phase of each approach is incremented or decremented of a value that depends on the traffic rates of the approaches. Notice that, the resulting values *NSgoD* and *EWgoD* are allocated to the value *AppNS.goD* and *AppEW.goD*.

As for the previously described problems, the description of the machine specification of the fully actuate mode problem can be defined by means of a `stm` characterized by the same states and transitions of the `stm` used to illustrate the requirements. As in the case of the previously described problems, the only differences between such `stm`s are related to the allocation of the I/O signals (in this case, such signals are allocated to the I/O ports of IntersectionController) and to the commands that are issued at firing time of each transition. Moreover, for each state, the activities invoked by its input transitions are the same activities invoked by the input transitions of the `stm` that depicts the semi-actuated mode machine specification.

Since the resulting description does not provide new aspects with respect to the what specified by the requirements, the machine specification is not reported.

8.4.5 Manual mode

This section describes the manual operating mode problem. In manual operating mode, the control of the intersection is handed over to an operator, which issues commands to the traffic lights of both approaches by means of a console.

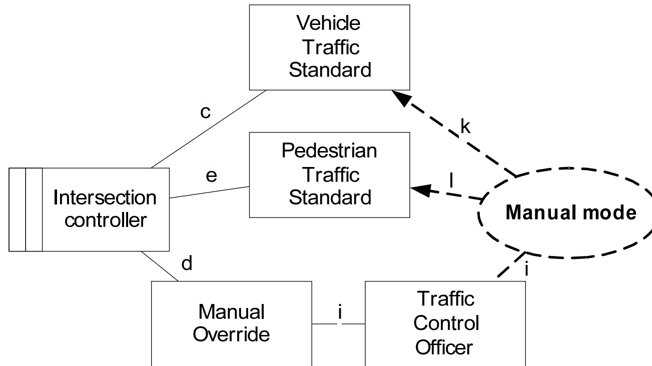

Figure 8.37: The Problem diagram for the manual operating mode problem

The operator is free to decide the duration of the green signal on the approaches of the intersection, to turn off the traffic lights (setting the flashing yellow) and to restore the traffic lights. The phases of the intersection are the same that were considered for all the other operating modes. Moreover notice that the duration of the other phases is preset and cannot be changed.

The operator is also allowed to set the duration of the phases, to change the operating mode, to set the starting and ending time of the controller.

In manual operating mode, neither the requests of the pedestrians nor the signals of the vehicle detectors are considered.

The structure of the problem is described by means of the problem frame diagram shown in Figure 8.37.

The problem is characterized by 1) the biddable domain *Traffic Control Officer*, which represents the operator that manually configures the state of the intersection; 2) by the causal domain *Manual Override*, which represents the console used by the operator to issue commands to the intersection controller; 3) by the machine domain *Intersection Controller*, which receives the commands of the operator and propagates orders to the traffic standards; finally, 4) by the causal domains *Pedestrian Traffic Standard* and *Vehicle Traffic Standard*, which represent the traffic lights of the intersection.

As shown in Figure 8.38, the problem is an instance of the Commanded behavior frame [37]: *Intersection controller* plays the role of *Control machine*, while *Manual Override* and *Traffic Control Officer* play the role of *Operator* and *Vehicle Traffic Standard* and *Pedestrian Traffic Standard* the role of *Controlled Domain.*

The phenomena shared by *Control machine* and *Operator* are composed of the commands that *Manual Override* sends to *Intersection Controller* in response to what specified by *Traffic Control Officer* by means of the console. The phenomena shared by *Control machine* and *Controlled domain* are the set of the commands

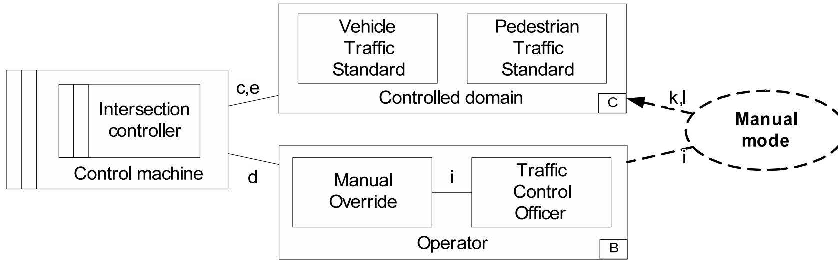

Figure 8.38: The Manual operating mode problem fits the Controlled behavior frame

that *Intersection Controller* sends to the vehicle and pedestrian traffic lights.

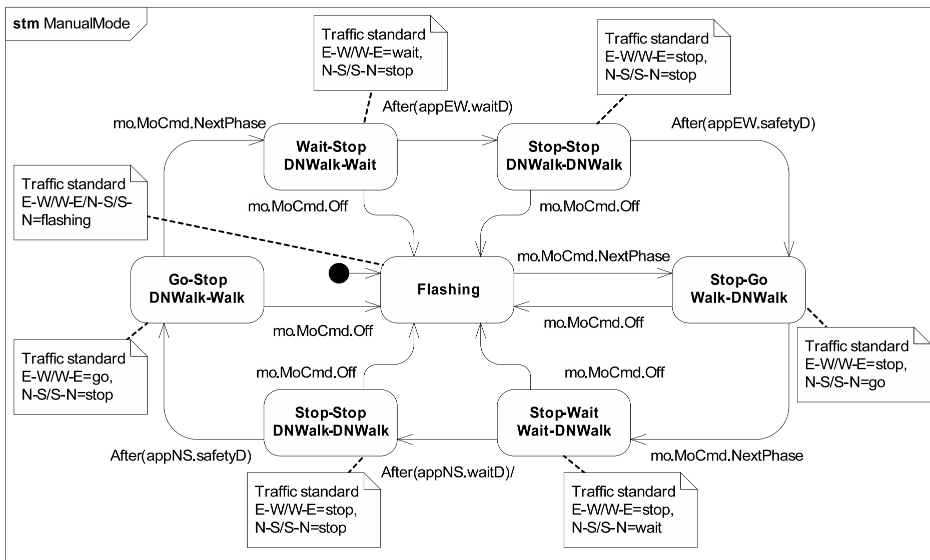

Figure 8.39: The `stm` diagram that describes the requirements of the problem Manual operating mode

The requirements of the problem are presented by means of the `stm` diagram shown in Figure 8.39.

At starting time the `stm` enters the state *Flashing*, where all the traffic lights (pedestrian and vehicular) are yellow flashing. The transition to the following phase *Stop-Go DNWalk-Walk* is triggered by the command *NextPhase* that in turn is issued by the operator by means of the console *Manual Override*. The permanency in this state completely depends on the intentions of the operator, i.e., the intersection controller can not force the passing to the following phase. The transition to the following phase *Stop-Wait Wait-DNWalk* can be triggered exclusively by another command *NextPhase*. Conversely, the passage between the phases *Stop Wait Wait-DNWalk* and *Stop-Stop DNWalk-DNWalk* as well as

the one between *Stop-Stop DNWalk-DNWalk* and *Go-Stop DNWalk-Walk* is automatically managed by the controller. This is done in order to assure the correct duration of transit phases that are fundamental for safety reasons. The transition are triggered by the same temporal conditions that were used in the previously presented operating modes.

Once entered one of the state of the stm, in case the operator decides to turn off the traffic lights of all the approaches it can do it by issuing the command *Off* by means of the console. This condition is expressed by means of the transitions to the state *Flashing*.

The description of the remaining states of the stm is specular to the one previously presented, hence, it is not provided here.

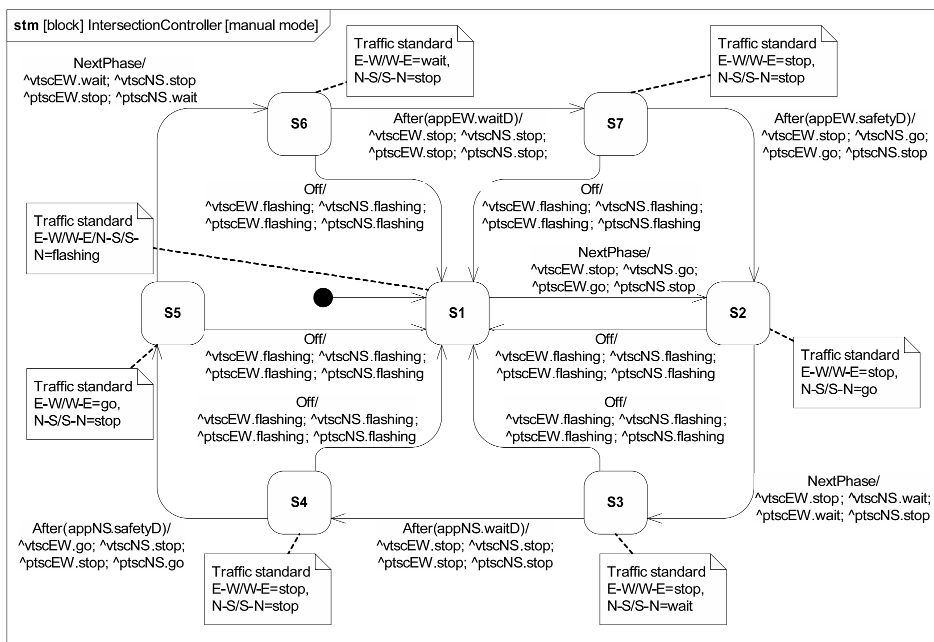

Figure 8.40: The stm diagram that describes the machine specification of the problem Manual operating mode

The machine specification of this problem is described by means of the stm diagram shown in Figure 8.40. As for the other problems, the stm is an extended version of the state machine that described the requirements of the problem. The states, representing the phases of the intersection, are the same, as well as the transitions, which in this case are triggered by signals allocated to the ports of the Block *IntersectionController*. Notice that whenever a transition fires the commands issued to the traffic lights are those illustrated in the other problems.

8.4.6 Preempted mode

This section describes the emergency preempted operating mode problem. Traffic signal preemption is a type of system that allows the normal operation of traffic lights to be preempted. The most common use of these systems is to give priority to emergency vehicles by changing traffic signals in the path of the vehicle to green while stopping conflicting traffic.

Preempted systems generally operate by means of radio signal, infrared signals or by visible strobe lights. Each emergency vehicle is equipped with an emitter, a device that emits visible flashes of light or radio or infrared pulses at a specified frequency. Receiver devices placed on or near intersection traffic control devices recognize the signal and preempt the normal cycle of traffic lights. Once the emergency vehicle passes through the intersection and the receiving device no longer senses the remote triggering device, normal operation resumes [58].

Notice that whenever the preempted sequence is enable, all the requests of the pedestrians and the signal of the vehicle detectors are ignored.

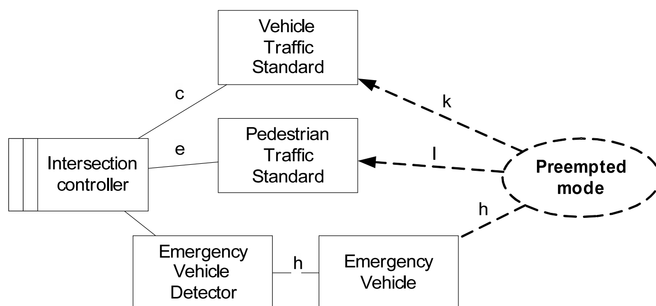

Figure 8.41: The Problem diagram for the preempted operating mode problem

The communication between the emergency vehicle and the receiver is one-way, i.e., the emergency vehicle continuosly re-transmit its signal while the receiver communicates the presence of the vehicle to the intersection controller. No acknowledgment message is sent back to the emergency vehicle. Moreover, the receiver is able to distinguish whether or not there are emergency vehicles on its approach, but it cannot understand how many vehicles are arriving.

The problem described here is related to the first generation of preempted traffic systems, which although it appears as quite primitive, are still widely used. Nowadays, state of the art solutions exploits the usage of GPS devices, wireless transponders, and complex full duplex communication protocols.

The problem is described by means of the Problem diagram shown in Figure 8.41. The problem is composed of 1) the biddable domain *Emergency Vehicle*, representing an emergency vehicle equipped with an emitter; 2) the causal domain *Emergency Vehicle Detector*, which represents the receivers of each approach; 3) the machine domain *Intersection Controller*, which represents the controller that once received the request of the emergency vehicle, enables the preempted sequence by issuing ad-hoc signals to the traffic lights; finally, 4) the causal domains *Vehicle Traffic Standard* and *Pedestrian Traffic Standard*, which represents the

traffic lights of the intersection.

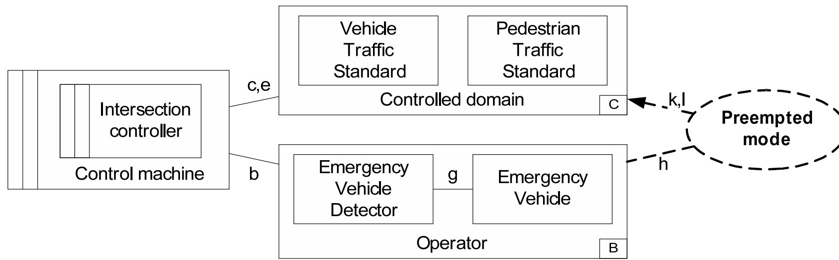

Figure 8.42: The Preempted operating mode problem fits the Controlled behavior frame

As shown in Figure 8.42, the problem is an instance of the Commanded behavior frame [37]: *Intersection controller* plays the role of *Control Machine*, while *Emergency Vehicle Detector* and *Emergency Vehicle* play the role of *Operator* and *Vehicle Traffic Standard* and *Pedestrian Traffic Standard* play the role of *Controlled Domain*.

The phenomena shared by *Control machine* and *Operator* are composed of the signals that *Emergency Vehicle Detector* issues to *Intersection Controller* whenever it receives signals from an emergency vehicle. The phenomena shared by *Control machine* and *Controlled domain* are composed of the set of commands that *Intersection Controller* sends to the traffic lights.

The requirements of the problem are illustrated by means of the `stm` diagram shown in Figure 8.43. The diagram is composed of two parallel `stm`s: the first one keeps track of the presence of emergency vehicles on the approaches, the second one describes the phases of the preempted sequence. In order to keep track of possibly emergency conditions in both the `stm`, the attributes *EVNS* and *EVEW* are introduced, which represent whether emergency vehicles are on the approach NS and EW, respectively.

At starting time, the first `stm` enters the state *No Emergency*. In this state no emergency vehicle has not been detected yet. The state can be left under the condition that an emergency vehicle is detected on the approach NS or EW. The detection is performed by the detectors of the approaches, which, according to the description provided in the `stm` of Figure 8.7, convert the continuously repeated signals of the emergency vehicles in a explicit notification to the controller. The passage to the state *EmergencyNS* and *EmergencyEW* is triggered by a signal of type *EVehicleRequest* characterized by the attribute *arriving* set to *true*. Such transitions differ for the allocation of the signals. In the first case the signal is allocated to the output port of the detector of the approach NS while in the second case to the one of the approach EW.

At firing time of such transitions, the event *startEmergency* is generated in order to notify to the second `stm` that the preempted sequence is enabled.

The `stm` may evolve by entering the state *EmergencyEW/NS* or returning to the state *NoEmergency*.

NoEmergency is reached is case the same detector that triggered the transition

to the current state notifies that the no emergency vehicle is on the approach. In this case the older operating mode is restored (by invoking again the operation *changeMode*).

EmergencyEW/NS is reached in case the detector of the approach opposite to the one that caused the entrance in the current state (*etNS* for the state *EmergencyEW*, and *etEW* for the state *EmergencyEW*) notifies that an emergency vehicle is arriving. The state *EmergencyEW/NS* represents an emergency condition in which emergency vehicles are on both approaches. The state can be exited by means of transitions to the previously described states *EmergencyEW* and *EmergencyNS*, which may fire in case the detectors notify there is no vehicle in their approaches.

The second `stm` describes the phases of the preempted sequence. At starting time the `stm` enters the state *No Emergency*, which represents that the intersection controller is operating a mode different from the preempted one, and no emergency vehicle has been detected yet. In case an emergency vehicle is detected the event *startEmergency*, generated by the first `stm`, triggers the output transitions from the current state. Notice that all the other states can be reached by means of a direct transition. The choice of which transition can fire strictly depends on further conditions such as the approach in which the emergency vehicle was detected, and the current phase of the intersection. As an example suppose that the current phase is the one represented by the state *Stop-Go Walk-DNWalk*. Furthermore, suppose that an emergency vehicle is detected on the approach EW. In this situation, the first `stm` notifies the detection by generating the event *startEmergency*, and enters the state *EmergencyEW*. This implies that *EVEW* is set to *true*, and, since no vehicle has not been detected yet on the approach NS, *EVNS* is set to *false*. As a consequence the `stm` enters the state *Stop-Go Walk-DNWalk*. The permanency in this state lasts until no emergency vehicle is on the approach NS (notice that nothing is said about the presence on the approach EW). As a consequence the `stm` enters the state *Stop-Wait Wait-DNWalk*. Suppose that no other emergency vehicle is detected on the approach NS. Hence, after *appNS.waitD* the `stm` passes to the state *Stop-Stop DNWalk-DNWalk*, and after *appNS.safetyD* of permanency, it reaches the state *Go-Stop DNWalk-Walk*, i.e., the state required by the emergency vehicle. The `stm` remains in this state since no emergency vehicle is detected on the approach EW. If no vehicle is detected, the first `stm` returns to the state *No Emergency*, and generates the event *stopEmergency*, which in turn triggers the transition to the state *No emergency* of the current `stm`.

Notice that the `stm` does not report the state *Flashing*. In case the intersection is in this state when an emergency vehicle is detected, depending on the approach where the vehicle is located, the `stm` directly passes to phase required by the emergency vehicle (i.e., one of the states *Stop-Go Walk-DNWalk* or *Go-Stop DNWalk-Walk*).

In case vehicles are detected on both approaches the requests are processed adopting a temporal order criterion. More specifically, in case two requests come from distinct approaches, the first request is served earlier.

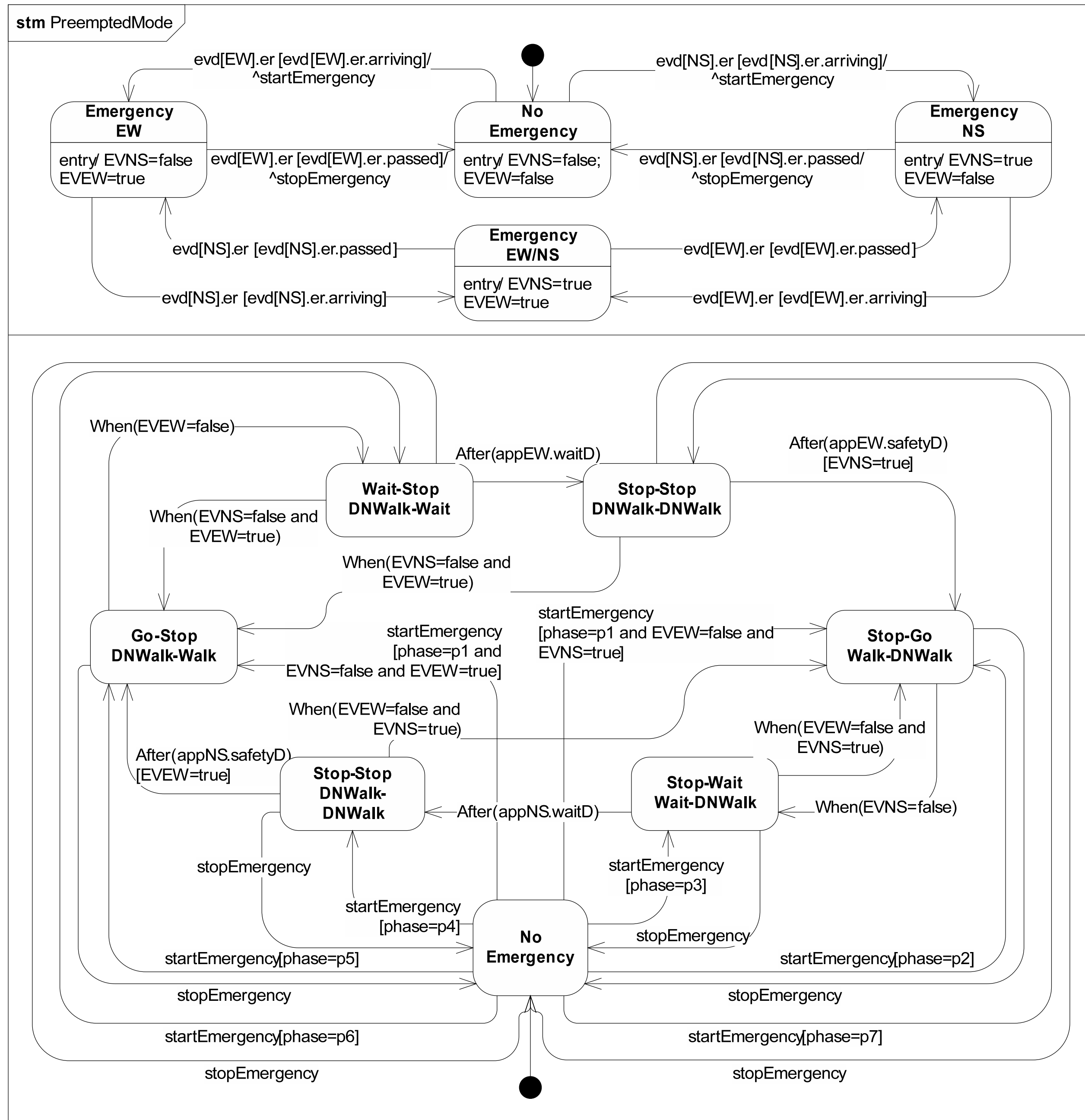

Figure 8.43: The stm diagram that describes the requirements of the problem Preempted operating mode

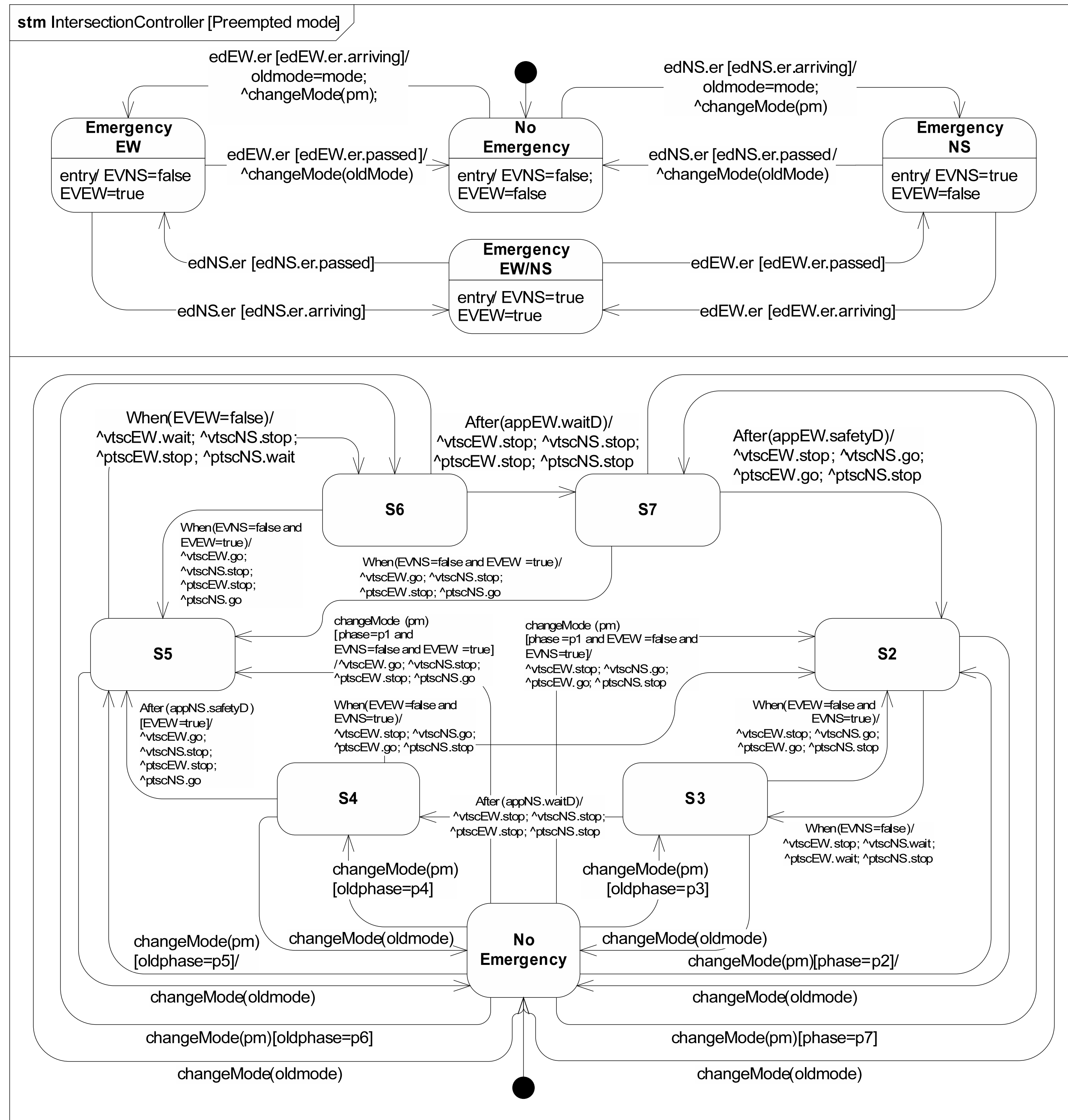

Figure 8.44: The stm diagram that describes the machine specification of the problem Preempted operating mode

If multiple emergency vehicles are moving at the same time on both the approaches, the green phase cannot be assured for all the vehicles. However, consider that emergency vehicles are equipped with siren and flashing lights, hence, they are allowed to cross the intersection also in case of red signal.

The machine specification for this problem is described by means of the `stm` diagram shown in Figure 8.44. Notice that such diagram is composed of two state machines similar to the ones used to describe the requirements. The diagrams differ for the allocations of the signals and for the actions performed at firing time of the transitions. As for the requirements case, the first `stm` keeps track of the emergency state, while the second one describes the phases and the operations performed during the emergency. In case a vehicle is detected, the current operating mode is stored in the variable *oldmode* and the operation *changeMode* is invoked. Such operation stores all the parameters of the current mode and enables the preempted sequence. The invocation event triggers the transitions of the second `stm`, which in turn evolves according to the same criteria introduced for the requirements. In case no vehicle is on either approach, the preempted sequence is exited by invoking the operation *changeMode*, which restores the older operating mode along with all its parameters.

8.4.7 Traffic lights states check

This section illustrates the Traffic lights check states problem, a problem transversal to all operating modes, which is related to the possibility of having one or more traffic lights in a state that is different from the expected one. As an example, some problems may occur in case a lamp is burn out or some of the internal components of a traffic light does not work correctly.

The problem is described by the problem frame diagram shown in Figure 8.45.

It is characterized by the machine domain *Intersection Controller*, by the causal problem domains *Vehicle Traffic Standard* and *Pedestrian Traffic Standard.* The involved domains share two different types of phenomena that represent the current state of the traffic lights and the commands issued by the controller, respectively.

The internal controller of the traffic lights performs some checks on the state of the lamps after issuing a command. Therefore, in case of some malfunctions, the local controllers inform the intersection controller of the error by specifying its description (the error type).

The requirements of this subproblem states that in case a severe malfunctioning may compromise the safety of the system all the traffic lights must be set to the state yellow flashing, while in case minor errors are detected, a description of the encountered problem must be logged.

As shown in Figure 8.46, the problem is an instance of the Required behavior frame [37]: *Intersection controller* plays the role of *Control Machine*, while *Vehicle Traffic Standard* and *Pedestrian Traffic Standard* play the role of *Controlled Domain.*

The phenomena shared by *Control Machine* and *Controlled Domain* directly correspond to the set of phenomena shared by *Intersection Controller* and *Vehicle Traffic Standard* and by *Intersection Controller* and *Pedestrian Traffic Standard.*

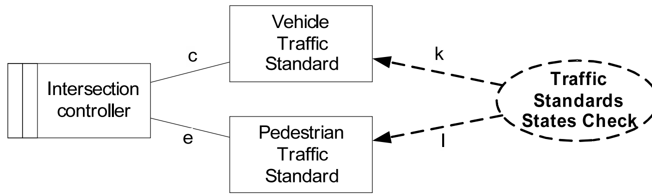

Figure 8.45: The problem frame diagram that describes the Traffic lights states check problem

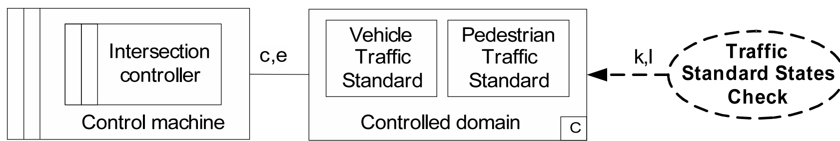

Figure 8.46: The Traffic lights states check problem fits the Required behavior frame

The requirements are illustrated by the `stm` diagram of Figure 8.47.

The `stm` diagram uses the same states that were considered for the definition of the requirements of all the previously presented operating modes. Each state is reached when a specific group of signals are received by the traffic lights of either approach. For each state, at entry time the combination of commands issued to the different traffic lights is stored and the actions *CheckVtsStates* and *CheckPtsStates* are invoked.

Figure 8.48 shows the `act` diagram that describes the action *CheckVtsStates*. The activity is characterized by four input parameter nodes named *vtss1*, *vtss2*, *vtss3* and *vtss4* of type *TrafficStandardState* and an input object node named *lastCmd* of type *TrafficStdCmdType*. Notice that *vtss1*, *vtss2*, *vtss3* and *vtss4* are allocated to the output port *tss* of the vehicular traffic standards *vts[N]*, *vts[S]*, *vts[E]*, *vts[W]*, while *lastCmd* is allocated to the attribute *lCmdV* used in the `stm` diagram of Figure 8.47.

At invocation time, after waiting for a certain reaction time, i.e., the minimum time required for the system to update the state of the traffic lights, the action *CheckState* is invoked. The effects of the action are partly described by the postcondition of Figure 8.48. Such a condition, depending on the errors revealed by the local controller of the traffic lights, specifies a different output value represented by the attribute *alarm*, an enumerative that indicates the severity of the encountered error. In case no error is detected, the execution of the activity is stopped. In case a minor error occurs, the activity logs the encountered problem by invoking the action *Log*, while in case a safety critical error is found a signal *Alarm* is generated and a description of the error is logged.

Notice that the action *CheckPtsStates* is equivalent to *CheckVtsStates*, with the exception of the input parameters that are allocated to the states of the

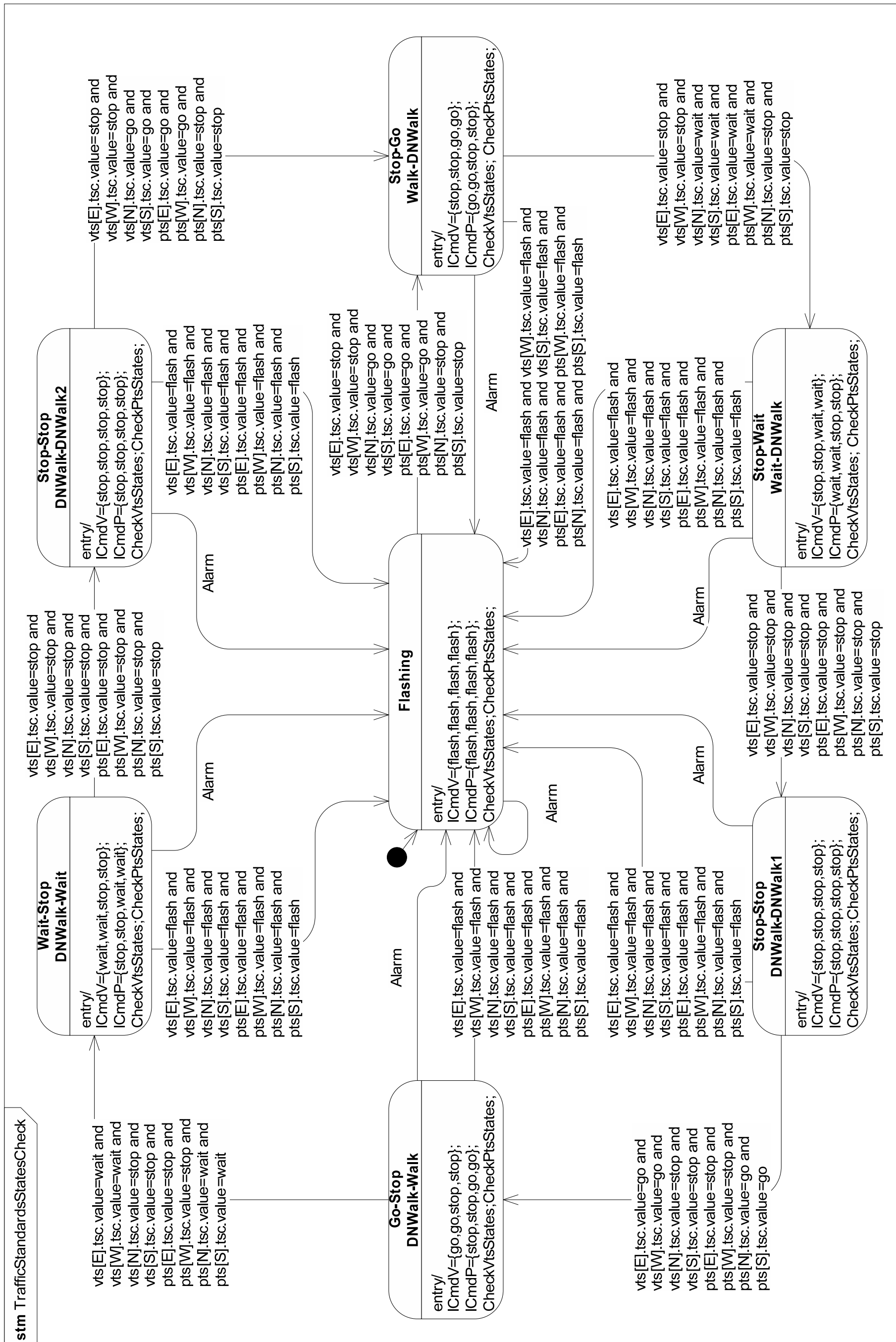

Figure 8.47: The stm diagram that depicts the requirements of the Traffic lights states check problem

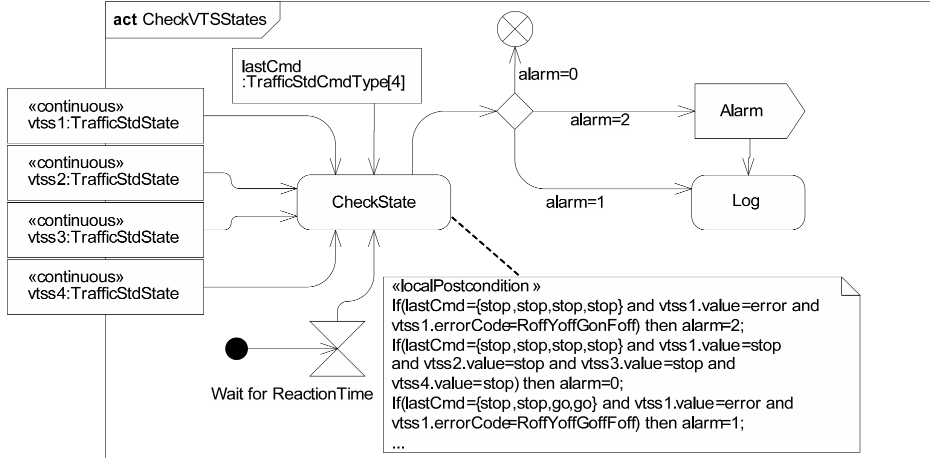

Figure 8.48: The `act` diagram that describes the action *CheckVtsStates*

pedestrian traffic standards, and to the attribute *lCmdP* of the `stm` of Figure 8.47 to store the last command issued to the pedestrian traffic lights.

The signal *Alarm* triggers some of the transitions of the `stm` diagram of Figure 8.47. More specifically, such signal causes the transitions from any phase of the system to the *Flashing* one, where all the traffic lights are in the state *Flashing*.

The machine specification is expressed by means of the `stm` diagram of Figure 8.49.

The `stm` is similar to the one used to express the requirements. The signals that trigger the transition of the states machine are allocated to the output ports of the Block *IntersectionController*. Such signals are generated by the different state machines that represent the machine specification for each supported operating mode. Similarly, to the requirements case, whenever a state is entered, the activities *CheckVtsStates* and *CheckPtsStates* are invoked. Such activities are equivalent to those used to express the requirements, with the exception of the allocation of their parameters. A further difference regards the effects of the transitions that are triggered by the signal *Alarm*. In this case, the commands that cause the flashing of the traffic lights are issued by means of the output ports *vtscEW*, *vtscNS*, *ptscEW* and *ptscNS*. In addition the operation *Changemode* is invoked in order to set the manual operating mode. In fact, in case a safety critical error occurs, after setting the state of the traffic lights to flashing yellow, no action can be automatically decided by the controller. A human intervention is required on some of the components of the system.

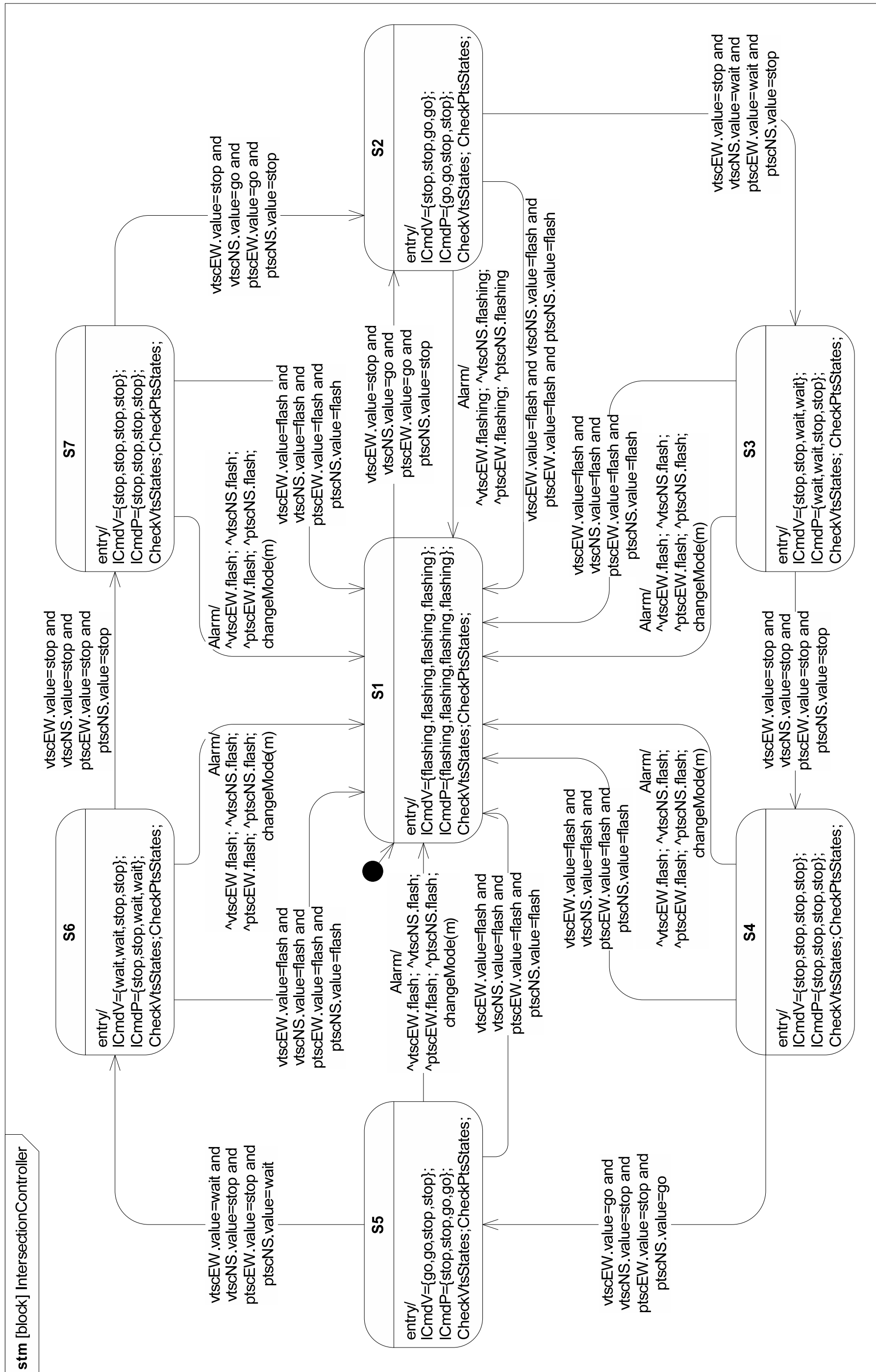

Figure 8.49: The `stm` diagram that described the machine specification for the Traffic lights states check problem

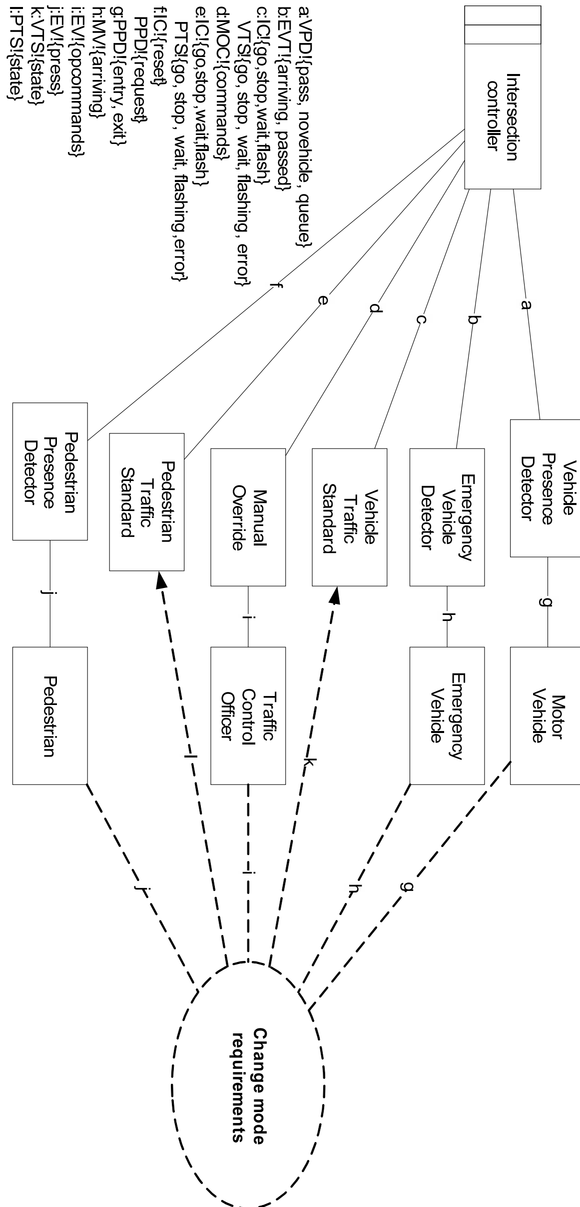

Figure 8.50: The Problem diagram for the Change mode problem

8.4.8 Change mode

The intersection controller has to support multiple operating modes. All the previously introduced problems describe single operating modes of the controller. In order to specify the general behavior of the intersection controller, it is necessary to compose the descriptions of such sub problems.

The need of composing the descriptions comes from the necessity to change the operating mode. In fact, the operator described in the manual operating mode problem, besides the previously introduced operations can also issue commands to change the current operating mode of the intersection controller.

The overall problem can be described by means of the Problem diagram shown in Figure 8.50.

As shown in Figure 8.51, this problem is an instance of the Commanded behavior frame [37]: *Intersection controller* plays the role of *Control Machine*, while *Pedestrian Presence Detector*, *Pedestrian*, *Vehicle Presence Detector*, *Motor Vehicle*, *Manual Override*, *Traffic Control Officer*, *Emergency Vehicle Detector* and *Emergency Vehicle* represent the domain *Operator* and *Vehicle Traffic Standard* and *Pedestrian Traffic Standard* represent the domain *Controlled Domain*.

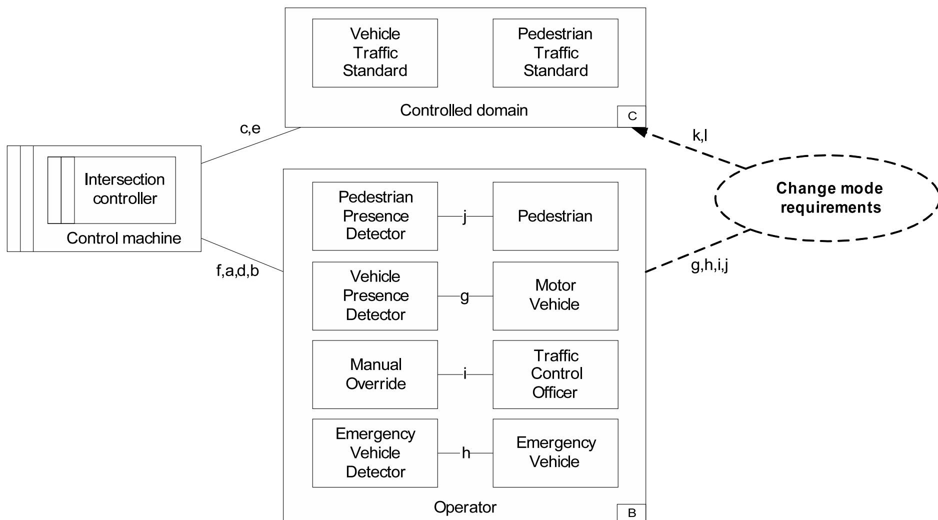

Figure 8.51: The Change mode problem fits the Controlled behavior frame

The phenomena shared by *Control machine* and *Operator* are composed of 1) the signals that *Vehicle Presence Detector*, *Pedestrian Presence Detector* and *Emergency Vehicle Detector* issue to *Intersection Controller* in order to notify the presence of a pedestrian, a vehicle or an emergency vehicle, respectively; 2) the commands that *Manual Override* sends to *Intersection Controller* as a response of what specified by *Traffic Control Officer* by means of the console. The phenomena shared by *Control machine* and *Controlled domain* are the commands that *Intersection Controller* issues to the vehicle and the pedestrian traffic lights.

The composed requirements of the problem are illustrated by means of the `stm` diagram shown in Figure 8.52.

The `stm` is defined by merging the `stm` of all sub-problems.
The composition is based on the following process.

- `stm` types identification.

  Two different types of `stm` are identified for the proposed sub problems. The first type of machine describes the phases of the intersection, while the second one keeps track of the current operating mode. The requirements of all the sub-problems are described by means of a single `stm` of the first type. The only sub problem whose requirements are described by means of both types of sub machines is the preempted operating mode problem.

  By applying this composition criterion, the overall `stm` of the general problem will be composed of two parallel `stms`.

- States definition.

  For each `stm` type, one state for each distinct state of the original sub-problems is defined.

  Since the first `stm` describes the phases of the intersection, and the phases are the same for all the operating modes, the resulting `stm` will be composed of the same states of the `stms` of all sub-problems.

  The second `stm` is exclusively introduced in the Preempted operating mode problem. All its states are directly reported.

- Composite states analysis.

  Since some of the states are composite, they contains parallel sub-machines, and thus it is necessary to define a different internal `stm` for each of the existing types of sub-machine used in each state of all the sub-problems. In other words, the same criteria described at the first and second steps of these guidelines are recursively applied to each composite state.

  In our case there are only two composite states among all sub-problems: the states *Stop-Go Walk-DNWalk* and *Go-Stop DNWalk-Walk*. Both of them are characterized by the same types of `stm`. A first type of `stm` keeps track of the request of the pedestrians, a second macro type considers the requests of the vehicle presence detectors. Such type, in turn, is further specialized in the description provided by the semi actuated and fully actuated operating mode problems. More specifically, in the fully actuated operating mode, a `stm` is defined for each semi approach in order to keep track of the current state of the whole road. Moreover, since this mode considers the request of the detectors of both approaches, the same `stm` is defined for both approaches (i.e., it is defined both for the state *Stop-Go Walk-DNWalk* and *Go-Stop DNWalk-Walk*). In semi actuated mode, the detectors are exclusively positioned along the secondary approach. As a consequence the sub-machines of the states *Stop-Go Walk-DNWalk* and *Go-Stop DNWalk-Walk* are different, and both of them describe the current state of the secondary approach EW.

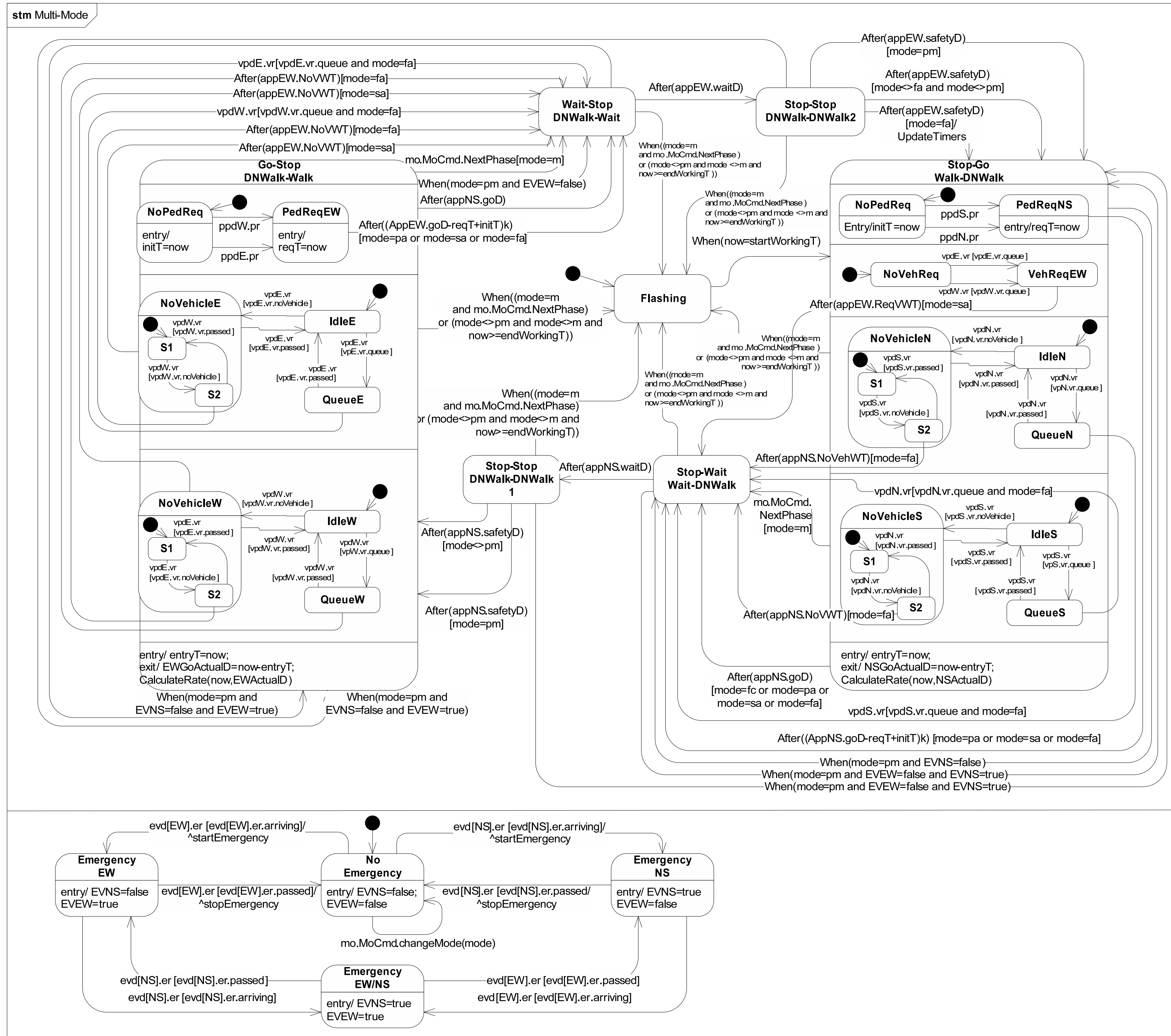

Figure 8.52: The stm diagram that depicts the requirements of the Mode change problem

As a result, four parallel `stm` are internally defined in the states *Stop-Go Walk-DNWalk* and *Go-Stop DNWalk-Walk* of the resulting general `stm`. Moreover, for each `stm`, all the states of the original machines are reported in the general one.

- Inclusion of the transitions.

  All the transitions defined in the machine of each previously introduced subproblem are reported in the final `stm`. The firing condition of the resulting transitions has to report the explicit reference to the operating modes for which the transition was originally defined. In case the same transition is defined for multiple operating modes, the firing condition of the resulting transition reports the list of the operating modes.

  This process is also recursively applied to the internal `stm` of the composite states.

- Inclusion of the action invocations.

  All the actions that are executed in case a transition fires or in case a state is entered or exited must be reported in the resulting general `stm`.

  In case the action in the original problem is executed at firing time of a transition, in the resulting `stm` its execution will be regulated by the firing conditions of the transition. As an example considers the action *Update-Timers* that is invoked at firing time of the transition between the states *Stop-Stop DNWalk-DNWalk* and *Stop-Go Walk-DNWalk* of the fully actuated operating mode problem. The action is reported in the composite `stm` by specifying a firing condition to the transition, which must be reported in order to assure that the action can be invoked only in case the operating mode is the fully actuated one.

  In case the action is executed at entering or exiting time of a state, in the resulting `stm` the action must be equipped with a pre condition that assures that it can be invoked only if the current operating mode is the one of the original problem. In case the action does not affect the evolution of the state machine (i.e., in case it does not generate events or modify attributes that may trigger a transition) or in case it is not necessary to constraint its execution, no additional pre condition is defined. Notice that all the actions that can be invoked at entry and exit time of the states of the resulting `stm` are not constrained.

- Merging of the states.

  Some of the reported `stm` may report the same description, hence, they can be merged. As an example consider the internal `stm` of the state *Go-Stop DNWalk-Walk*. The semi-actuated operating mode problem introduces a `stm` that describes the current state of the approach EW, while the fully actuated mode problem introduces a `stm` for each of the semi approaches E and W. By applying the composition criteria described at the previous steps, all these `stm` are defined in the state *Go-Stop DNWalk-Walk* of the resulting `stm`. Since the information reported by this `stm` is substantially the same,

the `stm` of the semi actuated mode problem is merged with the one of the fully actuated one. More specifically, the `stm` of the fully actuated mode reports all the states and transitions of the semi actuated problem. Only two transitions are not included in the `stm` of the fully actuated problem, i.e., the output transitions of the state *NoVehicleE* and *NoVehicleW*. As a consequence, the merging consists in defining a new output transition (the one defined in the semi actuated problem) for the states *NoVehicleE* and *NoVehicleW*.

A further requirement related to the current problem concerns the mode change. The operator can issue commands to change the current operating mode by means of the console. The required effects of the command sending are described in the second `stm` of Figure 8.52 (i.e., the `stm` that keeps track of the operating modes). The `stm` reports a transition that exits and enters the state *No Emergency*, which is triggered by the invocation of the operation *changeMode*. As specified by the `stm`, the operator can change the current operating mode only in case no emergency is under processing.

The machine specification can be described by means of a `stm` diagram similar to the one used to illustrate the requirements. As for the other subproblems, also in this case the `stm`s differ for the allocation of the signals and for the action that can be invoked at firing time of the transitions.

Chapter 9

Conclusions

The general goal of this PhD work concerned the definition of methodological guidelines to the usage of SysML for modeling real-time systems.

The Systems Modeling Language (SysML) [53] is a new general purpose language expressly defined to satisfy the requirements imposed by the UML for System Engineering Request for Proposal [64] that aims at extending UML with the notational elements needed to support systems engineering modeling purposes.

SysML does not provide any instrument to support the modeling of real-time systems, furthermore it does not provide any modeling process (the language is intended to be methodology independent [54]). As a consequence, methodological guidelines are required to support specific modeling aims.

This goal was initially addressed in Chapter 3, where on one hand we evaluated the capabilities of SysML as a notation for modeling systems characterized by real-time and safety critical requirements; on the other hand, we wanted to experiment with the application of the well known and sound concepts from the reference model for requirements and specification [26] to SysML based modeling. Namely, we modeled the Generalized Railroad Crossing system [29], which is a reliable and well-known benchmark for this kind of activities.

We found SysML well suited to support a model-centric requirement specification, since it provides dedicated constructs to describe all the aspects of a system: it supports the definition of both structural and behavioral features, and provides constructs to describe and organize the requirements.

The language also provides cross cutting constructs that allow the modeler to allocate behavioral elements on structural ones and vice-versa, and to relate the elements of the model to the requirements. Despite these remarkable features, the language is not fully satisfactory for modeling precise time requirements because it inherits some of the weaknesses of UML 2.

In particular, in order to express time related properties, we had to use an external language, although in the context of SysML Constraints: in fact, the possibility to employ “foreign” languages in the definition of constraints is compliant with the language specification. By means of a formal language like TRIO we were able to express time properties and behaviors in a rigorous way.

From a methodological point of view, it was easy to apply SysML along the

lines indicated by the reference model for requirements and specification, since SysML provides diagrams that are well suited to cover all types of descriptions, constraints and specifications that are necessary to obtain a complete, readable and coherent specification.

Although the experience introduced in Chapter 3 showed how easily the language can be applied to manage the requirements and specifications, the provided guidelines required to be systematized and extended in order to support the analysis of complex systems. This goal was addresses by proposing a requirements analysis technique based on SysML and Problem Frames.

Problem Frames [37], introduced in Chapter 4, are a sound requirement analysis approach built around the reference model for requirements and specifications that aims at driving the analyst from the phase of problem description, where the characteristics of the problem and its requirements are defined, to the specification of a machine that satisfies the requirements.

Although Problem Frames were proposed some years ago and a great interest for the theoretical aspects of the approach has been manifested by numerous requirement engineering research groups, the approach is scarcely adopted in industry.

Among the obstacles to the diffusion of Problem Frames there is the fact that the applicability of the approach to industrial cases has not been convincingly shown. In other words case studies of realistic complexity that evaluate the applicability of the approach are missing.

The work reported in Chapter 5 intended to partly address such issue by verifying the applicability of problem frames in the context of industrial software development. For this purpose, we employed Problem Frames to model the requirements of a system monitoring the transportation of dangerous goods.

The experience was generally successful: the usage of problem frames actually allowed us to achieve a fairly complete and clear comprehension of the requirements of the system.

We observed that requirements analysis based on Problem Frames needs further documentation and support in order to be successfully used in an industrial context, where clear and easy to use methodological guidelines are required. In particular, better documentation and examples should be made available, in order to support the analysts in recognizing problems. Moreover, a complete catalogue of available basic Problem Frames would be very helpful; otherwise the analyst may have difficulty in identifying the proper frame for a given problem (see the discussion about database querying in section 5.4).

In general, we believe that researchers should continue the development of the Problem Frames approach, in order to let it achieve the (not far) maturity level needed to increase its acceptance in industrial processes.

Another important weakness of Problem Frames concerns the lack of an adequate linguistic support. More specifically, Problem Frames are not equipped with a unique and clear way for expressing requirements; similarly, the approach does not provide any precise guidelines to express the behavioral aspects of problem domains, the specification of the machine and of the requirements. Therefore, the modeler has to choose a suitable notation to describe both the given domain behavior and the required behavior.

In order to address this issue, and also in order to provide SysML with effective methodological guidelines, in Chapter 6 we proposed the integration of the Problem Frames requirement analysis approach with the SysML notation. Notice that the goal was twofold: on one hand, we defined sound methodological guidelines to the usage of SysML for requirements analysis, on the other hand, we provided an effective notation to the analysis approach.

The experimental application of this integrated approach showed that Problem Frames can be effectively described by means of SysML diagrams and constructs, and that Problem Frames concepts can be effectively supported by SysML.

A relevant benefit of using SysML is represented by the description of the problem domains, that are mostly composed of non-software entities; note that this would hold even if we used a different requirement analysis methodology. SysML supports the definition and management of parametric constraints in a direct way, allowing the analyst to express properties and requirements by means of formal languages. Another important feature of SysML is the ability to deal with continuous behavior, which often characterizes physical problem domains. For instance, in the example presented in Chapter 6, the communication between the motor and the gate models a physical transmission, which maintains a continuous connection between the involved domains. SysML provides constructs to model phenomena –such as entities, signals, events, flows, etc.– in a concise and expressive way. This applies both to phenomena that are private to a domain and to shared phenomena.

In the literature several works like [43, 8] propose the usage of UML for supporting Problem Frames. Under all these respects, SysML showed to be more expressive and flexible than UML. Among the limitations of UML that are overcome by SysML there are the lack of support for modeling continuous behavior and the difficulty of including formal specifications in the models.

Jackson identifies some basic Problem Frames [37], i.e., shapes of a recurrent class of basic problems, and proposes concerns to solve them. In [37], a simple problem that fits the characteristics of each basic frame is illustrated.

In order to test the effectiveness of the combined approach, we experimented the modeling of the catalogue of basic Problem Frames proposed in [37].

The experience, illustrated in Chapter 7, reports the definition of the basic problem frames using SysML. Although relatively limited in scope, the experience showed that SysML completely supports the definition of Problem Frame concepts: we found no situation that could not be adequately expressed by means of SysML diagrams, or by means of constraints employing suitable notations. In particular, SysML provided a good level of abstraction for requirements analysis, and supported, either directly or via borrowed notations, the possibility to express real world features such as time-related issues or the continuous flow of data.

The basic examples presented in Chapter 7 underline how the analyst can benefit from the combination of Problem Frames and SysML. The analyst may choose several constructs and strategies to define properties associated with domains, to support the requirements or the machine specification. Moreover, he/she may adopt the constructs more suited to the analysis goals and to the adopted specification techniques.

Altough the Problem Frames catalogue represents a first step towards the

validation of the proposed approach, the intrinsic complexity of such problems is relatively low; hence, in order to validate the scalability of the approach we experimented the modeling of a case study of industrial complexity.

The adopted case study, illustrated in Chapter 8, concerns the modeling of a controller for a four way traffic intersection.

The case study allowed us to experiment the decomposition of the general problem into several sub-problems each of which matched a basic problem frame. The decomposition was driven by the operating modes that the intersection controller had to support: each mode represented a different problem to be addressed.

Once identified the sub-problems, the identification of the basic Problem Frames was relatively straightforward. Some of the sub-problems directly fitted without any adaptation the basic frames proposed by Jackson. Other problems, instead, simply required to abstract the descriptions by aggregating groups of domains that played a common role.

The specification of the machines that solved such sub-problems were defined by addressing the concerns proposed in [37] for the solution of the identified basic frames.

SysML well supported the specification of all the aspects of each sub-problem. It supported the representation of both the structural and behavioral aspects of the problem domains as well as the specification of the requirements and of the machines. Both requirements and machines were described by means of state machine diagrams and activity diagrams without the use of external notations.

Once all the sub-problems were identified and described, and once the machine specification was defined for each of these problems, we experimented the recomposition of the description of the general problem.

We applied a technique to recompose the description of requirements and machine specifications based on the merging of state machines. As a final result we came up with a state machine composed of different parts each of which was defined in one or more sub-problems.

As a final remark, this modeling experience showed the applicability of the technique to realistic complex problems, and we argue that this could be a first step towards the definition of a modeling approach that can be used in industry.

A further step towards the usage of the proposed combined approach in industry concerns the development of a tool supporting it. In order to achieve such goal, we are working on a meta-model that precisely specifies the constructs and rules needed for creating Problem Frames models. The proposed meta-model supports the construction of a tool and introduces both notational and methodological concepts of Problem Frames.

Such model is open and extensible: it allows one to integrate SysML elements, in order to support the description of the aspects of a problem that cannot be represented by means of the Problem Frames notation.

A tool that supports the metamodel is currently under development by exploiting Eclipse technologies. At present our prototype supports the definition of Problem Frames diagrams, and provides also advanced functionalities such as diagram partitioning, problem decomposition and domain decomposition.

At present the generated tool is dedicated to support the Problem Frames approach. We planned to fully support the combined Problem Frames - SysML

approach by using a transformation tool that drives the migration of a Problem Frames model to a SysML one. Such choice is motivated by the rich and complex notation of SysML; in other words we did not intend to build an `ad hoc` SysML editor, but we planned to define a software module that automatically generates a SysML model starting from a Problem Frames one compliant with the previously mentioned meta-model. All the aspects that can be represented by means of Problem Frames are automatically expressed in SysML, while the aspects that cannot be directly represented with the Problem Frames can be added to the generated model by using a dedicated SysML-based modeling IDE.

Bibliography

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[2] R. Alur and D.L. Dill. A theory of Timed Automata. *Theoretical Computer Science*, 1994.

[3] D. Bjorklund and J. Lilius. From UML behavioral descriptions to efficient synthesizable VHDL. In *Proceedings of the IEEE Norchip Conference*, 2002.

[4] S.J. Bleistein, K. Cox, and J. Verner. Problem Frames approach for e-business systems. In *Proceedings of the Int. Work. on Advances and Applications of Problem Frames (IWAAPF)*, 2004.

[5] C. Bock. SysML and UML2 support for activity modeling. *Systems Engineering*, 9(2), 2006.

[6] F. Bruschi, L. Baresi, E. Di Nitto, and D. Sciuto. SystemC code generation from UML models. In Eugenio Villar and Jean Mermet, editors, *System Specification and Design Languages*, pages 161–171. Kluwer CHDL Series, 2003.

[7] M.V. Cengarle and A. Knapp. Towards OCL/RT. In *Proceedings of the 11th Int. Symp. on Formal Methods Europe*, Berlin, Germany, 2002. Springer LNCS 2391.

[8] C. Choppy and G. Reggio. A UML-based approach for problem frame oriented software development. *Information and Software Technology*, 47(14):929–954, 2005.

[9] E. Ciapessoni, A. Coen-Porisini, E. Crivelli, D. Mandrioli, P. Mirandola, and A. Morzenti. From formal models to formally-based methods: an industrial experience. *ACM Transactions on Software Engineering and Methodology*, 8(1):79–113, January 1999.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

[11] P. Colombo, V. Del Bianco, and L. Lavazza. Towards the integration of SysML and Problem Frames. In *Proc. of the Int. Workshop on Advances and Applications of Problem Frames (IWAAPF)*, Leipzig (Germany), May 2008.

[12] P. Colombo, V. Del Bianco, and L. Lavazza. Using Problem Frames to model the requirements of a system for monitoring dangerous goods transportation. In *Proceedings of the Int. Workshop on Advances and Applications of Problem Frames (IWAAPF)*, Leipzig (Germany), May 2008.

[13] P. Colombo, V. Del Bianco, L. Lavazza, and A. Coen-Porisini. A methodological framework for SysML: a Problem Frames-based approach. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC)*, Nagoya (Japan), 2007.

[14] P. Colombo, V. del Bianco, L. Lavazza, and A. Coen-Porisini. An experience in modeling real-time systems with SysML. In *Proc. of Int. Work. on Modeling and Analysis of Real-Time and Embedded Systems (MARTES)*, 2006.

[15] P. Colombo, M. Pradella, M. Rossi, and G. Sassaroli. A UML 2-compatible language and tool for formal modeling real-time system architectures. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC 2006)*, Dijon, France, April 2006.

[16] K. Cox, J.G. Hall, and L. Rapanotti. A roadmap of problem frames research. *Information and Software Technology*, 47(14):891–902, 2005.

[17] V. del Bianco, L. Lavazza, and M. Mauri. A formalization of UML statecharts for real-time software modeling. In *Proceedings of the Sixth Biennial World Conference on Integrated Design Process Technology (IDPT 2002)*, Pasadena, California, USA, June 2002.

[18] V. del Bianco, L. Lavazza, and M. Mauri. Model checking UML specifications of real-time software. In *Proceedings of the Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002)*, Greenbelt, Maryland, USA, December 2002.

[19] V. del Bianco, L. Lavazza, M. Mauri, and G. Occorso. Towards UML-based formal specifications of component-based real-time software. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(2):179 – 192, March 2007.

[20] M. Elkoutbi, M. Bennani, R. K. Keller, and M. Boulmalef. Real-time system specifications based on UML Scenarios and Timed Petri Nets. In *Proceedings of the International Workshop on Communication Software Engineering (IWCSE2002), IEEE 2nd International Symposium on Signal Processing and Information Technology*, pages 362–366, Marrakech, Morocco, December 2002.

[21] S. Flake and W. Mueller. An OCL extension for real-time constraints. In *Advances in Object Modeling with the OCL*, pages 150–171. Springer Verlag, 2002.

[22] D. Garlan and M. Shaw. *Software Architecture*. Prentice-Hall, 1996.

[23] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO a logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2), May 1990.

[24] T. Groker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC.* Kluwer Academic Publishers, 2002.

[25] D. Gross and C. M. Harris. *Fundamentals of Queueing Theory.* Wiley, third edition, 1998.

[26] C.A. Gunter, E.L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *Software*, 17(3), May-June 2000.

[27] J. G. Hall, L. Rapanotti, and M. Jackson. Problem frame semantics for software development. *Software and Systems Modeling*, 4(2), 2005.

[28] M. Hause and F. Thom. Building embedded systems with UML 2.0/SysML. 2005. White Paper.

[29] C. L. Heitmeyer, R. D. Jeffords, and B.G. Labaw. Comparing different approaches for specifying and verifying real-time systems. In *Proceedings of the 10th IEEE Workshop on Real-Time Operating Systems and Software*, pages 122–129, New York, NY, USA, May 1993.

[30] C.L. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[31] INCOSE. *Systems Engineering Handbook*, 2004. INCOSE-TP-2003-016-02, Version 2a.

[32] M. Jackson. Problems and requirements. In *Proceedings of the 2nd IEEE nternational Symposium on Requirements Engineering (RE'95)*, York (England), 1995.

[33] M. Jackson. *Software Requirements and Specifications.* ACM Press Books, 1995.

[34] M. Jackson. Problem complexity. In *Proceedings of the 3rd Int. Conf.on Engineering of Complex Computer Systems (ICECSS'97)*, Como (Italy), 1997.

[35] M. Jackson. Problem Analysis Using Small Problem Frames. *South African Computer Journal*, 22:47–60, 1999.

[36] M. Jackson. Problem analysis and structure. In *In Engineering Theories of Software Construction, Proceedings of NATO Summer School.* IOS Press, 2000.

[37] M. Jackson. *Problem Frames - analysing and structuring software development problems.* Addison-Wesley ACM Press, 2001.

[38] M. Jackson. Problem Frames and Software Engineering. In *Proc. of Int. Work. on Advances and Applications of Problem Frames (IWAAPF)*, May 2004.

[39] A. Jha. Problem frames approach to web services requirements. In *Proc. of Int. Work. on Advances and Applications of Problem Frames (IWAAPF)*, 2006.

[40] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In *Proceedings of the 2nd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, 1992.

[41] R. Laney, L. Barroca, M. Jackson, and B. Nuseibeh. Composing requirements using Problem Frames. In *Proc. of Int. Conf. on Requirements Engineering (RE)*. IEEE CS Press, 2004.

[42] P. A. Laplante. *Real-Time Systems Design and Analysis*. Wiley Interscience, 2004.

[43] L. Lavazza and V. Del Bianco. Combining problem frames and UML in the description of software requirements. In *Proceedings of the Int. Conf. on Fundamental Approaches to Software Engineering (FASE)*, 2006.

[44] L. Lavazza, S. Morasca, and A. Morzenti. A dual language approach to the development of time-critical systems with UML. In *Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS) in conjunction with ETAPS 2004*, Electronic Notes in Theoretical Computer Science 116, pages 227–239, Barcelona, Spaign, March 2004. Elsevier.

[45] Z. Li, J. G. Hall, and L. Rapanotti. From requirements to specifications: a formal approach. In *Proceedings of the Int. Work. on Advances and Applications of Problem Frames (IWAAPF)*, 2006.

[46] G. Martin and W. Mller. *UML for SOC Design*. Springer, 2005.

[47] W.E. McUmber and B.H.C. Cheng. UML-based analysis of embedded systems using a mapping to VHDL. In *Proceedings of the IEEE International Symposium on High-Assurance Systems Engineering (HASE)*, pages 56–63, 1999.

[48] A. Moore. SysML hits the home straight. 2005. White Paper.

[49] K.D. Nguyen, Z. Sun, and P.S. Thiagarajan. Model-Driven SoC design via executable UML to SystemC. In *Proceedings of the IEEE International Real-time Systems Symposium (RTSS)*, 2004.

[50] OMG. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*, 2004. ptc/04-09-01.

[51] OMG. *UML Profile for Schedulability, Performance and Time, Ver. 1.1*, 2005. formal/05-01-02.

[52] OMG. *Object Constraint Language Specification, version 2.0*, 2006. formal/2006-05-01.

[53] OMG. *OMG Systems Modeling Language (OMG SysML) Specification*, May 2006. Final Adopted Specification, ptc/06-05-04.

[54] OMG. *OMG Systems Modeling Language (OMG SysML) Tutorial*, July 2006.

[55] OMG. *UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE)*, 2007. Specification, ptc/07-08-04.

[56] OMG. *Unified Modeling Language: Infrastructure, Ver. 2.1.2*, November 2007. formal/2007-11-02.

[57] OMG. *Unified Modeling Language: Superstructure, Ver. 2.1.2*, November 2007. formal/2007-11-02.

[58] M. J. Pacelli, C. J. Messer, and T. Urbanik II P.E. Development of an actuated traffic control process utilizing real-time estimated volume feedback. Technical report, Texas Transportation Institute - The Texas A& M University System, 2000.

[59] V. A. Pedroni. *Circuit Design with VHDL.* MIT Press, 2004.

[60] M. Pradella, M. Rossi, and D. Mandrioli. Architrio: a UML-compatible language for architectural description and its formal semantics. In *Proceedings of the 25th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2005)*, LNCS 3731, Taipei, Taiwan, October 2005.

[61] M. Pradella, M. Rossi, and D. Mandrioli. A UML-compatible formal language for system architecture description. In *Proceedings of the 12th International SDL Forum (SDL 2005)*, LNCS 3530. Springer, June 2005.

[62] F.J. Rammig. OCL goes real-time. In *Proceedings of the Fifth IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02).* IEEE, 2002.

[63] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A UML 2.0 profile for SystemC: toward high-level SoC design. In *Proceedings of the 5th ACM international conference on Embedded software*, Jersey City, New Jersey, USA, 2005.

[64] SE-DSIG (OMG Systems Engineering Domain Special Interest Group). *UML for systems engineering RFP*, March 2003. ad/03-03-41.

[65] R. Seater and D. Jackson. Problem Frame transformations: Deriving specifications from requirements. In *Proc. of Int. Work. on Advances and Applications of Problem Frames (IWAAPF)*, 2006.

[66] Y. Vanderperren, M. Pauwels, W. Dehaene, A. Berna, and F. Ozdemir. A SystemC based System on Chip modelling and design methodology. In W. Mueller, W. Rosenstiel, and J. Ruf, editors, *SystemC: Methodologies and Applications*. Kluwer Academic Publishers, 2003.

[67] Y. Vanderperren, G. Sonck, P. Van Oostende, M. Pauwels, W. Dehaene, and T.Moore. A design methodology for the development of a complex System-on-Chip using UML and executable system models. In *Proceedings of the Forum on Specification and Design Languages (FDL)*, 2002.

[68] W. Wolf. *Modern VLSI Design: System-on-Chip Design*. Prentice Hall, third edition, 2002.