



UNIVERSITY OF BERGAMO

SCHOOL OF DOCTORAL STUDIES

DOCTORAL DEGREE IN ENGINEERING AND APPLIED SCIENCES

XXIX CYCLE

SSD: ING-INF/05

**Rigorous Model-based Development of  
Programmable Electronic Medical Systems  
(PEMS): from Requirements to Code**

Doctoral Thesis

**Silvia BONFANTI**

Student ID 1004829

Supervisor:

**Chiar.mo Prof. Angelo Gargantini**

Academic year 2015/16



---

---

## Acknowledgements

---

I would like to thank my supervisor Prof. Angelo Gargantini, for his guidance, support and help during my PhD. I would like to thank Prof.ssa Elvinia Riccobene and Dr. Paolo Arcaini for their continuing and patient support.

A special thanks to my husband, my parents and my sister for their support and encouragement.

I would like to thank the SCCH (Software Competence Center Hagenberg) and all the staff, in particular Dr. Atif Mashkoor who has supported me in research activities during my stay at SCCH.

The Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria have supported this work in the frame of the COMET (Competence Centers for Excellent Technologies) project.



---

---

# Contents

---

<b>Acknowledgements</b>	<b>I</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Research questions and Objective . . . . .	3
<b>I State of the Art</b>	<b>7</b>
<b>2 Standards and regulations for the certification of medical devices software</b>	<b>9</b>
2.1 IEC 62304: Software Development Life Cycle . . . . .	11
2.2 FDA: General Principles of Software Validation . . . . .	13
2.3 Conclusion . . . . .	14
<b>3 Systematic Literature Review</b>	<b>17</b>
3.1 The SLR Process . . . . .	17
3.1.1 Papers selection . . . . .	18
3.1.2 Classification and Synthesis . . . . .	23
3.2 Conclusion . . . . .	31
<b>II Abstract State Machines</b>	<b>35</b>
<b>4 ASM-based development process</b>	<b>37</b>
4.1 ASMs Modelling, Validation & Verification . . . . .	38
4.2 Compliance of the ASMs process with regulations . . . . .	41
4.2.1 Compliance of Abstract State Machines process with IEC 62304	42
4.2.2 Compliance of Abstract State Machines process with FDA	
principles . . . . .	43
<b>5 Unified Syntax for ASM to Xtext</b>	<b>45</b>
5.1 UASM Header . . . . .	46
5.2 Transition Rules Definition . . . . .	47

## Contents

---

5.3	Type Definition . . . . .	55
5.4	Functions definition . . . . .	57
5.5	Terms . . . . .	59
5.6	Header . . . . .	68
5.7	Terminals . . . . .	68
5.8	Implementation: from UASM to <i>Asmeta</i> . . . . .	69
5.8.1	Coffee Vending Machine . . . . .	69
<b>6</b>	<b>Visualization for Abstract State Machines</b>	<b>73</b>
6.1	A visual notation for Abstract State Machines . . . . .	74
6.2	Visual Trees . . . . .	75
6.3	Visual Patterns . . . . .	77
6.3.1	Structural patterns . . . . .	77
6.3.2	Semantic Patterns . . . . .	78
6.4	Tool . . . . .	82
6.5	Preliminary evaluation of visual notation . . . . .	82
<b>7</b>	<b>Automatic Code Generator</b>	<b>85</b>
7.1	Model transformation . . . . .	85
7.1.1	Model-to-Text . . . . .	86
7.1.2	Model-to-Model . . . . .	86
7.2	Microcontrollers . . . . .	87
7.3	Design choices . . . . .	88
7.4	Transformation process . . . . .	89
7.5	Tool implementation . . . . .	92
7.5.1	Code Generation . . . . .	92
7.5.2	Hardware configuration . . . . .	98
7.5.3	The <i>Asm2C++</i> Eclipse Plugin . . . . .	99
7.6	Illustrative example . . . . .	100
7.6.1	Modelling . . . . .	101
7.6.2	Validation & Verification . . . . .	101
7.6.3	Hardware . . . . .	103
7.6.4	From UASM to C++ code . . . . .	104
<b>III</b>	<b>Case studies</b>	<b>109</b>
<b>8</b>	<b>3D4AMB: diagnosis and treatment for visual diseases</b>	<b>111</b>
8.1	Stereoacuity test . . . . .	111
8.2	3D4AMB projects . . . . .	113
8.3	StereoAcuity Test . . . . .	115
8.3.1	Modelling and refinement . . . . .	116
8.3.2	Validation & Verification of the SAM . . . . .	118
8.3.3	Scenario and test generation . . . . .	121

<b>9 Hemodialysis machine case study</b>	<b>125</b>
9.1 Requirements . . . . .	125
9.1.1 Hemodialysis machine architecture . . . . .	126
9.1.2 Hemodialysis therapy . . . . .	128
9.2 Modelling by refinement . . . . .	129
9.2.1 Ground model . . . . .	129
9.2.2 First refinement: preparation phase . . . . .	132
9.2.3 Second refinement: initiation phase . . . . .	137
9.2.4 Third refinement: ending phase . . . . .	143
9.3 Conformance checking . . . . .	146
9.4 Related Work . . . . .	149
9.5 Conclusions . . . . .	149
<b>Conclusion</b>	<b>151</b>
<b>Appendix A IEC and ISO deliverables</b>	<b>155</b>
<b>Appendix B Publications</b>	<b>157</b>
<b>Bibliography</b>	<b>167</b>





---

# CHAPTER 1

---

## Introduction

---

Many people get in contact with medical devices during their life, e.g. infusion pump, diagnostic X-ray equipment, hemodialysis machine. These devices were based on mechanical and electrical components, and all the hardware parts were tested to guarantee their correct functioning. Over the years, the software has increasingly become a critical component of the devices and this has given rise to a new subgroup of medical devices: Programmable Electronic Medical Systems (PEMS). PEMS are safety-critical system [78]. They are a combination of hardware and software to implement all the functionalities that have effects on people health and, in case of malfunctions, they can seriously compromise human safety. For this reason, the software installed on these devices must be guaranteed through rigorous processes to assure medical software safety and reliability. Correct operation of a medical device depends upon the software, whose development should adhere to certification standards or guidelines.

Several standards for the validation of medical devices have been proposed, but they mainly consider hardware aspects of the physical components of a device, and do not mention the software component. The only reference concerning regulation of medical software is the standard IEC (International Electrotechnical Commission) 62304 [3]. This standard provides a very general description of common life cycle activities of the software development, without giving any indication regarding process models, or methods and techniques to assure safety and reliability. The U.S. Food and Drug Administration (FDA), the United States federal executive department that is responsible for protecting and promoting public health through the regulation and supervision of medical devices, although accepts the IEC 62304 standard, also pushes towards the application of rigorous approaches for software validation. In [128], FDA defines several broad concepts

that can be used as guidance for software validation and verification, and requires these activities to be conducted throughout the software development life cycle. However, no particular technique or method is recommended.

The rigorous process presented in this thesis is based on the Abstract State Machines (ASMs) [41] formal method, a mathematically based techniques for the specification, analysis and development of software systems. Currently, the industries do not develop medical devices using formal methods, but in university research several groups apply experimentally formal methods to medical device software. Systematic Literature Review (SLR) is a good start point to better understand the current state of the art and get some tips to follow for both academic and industrial researcher. The formal approach based on Abstract State Machines (ASMs) proposes an incremental life cycle model for software development based on model refinement. It covers the main software engineering activities (specification, validation, verification, conformance checking), and it is supported by a wide range of tools [25].

Although, we believe that rigour of a formal method can improve the current normative and that ASMs have the required potential, evidence must be given about a smooth possible integration of the method into the standards for medical software development. Additionally, it must be studied how much the ASM process is compliant with the normative: which steps and activities of the standard IEC are covered by using ASMs, and which are not; which FDA principles are ensured, and to what extent.

In this thesis, the ASM development approach and its supporting *Asmeta* (ASM mETAmodeling) framework<sup>1</sup> are used to propose a rigorous development process for PEMS. The final goal is to provide a process able to guarantee the development of correct and controllable systems in a correct and controllable way. Up to now, ASMs and *Asmeta* framework has never been applied to model and to analyse medical devices. The application of the ASMs to the medical software, and the definition of a process for their correct development has leaded to some improvements of the method, mainly regarding the textual and graphical notations, and the automatic code generation from models.

A new rigorous notation, Unified Syntax for Abstract State Machine (UASM), has been defined to provide a stable language kernel for ASMs languages that support the interoperability of tools designed and implemented by different groups. In addition, the UASM language should be usable for communication with customers and non-experts (e.g. doctors and technician).

The mathematical notation used in ASMs can appear sometimes difficult to read and to understand by not expert stakeholders, and the size of the specification often consists of several pages of rules and formulas. For this reason, the availability of a visualization tool to provide a graphical representation of ASM models has been considered a good means for people to communicate and to get a common understanding.

Another limitation discovered during the application of ASMs to medical device, is the manual implementation of the code starting from the models. By performing this last step manually, the costs increase, the reuse of a formal spec-

---

<sup>1</sup><http://asmeta.sourceforge.net/>

ification is limited, some faults can be introduced in the code writing process, and it can be a barrier for a wider adoption of the ASMs. For this reason, we have devised a methodology to generate the desired source code from ASMs. The tool automatically translates the formal specification into the target code (C++ for Arduino in the present case) and it preserves the system behaviour and the properties verified during validation and verification.

### 1.1 Research questions and Objective

---

The goal of this thesis is to apply rigorous methods to derive medical software starting from requirements.

The thesis is divided into nine chapters, and each of these chapters (introduction excluded) try to answer to a research question (RQ).

#### **RQ1 Are there regulations for medical software?**

This is shown in Chapter 2 in which two documents are analysed, IEC 62304 and FDA general principles. The first is an international standard. It describes the life cycle activities of the software development without giving any information about the method to be applied. The FDA general principles provide the guidelines for the validation and verification of the medical software.

#### **RQ2 Are there application of formal methods to medical device software development?**

The answer is in Chapter 3. SLR is applied to point out which is the current state of the art. Statistical data and technical details are provided to understand the current applications of the rigorous methods to medical devices software.

#### **RQ3 ASMs are applicable to support the development of software for medical devices?**

Chapter 4 shows the *Asmeta* framework. The set of tools included in the framework support the user during the development and the analysis of formal specifications using ASMs.

#### **RQ4 How to increase designer productivity by improving ASM tools interoperability?**

There are different communities which have developed their tools based on their language to represent ASMs. Due to this, the diffusion in industrial field could be difficult because the final user has to choose between the frameworks and each of them provide different tools for different goals. This problem is taken on in Chapter 5. UASM is implemented using a framework for the development of programming languages and domain-specific languages. A translator from UASM to *AsmetaL* has been developed, by taking advantages from framework functionalities. Using the translator the UASM models can be translated in *AsmetaL* models and the *Asmeta* tools can be used to perform different analysis.

#### **RQ5 How to improve the readability and understandability of ASM specifications?**

When the system is complex and the textual model consists of several lines of code, for the user it would be difficult to understand the behaviour of the model. A tool for the visualization of the ASMs automatically translates the textual model

in the corresponding graphical notation. The graphical representation helps the user to easily understand the ASM specification.

### **RQ6 Is it possible to automatically translate a formal specification to code automatically?**

In Chapter 7, a tool for code generation is shown. The tool translates automatically UASM models into C++ code for Arduino platform. The translation tries to keep true the system behaviour and the properties verified during validation and verification.

### **RQ7 Are the tools applicable to real case studies?**

In Part III, the *Asmeta* tools are applied to two case studies. The first is a software to detect visual diseases developed within the 3D4AMB<sup>2</sup> (3D for the diagnosis and treatment of AMBlyopia) project (see Chapter 8). The second case study is the hemodialysis machine case study (see Chapter 9).

The material presented in this thesis have been published in the proceedings of the following international workshops and conferences:

- the content of Chapter 3 has been partially presented in “S. Bonfanti, A. Gargantini, and A. Mashkoor. A preliminary systematic literature review of the use of formal methods in medical software systems. In *23rd EuroAsiaSPI Conference, Graz University of Technology, Graz, Austria, 2016*”
- the content of Chapter 5 has been partially presented in “P. Arcaini, S.a Bonfanti, M. Dausend, A.o Gargantini, A. Mashkoor, A. Raschke, E. Riccobene, P. Scandurra, and M. Stegmaier. Unified syntax for abstract state machines. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016*, LNCS, pages 231–236. Springer International Publishing, 2016”
- the content of Chapter 6 has been presented in “P. Arcaini, S. Bonfanti, A. Gargantini, and E. Riccobene. Visual notation and patterns for abstract state machines. In Paolo Milazzo, Dániel Varró, and Manuel Wimmer, editors, *Software Technologies: Applications and Foundations: STAF 2016 Collocated Workshops: DataMod, GCM, HOFM, MELO, SEMS, VeryComp, Vienna Austria, July 4-8, 2016*, LNCS, pages 163–178. Springer International Publishing, 2016”
- the content of Chapter 7 has been partially presented in “S. Bonfanti, M. Caris-soni, A. Gargantini, and A. Mashkoor. *Asm2C++: a tool for Code Generation from Abstract State Machines to Arduino*. In *NASA Formal Methods Symposium, 2017*”
- the content of Chapter 8 has been presented in
  - “P. Arcaini, S. Bonfanti, A. Gargantini, A. Mashkoor, and E. Riccobene. Formal validation and verification of a medical software critical component. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 80–89. Institute of Electrical and Electronics Engineers Inc., 2015”

---

<sup>2</sup><http://3d4amb.unibg.it/>

- “Silvia Bonfanti, Angelo Gargantini, and Andrea Vitali. A mobile application for the stereoacuity test. In G. Vincent Duffy, editor, *Digital Human Modeling. Applications in Health, Safety, Ergonomics and Risk Management: Ergonomics and Health: 6th International Conference, DHM 2015, Held as Part of HCI International 2015, Los Angeles, CA, USA, August 2-7, 2015, Proceedings, Part II*, LNCS, pages 315–326. Springer International Publishing, 2015”
- “Angelo Gargantini, Fabio Terzi, Matteo Zambelli, and Silvia Bonfanti. A low-cost virtual reality game for amblyopia rehabilitation. In *Proceedings of the 3rd 2015 Workshop on ICTs for improving Patients Rehabilitation Research Techniques*, pages 81–84. ACM, 2015”
- the content of Chapter 9 has been presented in
  - “P. Arcaini, S. Bonfanti, A. Gargantini, and E. Riccobene. How to assure correctness and safety of medical software: The hemodialysis machine case study. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - ABZ 2016*, volume 9675, pages 344–359. Springer International Publishing, 2016”
  - “P. Arcaini, S. Bonfanti, A. Gargantini, A. Mashkoor, and E. Riccobene. Integrating Formal Methods into Medical Software Development: the ASM approach (*submitted at Science of Computer Programming*)”

This thesis reports the results of an ongoing research activity carried out by a scientific group I was part of in the last two years. Some parts of the presented work have been developed by co-authors of the papers published, but they are reported in the thesis for completeness and to ensure that the reader is able to understand the content. My contribution can be divided into three categories:

- Development: I have developed tools/software.
- Experimental: I have tested the tools already available to verify the applicability to case studies.
- Conceptual: I have worked on the theoretical part of the topic.

Following, I am going to make precise my contribution for each chapter of the thesis.

**Chapter 2:** I have searched the standards, technical reports, guidelines and other documents for the certification of medical devices software. Subsequently, I have selected the most important standard/guidelines used by companies.

**Chapter 3:** I have collected the papers from different repositories, I have selected the important papers and I have analysed them to perform the analysis.

**Chapter 4:** This is a background chapter regarding the process adopted by *Asmeta* framework for the analysis of the models: from requirements to validation and verification of the models. This process has been used for the case studies in Chapter 8 and Chapter 9.

**Chapter 5:** I have implemented the UASM grammar using the Xtext framework. Some ambiguities discovered during the implementation have been resolved. Based on Xtext, I have developed a translator from UASM to **AsmetaL**. The translator offers the opportunity to the user to use the tools of the **Asmeta** framework to perform the analysis of the model.

**Chapter 6:** I have applied the control state ASMs to the case study presented in Chapter 9 and I have identified the limitation of the notation. I have cooperated in the definition of the notation presented in the thesis and I have applied the notation to the case studies.

**Chapter 7:** I have worked on the theoretical part of the code generator and I have partially contributed to the development of the code generator. Furthermore, I have worked on the case study definition and I have contributed in the process from requirements to code.

**Chapter 8:** I have applied the **Asmeta** process to the stereoacuity test. I have developed and validated the models and I have verified the properties defined in the requirements document.

**Chapter 9:** I have applied the **Asmeta** process to the hemodialysis case study. I have developed and validated the models and I have verified the properties defined in the requirements document.



**Part I**  
**State of the Art**





---

## CHAPTER 2

---

### Standards and regulations for the certification of medical devices software

---

Due to the criticality of medical devices, in the world, lots of institutions provide standards/guidelines for their certifications. For example, IEC (International Electrotechnical Commission), ISO (International Organization for Standardization), European Union, FDA (Food and Drug Administration), ANSI (American National Standards Institute), UNI (Italian Organization for Standardization) and DIN (German Institute for Standardization). The European Union (EU) puts into force council directives concerning medical devices. The council directive 93/42/EEC [50] is the first introduced by EU. The directive defines medical device as “any instrument, apparatus, appliance, material or other article, including the software necessary for its proper application intended by the manufacturer to be used for human beings” for the purpose of diagnosis, prevention, monitoring, treatment or alleviation of disease, compensation for an injury or handicap, investigation, replacement or modification of the anatomy or of a physiological process, control of conception. The directive classifies the medical devices in non-invasive devices, invasive devices (in whole or in part, penetrates inside the body) and active devices (depend on a source of electrical energy or power). The council directive 2007/47/EC [58] extends the definition of medical device as “any instrument, apparatus, appliance, software, material or other article, whether used alone or in combination, together with any accessories”. This directive defines “stand alone software” as an active medical device. European Commission – the executive body of the European Union – provides “MEDDEVs”<sup>1</sup>, a range of guidance documents, to assist stakeholders in implementing Directives related to Medical Devices. These documents do not give information about the process to develop

<sup>1</sup>[http://ec.europa.eu/growth/sectors/medical-devices/guidance/index\\_en.htm](http://ec.europa.eu/growth/sectors/medical-devices/guidance/index_en.htm)

medical devices software, but they only provide information to classify a software as medical device or not. However, the new amendments introduced by Directive 2007/47/EC have not been incorporated into all MEDDEVs yet. At present, the most important standards for medical devices certification were introduced by IEC and ISO. The standards are:

- IEC 61508-3:2010 [8] defines software requirements for functional safety of electrical/electronic/programmable electronic safety-related systems
- ISO 14971:2007 [4] specifies a process for a manufacturer to identify the hazards associated with medical devices, including in vitro diagnostic (IVD) medical devices, to estimate and evaluate the associated risks, to control these risks, and to monitor the effectiveness of the controls.
- IEC 62304:2006 [3] defines common guidelines for the life cycle of software of medical devices.
- IEC 60601-1:2005 [2] contains requirements concerning basic safety and essential performance that are generally applicable to medical electrical equipment.
- ISO 13485:2003 [1] specifies requirements for a quality management system where an organization needs to demonstrate its ability to provide medical devices and related services that consistently meet customer requirements and regulatory requirements.

IEC and ISO have published other deliverables (see Appendix A), considering medical device software, they have developed the following documents:

- IEC/TR 80002-1:2009 [6] provides guidance for application of requirements contained in ISO 14971:2007 to medical device software with reference to IEC 62304:2006. This technical report aims to provide guidelines when a software is included in the medical device/system.
- IEC/TR 80002-3:2014 [10] is a Technical Report (TR) and provides the description of software life cycle processes for medical device. These processes are derived from IEC 62304 considering safety class and are aligned with the software development life cycle processes of ISO/IEC 12207:2008 [5] and are presented herein in full compliance with ISO/IEC 24774:2010 [9]. This TR does not address FDA guidance documents or software development tools. It describes the process model for medical device software development limited to life cycle process described in IEC 62304.

The most important standard is IEC 62304:2006 which defines the guidelines for software development and explains how the other standards are integrated into the process (see Sect. 2.1). Furthermore, FDA wrote the *General Principles of Software Validation* [128], that specifies general principles applicable to validation and verification of medical device software (see Sect. 2.2).

## 2.1 IEC 62304: Software Development Life Cycle

---

The IEC 62304 standard has emerged as a global benchmark for management of the software development life cycle and it is adopted by the European Union and the United States. The standard is approved by both IEC and ISO as an international standard. It is adopted by CENELEC (European Committee for Electrotechnical Standardization), ANSI (American National Standard Institute), SFDA (State Food and Drug Administration) of China and Japan Industry. It is recognized by FDA for use in premarket submissions. Even though the standard establishes a common guidelines for medical device software life cycle processes and tasks, it does not give guidelines for validation and verification of the software, even when the medical device is only software. The standard IEC 62304 defines safety classes for software based on the potential to create an injury to the patient. The safety classes are:

- Class A: No injury or damage to health is possible
- Class B: Nonserious injury is possible
- Class C: Death or serious injury is possible

The standard defines “serious injury” as: “*Injury or illness that directly or indirectly*

*a) is life threatening*

*b) results in permanent impairment of a body function or permanent damage to a body structure, or*

*c) necessitates medical or surgical intervention to prevent permanent impairment of a body function or permanent damage to a body structure.”*

IEC 62304 defines specific processes and tasks for each class (higher safety class (B and C) means additional tasks and more detailed documentation) as shown in Table 2.1<sup>2</sup>.

The first step in Table 2.1 (5.1) consists in defining the life cycle model. The plan shall address: - the processes to be used in the development of software; - the deliverables of the activities and tasks; - how to achieve traceability among system requirements, software requirements, software test, and risks control; - software configuration and software problem resolution. The development plan shall be updated as appropriate.

Software requirements (5.2) consists in defining and documenting functional and non-functional requirements. It requires to measure the risk of potential software defects in the requirements and to re-evaluate the risk when the requirements are established or updated. Moreover, the manufacturer should verify software requirements to avoid contradiction and ambiguity.

Software architecture (5.3) is obtained from the software requirements. It requires to describe the software structure, to identify the software elements, to specify functional and performance requirements for the software elements, to identify software elements related to risk control, and to verify the software architecture with respect to the software requirements.

Regarding the software architecture shall be detailed into software units and

---

<sup>2</sup>The numbers are the same used in the IEC 62304 standard

N	Software documentation	Class A	Class B	Class C
5.1	Software development planning	X	X	X
5.2	Software requirements analysis	X	X	X
5.3	Software architecture		X	X
5.4	Software detailed design			X
5.5	Software unit implementation	X	X	X
	Software unit verification		X	X
5.6	Software integration and integration testing		X	X
5.7	Software system testing	X	X	X
5.8	Software release	X	X	X
6	Software maintenance process	X	X	X
7	Software risk management process		X	X

**Table 2.1:** IEC 62304 software development process

for each unit a detailed design (5.4) shall be provided.

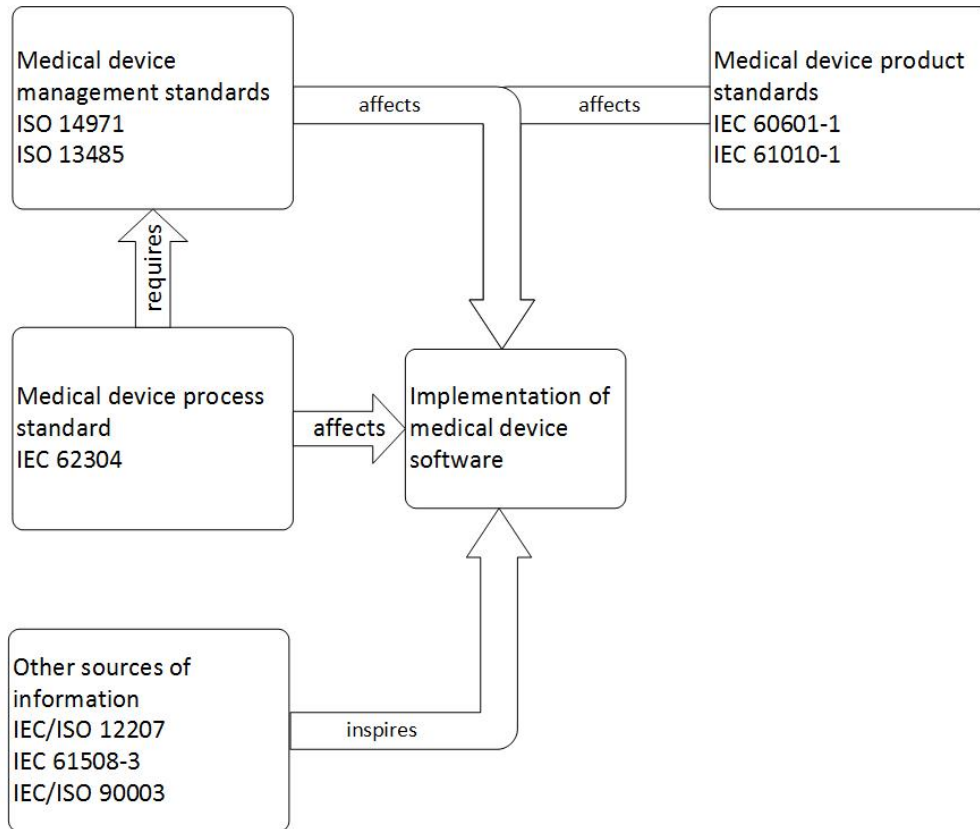
After the implementation of software unit, the manufacturer shall verify it and systematically document the results (5.5). Each software unit is integrated (5.6) based on the integration plan and the final system is verified. The tests (5.7) are performed again after each changes on the software. The software release (5.8) includes the demonstration, by a device manufacturer, that software has been validated and verified. In case of anomalies the manufacturer has to document and establish a process to resolve them. Also the maintenance process (6) is defined. The last step that the manufacturer shall follow is the analysis of risk (7). This analysis aims to identify causes that could contribute to a hazardous situation.

**Integration of IEC 62304 with other standards** Figure 2.1 shows how other standards are integrated into the process described in IEC 62304:2006. These standards are briefly described and not further detailed, because they consider hardware/management of medical devices.

ISO 13485:2003 and ISO 14971:2007 provide a management environment that lays out a foundation to develop products. ISO 13485:2003 defines requirements for the quality management system, in particular for the design and development of planning, inputs, outputs, review, verification and validation. ISO 14971:2007 amplifies requirements for the risk management process to evaluate patient, operator, other person, other equipment and environment risks. It introduces a process through which the manufacturer can identify hazards associated with a medical device, estimate and evaluate risks, control the risks and monitor the effectiveness of the control.

ISO/IEC 90003:2014 [11], IEC 61508-3:2010 and ISO/IEC 12207:2008 [5] can be used as additional guidelines, methods, tools and techniques to implements re-

## 2.2. FDA: General Principles of Software Validation



**Figure 2.1:** *Integration of standards with IEC 62304:2006*

quirements in IEC 62304:2006. ISO/IEC 90003:2014 provides guidance to the acquisition, supply, development, operation and maintenance of computer software and related support services. IEC 61508-3:2010 defines objectives and requirements for safety software life cycle steps. Moreover, it introduces objectives and requirements regarding the integration between hardware and software in electronic devices. ISO/IEC 12207:2008 defines requirements for software life cycle process in general, i.e. not restricted to medical devices. IEC 60601-1:2005 and IEC 61010-1:2010 [7] give specific direction for creation of a safe medical device. In more details, both standards refers to the electrical part of medical device. IEC 61010-1:2010 specifies safety requirements for electrical test and measurement equipment, electrical industrial process-control equipment and electrical laboratory equipment. IEC 60601-1:2005 contains general requirements concerning safety and performance applicable to medical electrical components of the device.

## 2.2 FDA: General Principles of Software Validation

Food and Drug Administration is a federal agency of the United States and is responsible for protecting and promoting public health through the regulation and supervision of medical devices. The *General Principles of Software Validation* [128] defines several broad concepts that can be used as guidance for software

validation and verification activities. Moreover, it requires these activities to be conducted throughout the software development life cycle. The document lists the following general principles as guidelines:

1. A documented SOFTWARE REQUIREMENTS specification should provide a baseline for both V&V.
2. Developers should use a mixture of methods and techniques to PREVENT THE INTRODUCTION OF DEFECTS into the software development process and not on trying to test the software code after it is written.
3. Software validation should be planned early and conducted throughout the software life cycle.
4. SOFTWARE LIFE CYCLE contains documents necessary to support the software validation and software engineering tasks. Furthermore, software life cycle contains verification and validation tasks that guarantee the intended use of the software.
5. Software V&V process should be defined and controlled through the use of a PLAN.
6. Software V&V process should be executed through the use of PROCEDURES.
7. When a change occurs, VALIDATION analysis should be conducted not just for the individual change, but also to determine the impact on the entire software system. The developer should show that the change has not altered the system behaviour.
8. VALIDATION COVERAGE should be based on software complexity and safety risk. As the risk increases, additional validation activities should be added to guarantee the coverage of all risks.
9. V&V activities should be conducted using the quality assurance precept of INDEPENDENCE OF REVIEW to guarantee the quality of software. It is better that V&V are assigned to staff members that are not involved in design or implementation of software.
10. Device manufacturer has FLEXIBILITY in choosing validation principles, but retains ultimate RESPONSIBILITY for demonstrating that the software has been validated.

### 2.3 Conclusion

---

In this chapter, standards and guidelines adopted for the certification of medical device software are identified. The standard IEC 62304:2006 identify the step of software development life cycle, without reference to a specific life cycle. The company who wants to certify the software has to map his process applied for the software development with the clauses of IEC 62304. To guarantee that a software is certifiable, the mapping should cover 100% of IEC 62304 clauses. When the document is available, it is submitted to the organisations responsible for certifying

the software. The organisation evaluates the documentation and provides the certification or changes to fit the presented process with the standard. The same process is applied to get the certification by FDA.

If we consider rigorous methods from the point of view of FDA, they are suggested as methodology for the development of medical device software [78]. For this reason, in Section 4.2.2 we show the compliance of ASMs with FDA principles. We have applied the same process to the standard IEC 62304. In Section 4.2.1, the mapping between the standard IEC 62304 and ASMs process is presented, but not all the steps are covered by ASMs.





---

## Systematic Literature Review

---

Systematic Literature Review (SLR) is the review of a clearly formulated question that uses systematic and explicit methods to identify, to select, to appraise relevant research, to collect and to analyse data from the studies that are included in the review. Through this analysis, we give an overview of the research literature about formal methods applications to model, to verify and to validate medical systems. Moreover, we consider processes and tools to translate models written using formal languages in machine code. We would like to underline that the SLR considers only formal methods applied to medical device/software.

The goal of SLR is twofold:

- to *provide guidance to researchers* starting to work on this topic
- to assess *the state of the art* which is more useful for researchers already working on this subject.

We have applied a Systematic Literature Review process to the topic of rigorous methods for designing and validation of medical software and systems. The goals of this process are (1) to gather a sufficient number of relevant articles, (2) to perform a series of analyses, and (3) to publish the results of the findings to allow researchers to browse in the collected data. This activity follows a systematic process to avoid possible biases, inclusive in order to include as much information as possible, but at the same time capable of identifying only relevant papers. We followed the guidelines presented in [56,84,92] with some adjustments.

### 3.1 The SLR Process

---

Systematic Literature Review is the review of a clearly formulated question that uses systematic and explicit methods to identify, to select, to appraise relevant

research, to collect and to analyse data from the studies that are included in the review [74]. SLR defines a precise process for literature review: criteria for inclusion and exclusion are explicitly stated and therefore it provides a transparent and replicable selection process. The final aim is to minimize the bias and increase objectivity of the review outcomes.

In Figure 3.1 is shown the SLR process, it is divided into two main steps:

- **Papers collection:** collect papers for the SLR analysis;
- **Classification and analysis:** classify and analyze papers obtained in the first step.

The final goal of this process is to provide a list of papers that fit with the topic of the SLR and several analysis over the selected papers. The first step consists in the definition of terms used to perform queries on different databases. Once the user runs queries on databases (see Section 3.1.1 for further details on databases), the results can be saved in different file formats. We have chosen the `.bib` file type, which is very often used for the bibliography, it adopts a standardized format, and it is easily processable with tools and own scripts since it is a textual file. After that, all obtained `.bib` files are merged into the `Merged.bib` file. The user checks whether all already known papers (a list of papers already known to the user about the topic) are included into the file `Merged.bib`. If not, s/he has to identify terms from omitted papers and rerun queries to also include them. Then the pre-processing activity is performed (e.g., delete duplicates and unrelated papers). The result of pre-processing is the `Final.bib` file ready for the analysis. Then `Final.bib` is translated into the `.xml` file using our own `bib2xml` script.

Starting from the `.xml` file, we perform QUANTITATIVE ANALYSIS (see Section 3.1.2) to automatically extract a set of statistical information. Quantitative analysis has a low degree of human interaction because only fields in the bibliographic entries are used (e.g., year, type of publication, and number of citations). We use our own scripts written using EXtensible Stylesheet Language Transformation (`.xslt`), and results are saved into `.txt` or `.csv` files. After that, results are plotted using `pgfplot` package provided by L<sup>A</sup>T<sub>E</sub>X. The quantitative analysis results are statistical data obtained using mathematical measurements and calculations.

On the other hand, starting from `Final.bib`, we collect PDF files of resulted papers and complete a QUALITATIVE ANALYSIS to provide more technical details (e.g., the case study analyzed and the notation used). Qualitative analysis (see Section 3.1.2) requires user interaction. Furthermore, starting from the `.xml` file, we generate a HTML site by using the `.xslt` script. The web site contains all publications listed in the `Final.bib` file and is available on-line<sup>1</sup>.

### 3.1.1 Papers selection

During a preliminary analysis [39], we used only Scopus (<http://www.scopus.com>), but we missed some relevant papers published in journals and in proceedings not included in the Scopus database. For this reason, now we consider other repositories besides Scopus, i.e., Web of Science (<https://apps.webofknowledge.com>), DBLP (<http://dblp.uni-trier.de/>) and Google Scholar (<https://scholar>).

---

<sup>1</sup>The list of publications is available at <http://cs.unibg.it/bonfanti/FMMedicalDeviceSLR/>

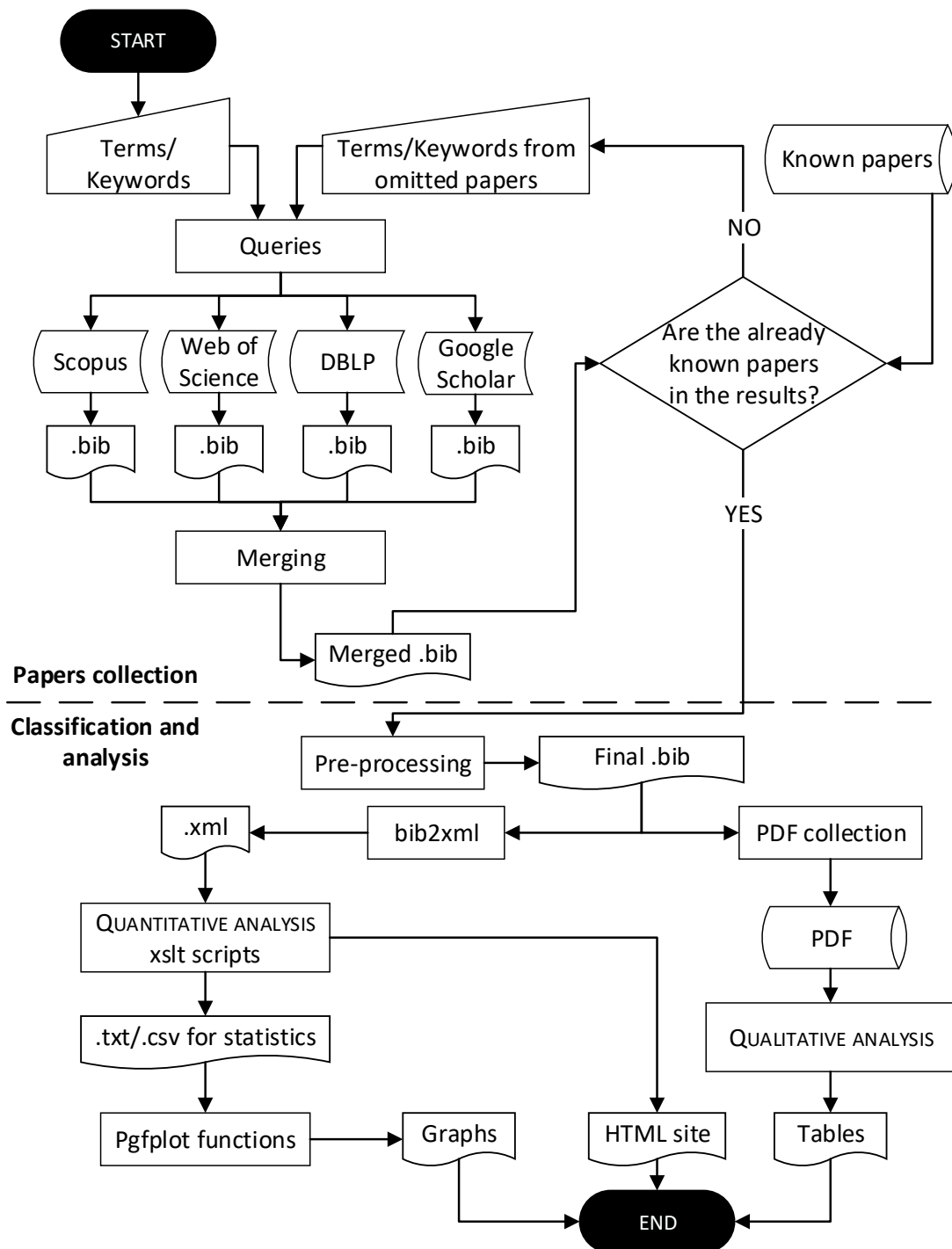


Figure 3.1: SLR Process

google.com/). We do not consider IEEE Xplore (<http://ieeexplore.ieee.org/>), Springer (<http://www.springer.com>), and ACM Digital Library (<http://dl.acm.org/>) because they are already included in Scopus and DBLP. Furthermore, we noticed that the search function of IEEE Xplore has some limitations: during the search activity some queries returned unrelated results which did not

contain the terms entered in the query.

**Scopus** is the largest database owned by Elsevier, it contains scientific journals, books and conference proceedings. It encloses more than 60 million records, over 21.500 peer-reviewed journals, over 360 trade publications, 7.2 million conference papers, 27 million patents, 5.000 articles-in-press from international publishers including Cambridge University Press, the Institute of Electrical and Electronics Engineers (IEEE), Nature Publishing Group, Springer, Wiley-Blackwell. It includes more than 113.000 books that will increase by 10.000 each year. Thomson Reuters maintains **Web of Science** that provides a citation search. It gives access to several database in different disciplinary researches: Conference Proceedings Citation Index (covers more than 160.000 conference titles), Science Citation Index Expanded (covers more than 8.500 journals encompassing 150 disciplines), Social Sciences Citation Index (covers more than 3.000 journals in social science), Arts & Humanities Citation Index (covers more than 1.700 arts and humanities journals), Book Citation Index (covers more than 60.000 books), Current Chemical Reactions (more than one million) and Index Chemicus (more than 2.6 million compounds). **Dblp** provides open bibliographic information of major computer science journals and proceedings. It contains more than 3 million publications relating to more than 4.500 conferences and more than 1.400 journals. This service is provided by University of Trier and Schloss Dagstuhl. **Google Scholar** furnishes a way to search literature, but contains large quantity of documents and it is difficult to select those important. It is possible to search across many sources: articles, theses, books, abstract, court opinions, academic publishers, professional societies, on-line repositories, universities, patents. Initially we performed some queries, but a set of known paper were missing. To cope with this problem, we identified new queries to include the missing papers. In this way, we are able to include similar papers as well. The queries are a combination of terms using “and” and “or” logical operators. The majority of queries perform the search only in titles, while few of them are executed on author keywords. This limitation is due to the search functions provided by the repositories, as shown below. Due to the intrinsic limited functionality, the number of queries run for each repository is not the same. Afterwards, the queries executed are listed for each repository. The keywords used depends on the person who perform the SLR. We have choose the below keyword because the SLR is about the application of **formal methods** applied for **validation** and/or **verification** to a **medical software** or **device**. The keywords **pacemaker**, **infusion pump** and **cardiac** are added because they derived from known papers (see the recursion in Figure 3.1).

**Scopus** Scopus has the most performing search system. It is possible to execute an advanced search e.g. by title (TITLE), author keywords (AUTHKEY) and combine them using logical operators. Furthermore, it allows to search terms that start with a prefix or end with a suffix by using ‘\*’ symbol (e.g. “method\*” means “method, methods, methodology, methodologies”). For the research purpose the queries performed are:

- TITLE (medical) AND (TITLE (software) OR TITLE (device\*)) AND TITLE (formal method\*)

- TITLE (formal specification) AND TITLE (medical)
- TITLE (formal\*) AND TITLE (infusion pump)
- TITLE (formal\*) AND TITLE (medical software)
- TITLE (formal\*) AND TITLE (medical device\*)
- AUTHKEY (formal) AND (AUTHKEY (medical) AND (AUTHKEY (device\*) OR AUTHKEY (software) OR AUTHKEY (service\*)))
- AUTHKEY (formal method\*) AND AUTHKEY (health)
- TITLE (formal) AND TITLE (pacemaker)
- TITLE (medical) AND (TITLE (software) OR TITLE (device\*)) AND TITLE (\*formal verification)
- TITLE (medical) AND (TITLE (software) OR TITLE (device\*)) AND (TITLE (verification))
- TITLE (medical) AND TITLE (formal method\*)
- TITLE (formal AND cardiac AND (specification OR modeling OR modelling))

For each query a set of papers has been found, some of them were not consistent with our research and some papers were duplicate.

**Web of Science** Web of Science is similar to Scopus but it is not allowed to perform advanced search e.g. by keywords or by author keywords. Due to the limited advanced search respect to Scopus, less queries have been performed:

- TI = (medical) AND (TI = (software) OR TI = (device\*)) AND TI = (formal method\*)
- TI = (formal specification) AND TI = (medical)
- TI = (formal\*) AND TI = (infusion pump)
- TI = (formal\*) AND TI = (medical software)
- TI = (formal\*) AND TI = (medical device\*)
- TI = (formal) AND TI = (pacemaker)
- TI = (medical) AND (TI = (software) OR TI = (device\*)) AND TI = (\*formal verification)
- TI = (medical) AND (TI = (software) OR TI = (device\*)) AND (TI = (verification))
- TI = (medical) AND TI = (formal method\*)

**Dblp** Dblp has a limited search technology. It is not possible to specify the search field (e.g. title, keywords) and it is not allowed the words search by suffix or prefix. Considering these limitations, the queries chosen are more bounded:

- medical software | device formal method
- formal specification medical

- formal infusion pump
- formal medical software
- formal medical device
- formal pacemaker
- medical software | device verification
- medical formal methods

**Google Scholar** Google Scholar search provides two options: the first limits the search to the title, the second extends the search to all document (title, content, keywords, authors) at the same time. The second option introduces many papers out of scope, so only the first is considered. The queries executed in this repository are:

- allintitle: medical formal (method OR methods) software OR devices OR device
- allintitle: formal specification medical
- allintitle: formal infusion pump
- allintitle: formal medical software
- allintitle: formal medical device OR devices
- allintitle: formal pacemaker
- allintitle: medical formal verification software OR device OR devices
- allintitle: medical formal method OR methods

Google Scholar includes unaudited documents e.g. technical reports, that are out of search because they are not formally recognized by the community. Furthermore, we left out patents and citations.

In Table 3.1, the number of queries executed for each repository and the number of entries found are shown (328 papers). Note that, the number of entries includes duplicates (177 papers) between the queries in the same repositories, because some papers fit in more than one query. Duplicates exist also between different repositories because they search papers in common databases. The remaining papers, 151, have been analysed and 80 were not consistent with our SLR topic. 71 paper have been analysed in the classification and synthesis section.

	Scopus	Web of Science	DBLP	Google Scholar
N° queries	12	9	3	8
N° entries	134	55	60	79

**Table 3.1:** *Number of queries and entries for each repository*

By following the process defined in Figure 3.1, the next step consists of exporting all papers found in the repositories and merging them in the `Merged.bib` file.

### 3.1.2 Classification and Synthesis

Before performing classification, we made pre-processing activities that consist of the following steps:

1. conform authors names using the format (`first name initial, surname`),
2. update authors names: add missing letters, e.g., ö, é, ś,
3. delete duplicates,
4. delete unrelated papers.

The first activity is automatically performed using our own script. The script analyses the author field and adjust names and surnames considering the target format. We perform the second activity manually, because we did not find any process to make it automatic. The third activity is performed using JabRef tool<sup>2</sup>. The tool has a function to find duplicates, after a duplicate is identified the user can choose how to handle it. The possible solutions are: keep one of them, keep both, or keep the merged entry only. The last activity is the most time-consuming because it requires involvement of the one who is performing the SLR. We analyze all papers by reading their abstracts and over-viewing their content in ambiguous cases. We delete all papers that do not describe the application of formal methods to a medical device software. Moreover, we remove all documents that are not peer-reviewed in order to maintain the number of documents manageable and the quality of the collection high. The result of pre-processing activity is the `Final.bib` file. Starting from this file, we perform two types of analysis: Quantitative Analysis and Qualitative Analysis.

#### Quantitative analysis

Quantitative analysis has a low degree of human interaction and we performed it by using own scripts written in EXtensible Stylesheet Language Transformation (.xslt). For this analysis, we consider the number of publications written over the years, the type of publications, the number of papers written by authors and the number of citations. We answer the set of research questions as shown below.

**RQ1: Which is the trend of publications?** As a first question, we wanted to observe the trend of publications about formal methods applied in medical field. We analyze the number of publications from 1992 (the year of the oldest publication we found) until 2016. As shown in Figure 3.2, until 2004 only three publications have been published. Starting from 2004, the interest on application of formal methods to medical devices increases. From 2011, the behavior of the number of papers shows a progressive increase until 2016<sup>3</sup>. The overall behavior of the graph shows an increasing interest in this topic by the community mostly in the last years.

<sup>2</sup><http://www.jabref.org/>

<sup>3</sup>The search is performed in January 2017, few papers published at the end of 2016 could be missed due to the delay in databases update.

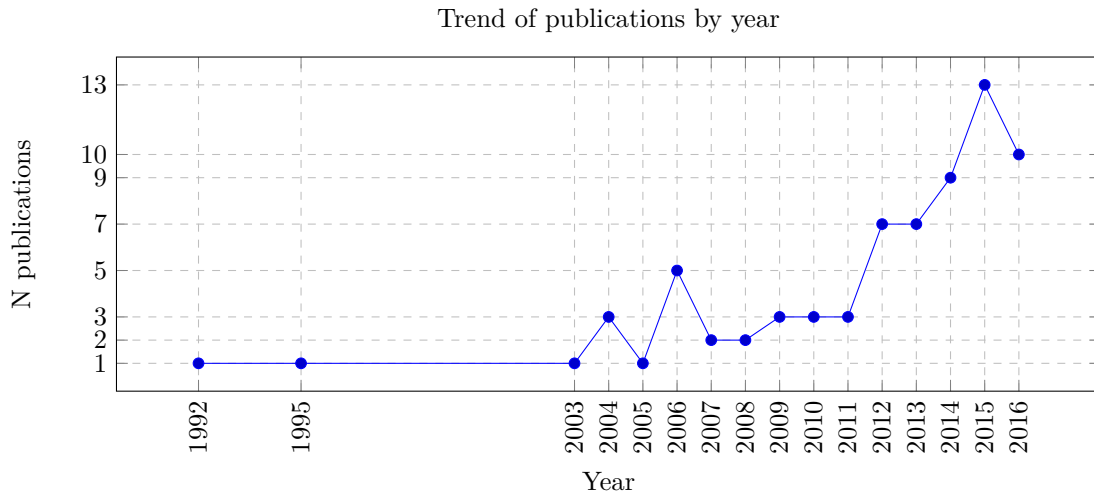


Figure 3.2: RQ1: Trend of publications

**RQ2: Which is the trend of publications considering the type?** In Figure 3.3, the pie chart shows the percentage of publications grouped by type. Note that from the databases we obtain only journals and conference papers (InProceedings). The number of publications in proceedings (approx. 63%) is greater than the number of publications in journals (approx. 37%).

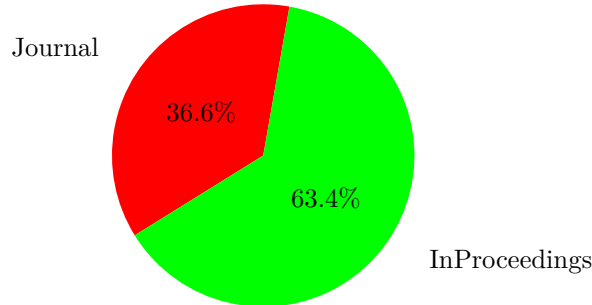
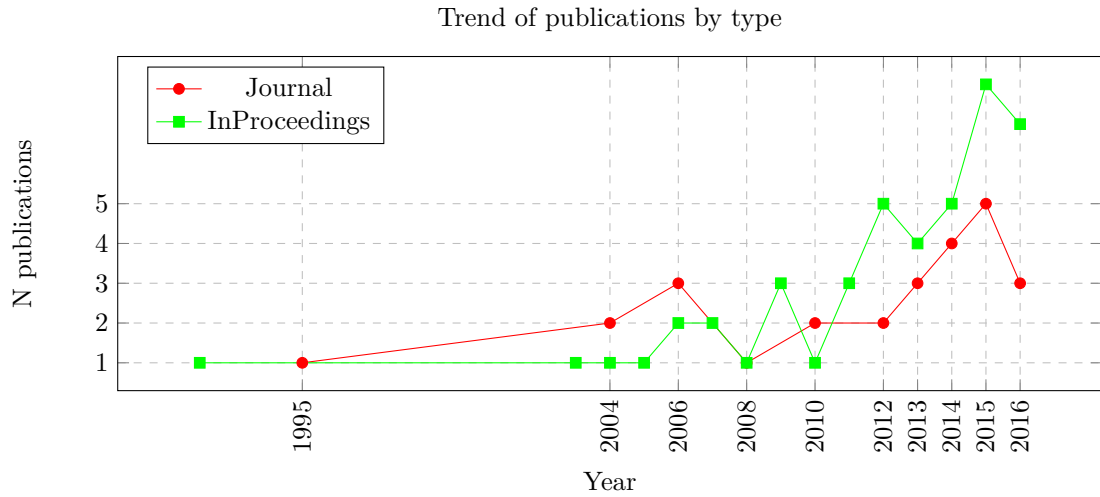


Figure 3.3: RQ2: Types of publications

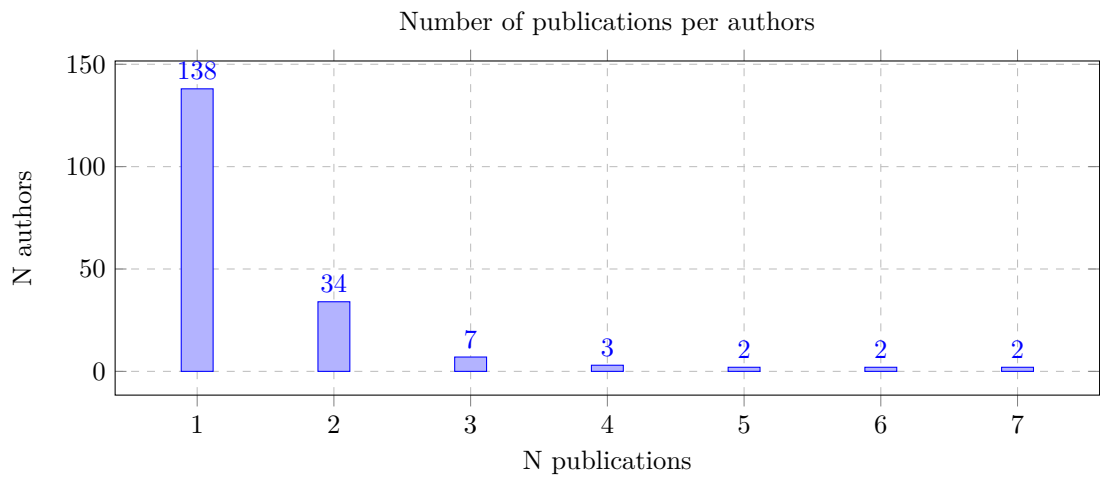
Figure 3.4 depicts the trend of the number of publications in journals and in proceedings. In the last few years, we notice an increase in number of publications in conferences. This confirms that the subject is still rather novel and there is an increasing interest of the computer science community towards this topic.

**RQ3: How many papers about this topic have been written by the same author?** Figure 3.5 shows the number of publications per author. The most apparent observation is that the majority of authors (approx. 73.5%) have published only once about this topic and 18% of authors have two publications. Only 8.5% of authors have more than two publications. An explanation could be that this topic is rather new in the scientific community and authors are starting their activities in these years. We expect an increase in number of publications per author in upcoming years due to ongoing scientific studies.

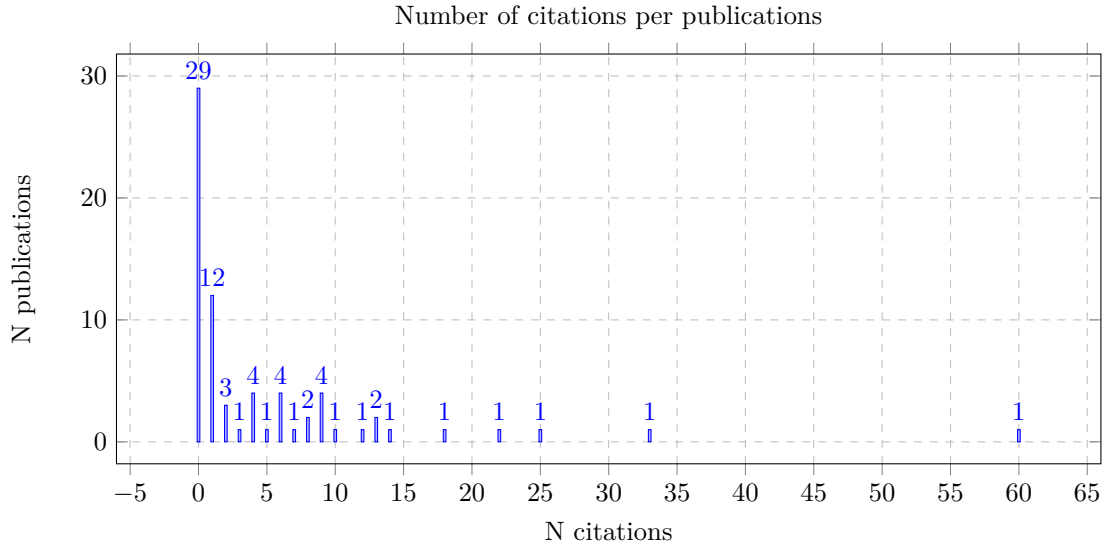




**Figure 3.4:** *RQ2: Publications in Journal/Proceedings per year*



**Figure 3.5:** *RQ3: Number of papers by the same author*



**Figure 3.6:** *RQ4: Number of citations/publication*

**RQ4: Which are the most cited publications?** Before introducing which are the most cited papers, we analyzed the general behavior of the number of citations (see Figure 3.6). Overall, approx. 40% of publications do not have citations. Approx. 45% of publications have less than ten citations and approx. 15% of publications have more than ten citations. This low percentage of citations could be due to the novelty of this topic in the scientific community.

Table 3.2 shows the most cited publications by considering only the citations given by Scopus. The publication with most citations [127] presents the utilization of formal methods in the improvement of medical protocols. A new formal language and theorem prover have been defined to help medical experts in medical protocol definition. The second paper [78] introduces the importance of formal methods used in pre-market and post-market analysis. Paper number three [28] applies formal methods to infusion pumps. The authors model the system using Extended Finite State Machines (EFSM) and apply validation and verification techniques using the UPPAAL framework. Furthermore, they generate test cases from the formal specification of the system to test the machine code. Also paper four [14] models the infusion pump using EFSM and applies verification technique using the Software Cost Reduction (SRC) tool. Paper number five [77] models a Clinical Neutron Therapy System using the Z notation. Paper number six [68] models pacemaker using Z tools. We can notice that the majority of these papers are regarding infusion pump and pacemaker case studies. These case studies are those provided by authoritative groups as explained in RQ6. In paper number seven [89], authors verify the model of pacemaker using bounded model checking. Paper number eight [106] models pacemaker using the Event-B method and performs validation and verification activities. Paper number nine [49] verifies pulse oximeter using different formal tools. Infusion pump is modeled and verified in the last paper [94] using Prototype Verification System (PVS) tools.

N	Publications	# cit
1	[127]A. Ten Teije, M. Marcos, M. Balsler, J. Van Croonenborg, C. Duelli, F. Van Harmelen, P. Lucas, S. Miksch, W. Reif, K. Rosenbrand, and A. Seyfang. Improving medical protocols by formal methods. <i>Artificial Intelligence in Medicine</i> , 36(3):193–209, 2006	60
2	[78]R. Jetley, S.P. Iyer, and P. Jones. A formal methods approach to medical device review. <i>Computer</i> , 39(4):61–67, 2006	33
3	[28]D. Arney, R. Jetley, P. Jones, I. Lee, and O. Sokolsky. Formal methods based development of a PCA infusion pump reference model: Generic infusion pump (GIP) project. In <i>High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability</i> , pages 23–33, Cambridge, MA, June 2007	25
4	[14]R. Alur, D. Arney, E.L. Gunter, I. Lee, J. Lee, W. Nam, F. Pearce, S. Van Albert, and J. Zhou. Formal specifications and analysis of the computer-assisted resuscitation algorithm (CARA) Infusion Pump Control System. <i>International Journal on Software Tools for Technology Transfer</i> , 5(4):308–319, 2004	22
5	[77]J. Jacky. Specifying a safety-critical control system in Z. <i>IEEE Transactions on Software Engineering</i> , 21(2):99–106, Feb 1995	18
6	[68]A.O. Gomes and M.V.M. Oliveira. Formal specification of a cardiac pacing system. In <i>International Symposium on Formal Methods - FM 2009</i> , volume 5850 LNCS, pages 692–707, 2009	16
7	[89]C. Li, A. Raghunathan, and N.K. Jha. Improving the trustworthiness of medical device software with formal verification methods. <i>IEEE Embedded Systems Letters</i> , 5(3):50–53, Sept 2013	14
8	[106]D. Méry and N.K. Singh. Formal specification of medical systems by proof-based refinement. <i>Transactions on Embedded Computing Systems</i> , 12(1), 2013	13
9	[49]L. Cordeiro, B. Fischer, H. Chen, and J. Marques-Silva. Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In <i>Proceedings - 2009 International Conference on Embedded Software and Systems, ICESS 2009</i> , pages 396–403, 2009	13
10	[94]P. Masci, P. Curzon, M.D. Harrison, A. Ayoub, I. Lee, and H. Thimbleby. Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example. In <i>EICS 2013 - Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems</i> , pages 81–90, 2013	12

Table 3.2: Publications with most citations

### Qualitative analysis

Qualitative analysis is based on user interaction. It cannot be carried out by automatic scripts like quantitative analysis. The publications have to be analysed by the user to extract the following information:

- the case study analysed (e.g. pacemaker, infusion pump)
- the target of the application (e.g. model verification, model validation, certification)
- the notation used (e.g. automata, state chart, abstract state machine)
- the tools used (e.g. MATLAB, SPIN, Asmeta, Rodin)
- the methodology applied (e.g. modelling, modelling by refinement, verification, conformance checking)

Notation type	Languages
Logic	Higher-Order Logic, Linear Temporal Logic (LTL), Computational Tree Logic (CTL), Temporal Ordering of Events, Timed Computational Tree Logic (TCTL)
State Based	Automata, B, Extended Finite State Machines (EFSM), Abstract State Machine (ASM), Z, Circus, State Machine, Event-B, Vienna Development Method (VDM), UML activity diagram, UML state machines, Algebraic State-Transition Diagrams (ASTD)
Event Based	Predicate/Transition Nets, Petri Nets, Activity Newtorks, Timed Transition System (TTS)
Programming Languages	Visual Contract Language (VCL), C, Mixed Signal Assertion Language (MSAL)

Table 3.3: Notations used in the literature

**RQ5: Which are the notations used?** In Table 3.3, the notations used in resulted papers are classified into five macro categories:

- Logic: the notation is based on a logical language that consists of logical symbols and is characterized by having a fix interpretation. The combination of these symbols compose well-formed formulas.
- State Based: in a state-based approach, an execution of a system is viewed as a sequence of states, where a state is an assignment of values to some set of components [12].
- Event Based: an event-based approach views an execution as a sequence of events [12].
- Programming Languages: the notation is based on programming languages.

The most used notations are Automata, Event-B, Z, and Extended Finite State Machine (EFSM). All of them are state-based notations. One of the possible reasons why state-based notations are popular in medical software systems is because they support a model-driven development paradigm where requirements are transformed into functional code through a systematic process. In this fashion, it is easier to manage complexity, it is easier to reason about the behavior of the system, and efforts spent in earlier phases of the development ultimately result in generation of code that is correct by construction.

**RQ6: Which are the case studies analyzed and which activities are performed?** Table 3.4 shows which are the methods applied for each case study analyzed Starting from the second column, we have identified a set of steps applied during the software development process. The first activity performed is modeling. Depending on the tool used by the user, a system is modeled using different notations (see RQ5). After that, a set of activities can be performed on the model. Model verification verifies whether certain properties are compliant with the model definition. Model simulation checks the behavior of the system. Software validation analyses the behavior of software as compared to the model. Code generation derives software directly from the model previously defined using a tool. Certification activity aims to identify a connection between the process applied using formal methods and the standards/guidelines that guide the medical software cer-

tification. The last column of Table 3.4 collects papers that provide a theoretical study concerning medical devices specified in the first column. The first column of Table 3.4 shows the application of formal methods to medical devices. The most analyzed case study is about infusion pumps provided by FDA [132]<sup>4</sup>. The case study invites the formal methods community to provide techniques and tools that improve the overall reliability of medical devices. Another case study provided by Software Quality Research Laboratory (SQRL) to the formal methods community is pacemaker<sup>5</sup>. Recently, at ABZ 2016 conference<sup>6</sup>, the hemodialysis machine case study has been provided to advocate the use of formal methods in medical applications [100]. The remaining case studies proposed by researchers are derived from their own experiences.

The hemodialysis machine case study is modeled by [16, 33, 60, 67, 75, 90, 97–99]. Some of them perform validation [16, 75, 98, 99] and verification [16, 33, 67, 90, 97, 98]. Paper [98] generates the machine code starting from formal specification, and paper [16] defines a set of characteristics to fit formal methods with the standards for medical software certification. A model of pacemaker is described in papers [68, 69, 81, 82, 85–87, 104, 106, 115, 118]. Other authors apply model verification [81, 82, 85, 86, 89, 106, 115, 118] and model validation [106, 125]. Papers [69, 81, 104] contribute to develop a new step that consists in translating the model into machine code. The infusion pump case study is modelled [14, 28, 31, 34, 37, 52, 72, 79, 80, 93–96, 107, 109, 113, 116, 119], verified [14, 28, 31, 52, 72, 79, 80, 93–96, 107, 113, 116, 119] and validated [34, 37, 80, 116, 119] by using different tools and languages. Paper [93] generates the machine code starting from the formal model and in [28, 79] authors use different approaches to validate software. In paper [52], authors develop a method to support the certification process, indeed, Jetley et al. [78] explain how to apply formal methods to premarket and post-market evaluations in case of medical devices (infusion pump). Formal methods are also used in medical image processing for modeling and model verification [27, 29, 53, 55]. The paper [108] provides a survey about formal methods applied to image processing. Stereoacuity test [15] determines which is the acuity level in patients. The paper shows a process that includes modeling, model simulation, model verification, model validation and software validation by comparing the code with the formal requirements model. Electrocardiography is the process of recording the electrical activity of the heart. The system is modeled [35, 36, 105, 122, 123], verified [105], and validated [122, 123] using different tools. E-health systems are a recent classification of health-care systems supported by electronic processes and communication. Paper [126] models, verifies and validates the system, while [30] models an e-health system using a different language. Suggestions on how to use formal methods are given also in the medical protocol case study [32]. Papers [70, 127] propose a model of medical protocol and verify whether the overall process is correct or not. Model verification is performed for systems responsible for pulse oximeter [49] and imatinib dose [117]. The models [43, 48, 51, 88, 91, 101, 120] of medical device connections are verified [43, 48, 51, 54, 76, 88, 91, 101, 120] and vali-

<sup>4</sup><http://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/>

<sup>5</sup><http://sqr1.mcmaster.ca/pacemaker.htm>

<sup>6</sup><http://www.cdcc.faw.jku.at/ABZ2016/>

dated [48, 51, 76, 120]. Model validation, model verification and modelling activities are performed for left ventricular assist device [13]. A formal model is defined for clinical neutron therapy system [77] and for syringe pump [42], furthermore syringe pumps are validated in papers [42, 125]. Overall, the most common activities performed are modelling, model verification and model validation. A trend towards code generation has also been observed in recent publications.

**RQ7: Which are the tools used for each performed activity?** Table 3.5 shows which tools or tool families are used for each software development activity. The majority of tools operate on a models written in their specific languages. After the modeling process, a number of activities are performed on the model. The validation step checks whether the model specified by the user behaves properly. Verification, on the other hand, analyses whether the model is consistent and expected properties are verifiable. There exist different verification techniques, those used in the papers found are:

- *Model Checking*: it is applied to finite-state systems. The properties are translated in formulas of a temporal logic (e.g., CTL -Computational Tree Logic- or LTL -Linear Temporal Logic) and efficient symbolic algorithms are used to verify that all possible configurations validate the specified properties. If the property is false, a counterexample is displayed. The limitation of this technique is the state space explosion: the transition graph grows exponentially making difficult the development of an efficient algorithm.
- SMT (Satisfiability Modulo Theories) solvers: they are given an expression with boolean variables along with and/or predicates, based on which they determine the conditions that would make the expression true.
- Theorem Provers: they prove axioms derived from the desired behaviour of the system. Support tools, however, are not always able to provide a proof given an axiom.

Software validation returns differences between the behaviour captured by the real system and the requirements model defined by the user. Code generation automatically produces the code for the real system, while test case generation derives test cases for the real system.

Regarding the types of activities, Table 3.5 hints that formal approaches are primarily used for modelling purposes. Also verification and in particular model checking play a predominant role in this area. The main advantage of model checking is that it is fully automated, so it is the preferred mean of verification. Activities like software validation (e.g., testing) and code generation are surprisingly not common areas for the application of formal methods in medical software systems.

Regarding the tools used, we can see that most approaches use a rather small set of tools including B tools, MATLAB and Simulink, and UPPAAL. These tools have a good support and a commercial backing. Although tools like Mathworks' MATLAB and Simulink are strictly commercial, other tools like B tools and UPPAAL allow and encourage non-commercial uses as well. Other tools are used only in few case studies.

---

## 3.2 Conclusion

---

In Section 3.1.2 qualitative analysis and quantitative analysis have been performed. In the last years, the interest of the communities to the applications of formal methods to medical device development is increased due to the critical issue of the topic. The novelty of the topic is confirmed by the type of publications as there are more publications in conferences than in journals. The topic novelty explains the fact that authors do not have a large number of publications and the papers are not cited by many publications. The most cited papers apply formal methods to infusion pump and pacemaker case studies, the case studies provided by authoritative groups. These are also the case studies most analysed in general by the community (see Table 3.4). Considering the tools, the most used are B tools, MATLAB and Simulink and UPPAAL. This is reflected also in RQ5, where the most used notations are the state-based notations, those handled by the most used tools.

**Comparison of ASMs with other approaches** Considering the ASMs, only the case studies presented in this thesis have been found<sup>7</sup>. The main activities presented using ASMs are modelling, model validation, model verification using model checking and software validation. If we consider the all methodologies applied by the other approaches, we can notice that ASMs cover most of the activities presented in Tables 3.4 and 3.5. The code generator from ASMs is not applied to medical device case studies because it is a new feature of the **Asmeta** framework and it was not available at the time of the case studies development. Furthermore, the current version (see Chapter 7) translates ASMs in C++ code for Arduino, which is not consistent with the final platform associated to the medical case studies.

Compared to the other development environments, **Asmeta** provides an integrated environment. For each activity performed, a tool is available and all the tools are integrated with each other. This is a vantage of this framework because the users do not have to care about the transformation of models into different language to perform different activities.

---

<sup>7</sup>We are considering the application to medical devices software.

Table 3.4: Application of formal methods to medical devices

Medical device	Main Activity						Survey
	Modelling	Model verification	Model validation	Software validation	Code generation	Certification	
Hemodialysis Machine	[16, 33, 60, 67, 75, 90, 97-99]	[16, 33, 67, 90, 97, 98]	[16, 75, 98, 99]		[98]	[16]	
Pacemaker	[68, 69, 81, 82, 85-87, 104, 106, 115, 118]	[81, 82, 85, 86, 89, 106, 115, 118]	[106, 125]		[69, 81, 104]		
Left Ventricular Assist Device	[13]	[13]	[13]				
Infusion pump	[14, 28, 31, 34, 37, 52, 72, 79, 80, 93-96, 107, 109, 113, 116, 119]	[14, 28, 31, 52, 72, 79, 80, 93-96, 107, 113, 116, 119]	[34, 37, 80, 116, 119]	[28, 79]	[93]	[52]	[78]
Syringe Pump	[42]		[42, 125]				
Medical image processing	[27, 29, 53, 55]	[27, 29, 53, 55]					[108]
Stereoacuity test	[15]	[15]	[15]	[15]			
ECG (Electrocardiography)	[35, 36, 105, 122, 123]	[105]	[122, 123]				
e-Health system	[30, 126]	[126]	[126]				
Medical protocol	[70, 127]	[70, 127]					[32]
Pulse oximeter		[49]					
Medical device connection	[43, 48, 51, 88, 91, 101, 120]	[43, 48, 51, 54, 76, 88, 91, 101, 120]	[48, 51, 76, 120]				
Clinical Neutron Therapy System	[77]						
Imatinib dose	[117]	[117]					



Table 3.5: Tools used for each methodology

	Modelling	Model validation	Verification			Code generation			
			Model checking	SMT	Theorem prover		Other Not specified	Software validation	
<b>MATLAB and Simulink</b>	[34, 35, 52, 79, 80, 85, 107, 116, 122, 123]	[34, 80, 116, 122, 123]	[79, 80, 107]	[107]			[52, 85, 107, 116]	[79]	
<b>SPIN</b>	[55, 80]		[55, 80]						
<b>UPPAAL</b>	[27-29, 53, 79-82, 115, 117]	[80]	[27-29, 53, 79-82, 115, 117]	[115]				[28, 79]	[81]
<b>SCR</b>	[14, 116]	[116]	[14]						
<b>Asmeta</b>	[15, 16]	[15, 16]	[15, 16]					[15]	
<b>Z tools</b>	[30, 31, 43, 68, 77]		[31, 43]						[69]
<b>SAL</b>	[13, 95, 113]	[13]	[95]	[13, 113]					
<b>B tools</b>	[33, 42, 60, 75, 97-99, 104-106, 109, 118, 119]	[42, 75, 98, 99, 106, 119]	[97, 98, 106]		[98]		[33, 105, 118, 119]		[98, 104]
<b>PVS</b>	[37, 52, 93, 94, 96]	[37]			[93, 96]		[52, 94]		[93]
<b>VDM</b>	[126]	[126]					[126]		
<b>ProVerif</b>	[101]						[101]		
<b>Perfect Developer</b>	[69]								
<b>Real-Time Maude</b>	[91, 125]	[125]	[91]						
<b>NuSMV</b>			[49]						
<b>CBMC</b>			[49]				[89]		
<b>SATABS</b>			[49]						
<b>CEGAR</b>	[82]		[82]						
<b>VCB</b>	[87]								
<b>Asbru</b>	[70]								
<b>KIV</b>									
<b>ABV</b>	[90]							[70]	
<b>Yices</b>	[13]		[13]					[90]	
<b>Z3</b>	[115]		[115]						
<b>BLESS</b>	[86]							[86]	
<b>OSCP</b>	[88]							[88]	
<b>IVY workbench</b>	[72]		[72]						
<b>Circus</b>	[67]		[67]						
<b>CellExcite</b>	[36]								
<b>LOTOS tools</b>	[51]	[51]					[51]		
<b>V2T</b>	[48, 120]	[48, 120]					[48, 120]		
<b>MathSAT 5</b>				[54]					





## Part II

# Abstract State Machines

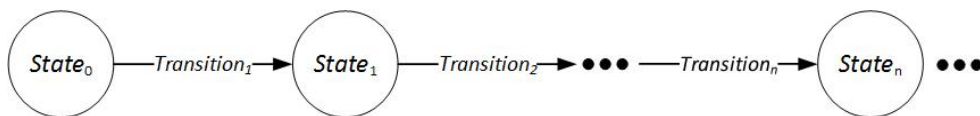


---

## ASM-based development process

---

Abstract State Machines (ASMs) are an extension of Finite State Machines where unstructured control states are replaced by states with arbitrary complex data. A *state* represents the instantaneous configuration of the system under development, and *transition rules* describe the change of state. ASMs *states* are multi-sorted first-order structures, i.e., domains of objects with functions and predicates defined on them. ASMs *transition rules* express how function interpretations are modified from one state to the next one, and therefore describe the system configuration changes.



**Figure 4.1:** *ASMs State-Transition*

The basic form of a transition rule is the *guarded update*: “**if** *Condition* **then** *Updates*”, where *Updates* is a set of update functions of the form  $f(t_1, \dots, t_n) := t$  which are simultaneously executed when *Condition* is true;  $f$  is an arbitrary  $n$ -ary function and  $t_1, \dots, t_n, t$  are terms. An ASMs state is represented by a set of couples (*location*, *value*). ASMs *locations* represent the abstract ASMs concept of basic object containers (memory units). Location *updates* represent the basic units of state change. Besides **if-then**, there is a limited but powerful set of *rule constructors*: **par** for simultaneous parallel actions, **seq** for sequential actions, **choose** for nondeterminism (existential quantification), **forall** for unrestricted synchronous parallelism (universal quantification). Functions that never change during any run of the machine are *static*. Those updated by agent actions are

*dynamic*, and distinguished between *monitored* (only read by the machine and modified by the environment), and *controlled* (read and written by the machine). A *computation* of an ASMs is a finite or infinite sequence  $s_0, s_1, \dots, s_n, \dots$  of states of the machine, where  $s_0$  is an initial state and each  $s_{n+1}$  is obtained from  $s_n$  by simultaneously firing all the transition rules which are enabled in  $s_n$ . The (unique) *main rule* is a transition rule and represents the starting point of the computation. The ASMs can have more than one *initial state*. It is possible to specify state *invariants* (set of conditions always true during the execution).

For system specification, the ASMs method builds upon two further concepts:

- *ground model*, an ASM which is a first reference model for the design;
- *model refinement*, a general scheme for stepwise instantiations of model abstractions to concrete system elements, providing controllable links between the more and more detailed descriptions at the successive stages of system development.

The modelling activity is supported by a number of V&V activities on models, already applicable at the ground level and along the chain of refined models, that helps to guarantee correctness of the developed system.

This is a short introduction about ASMs, more information can be found in [41].

### 4.1 ASMs Modelling, Validation & Verification

---

A rigorous process for ASMs-based development [23], based on the concepts of ground model and model refinement, is depicted in Figure 4.2. The modelling activity is complemented with a number of other activities on models and eventually on code. All these activities help the modeller to develop a correct system in a correct way.

A set of tools exists to support the developer in the various activities and to make the ASMs method useful in practice. The process is not automatic, but it is automatable if an order among the activities is imposed. Tools are part of the **Asmeta** (ASM mETAmodeling) framework<sup>1</sup> [25], and are strongly integrated in order to permit reusing information about models during different development phases. The IDE **AsmEE** is available to assist the user when editing an ASMs model by using the concrete syntax **AsmetaL** [65].

An abstract *ground model* (ASM0 in Figure 4.2) is specified using terms of the application domain by reasoning on the informal requirements (generally given as a text in natural language), possibly with the involvement of all stakeholders. The ground model should be *correct*, i.e., it reflects the intended initial requirements, and *consistent*, i.e., it removes ambiguities of the initial textual requirements. However, it does not need to be *complete*, i.e., it may leave some given functional requirements unspecified.

From the ground model, by step-wise refined models, further details are added to capture all the functional requirements and provide descriptions of the complete software architecture and component design of the system. In this way, the complexity of the system can be always taken under control, and it is possible

---

<sup>1</sup><http://asmeta.sourceforge.net/>

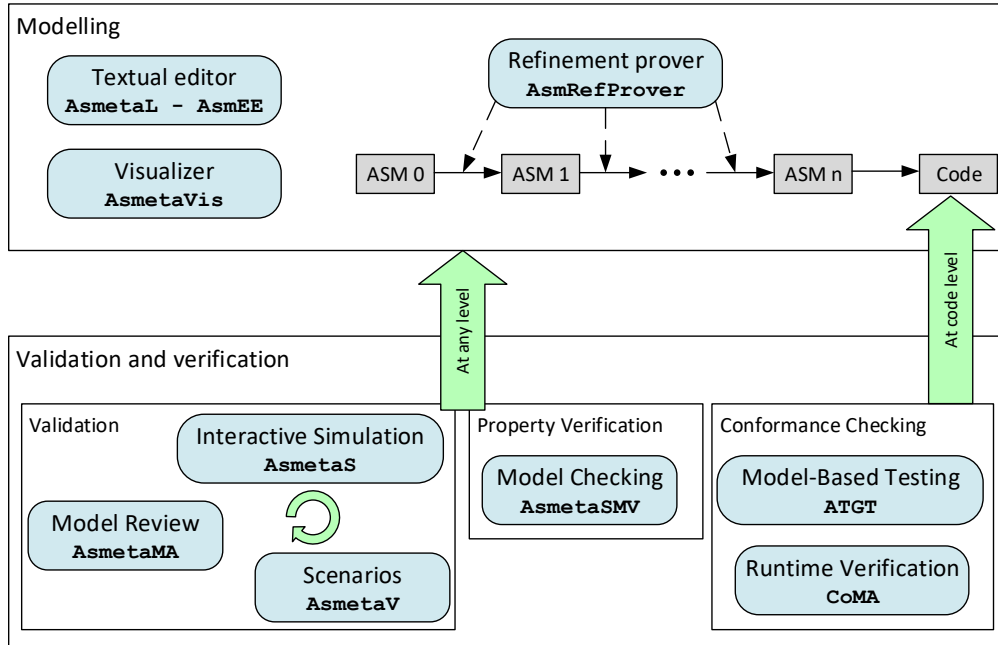


Figure 4.2: ASMs development process

to bridge, in a seamless manner, the gap between specification and code. Each time a model is specified as a refinement of an abstract one, refinement correctness should be checked. This can be done by hand, but we provide an automatic way to achieve this assurance in case of *stuttering refinement*, a restricted form of ASMs refinement. The tool **ASMRefProver** automatically checks stuttering refinement between two ASMs models.

Modelling is sustained by the visualizer **AsmetaVis**, it provides a visual notation that supports the user to better understand the current behaviour of the model (see Chapter 6).

Modelling activity is supported, at each level of refinement, by model *validation* and *verification* (V&V). Model validation should be applied, already at ground model level, in order to ensure that the specification reflects the user needs and statements about the system, and to detect faults in the specification as early as possible with limited effort.

ASMs model validation is possible by means of the model simulator **AsmetaS** [65], the model validator **AsmetaV** [46] and the model reviewer **AsmetaMA** tool [20].

The simulator **AsmetaS** allows to perform two type of simulations: *interactive simulation* and *random simulation*. The first asked to the user the values of parameters that depend on him/her, while the second itself randomly chooses the values for monitored functions (those that depends on the environment).

Model validator **AsmetaV** allows to build and execute *scenarios* of expected system behaviours. *Scenario-based validation* permits to automatize the simulation activity, so scenarios can be rerun after specification modifications. In scenario-

based validation the designer writes a scenario (using the textual notation `Avalla`) specifying the expected behavior of the model; scenarios are similar to test cases. The tool `AsmetaV` reads the scenario and executes it using the simulator `AsmetaS`. `Avalla` provides constructs to express scenarios as interaction sequences consisting of actions committed by the user to `set` the environment (i.e., the values of monitored/shared functions), to `check` the machine state, to ask for the execution of certain transition rules, and to enforce the machine itself to make one `step` (or a sequence of steps by command `step until`) as reaction to the user's actions. We wrote several scenarios for the different refinement steps. We discovered that such scenarios had several common parts, since they had to perform the same actions and same checks in different parts of their evolution. Therefore, we extended the validator with the possibility to define *blocks of actions* that can be reused in different scenarios: a block is a named sequence of `Avalla` commands delimited by keywords `begin` and `end`. A command block can be defined in any `Avalla` scenario and can be called by means of the command `execblock` in other parts of the same scenario or in other scenarios. A block can also be nested in another block.

A further validation technique is model review `AsmetaMA` (a form of static analysis) to determine if a model has sufficient *quality* attributes (as minimality, completeness, consistency). Common vulnerabilities and defects that can be introduced during ASMs modelling are checked as violations of suitable *meta-properties* (*MPs*, defined in [20] as CTL formulae). Meta-properties are devised in three categories:

- *consistency* guarantees that locations (memory units) are never simultaneously updated to different values;
- *completeness* requires that every behaviour of the system is explicitly modelled;
- *minimality* guarantees that the specification does not contain elements defined or declared in the model but never used.

The violation of a meta-property means that a quality attribute (*minimality*, *completeness*, *consistency*) is not guaranteed, and it may indicate the presence of an actual fault (i.e., the ASM is indeed faulty), or only of a *stylistic defect* (i.e., the ASM could be written in a better way).

Validation usually precedes the application of more expensive and accurate methods, like formal requirements analysis and verification of properties, that should be applied only when a designer has enough confidence that the specification captures all informal requirements. Formal verification of ASMs is possible by means of the model checker `AsmetaSMV` [19]. The tool translates `AsmetaL` models to models of the model checker `NuSMV`; so both *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulas can be proved.

When an actual code of the system implementation is available, either derived from the model as last low-level refinement step, or externally provided, also *conformance checking* is possible. Both *model-based testing* and *runtime verification* can be applied to check if the implementation conforms to its specification. `Asmeta` supports conformance checking w.r.t. Java code. The tool `ATGT` [64] can



be used to automatically generate tests from ASMs models<sup>2</sup> and, therefore, to check the conformance *offline*; CoMA [21], instead, can be used to perform runtime verification, i.e., to check the conformance *online*.

In MBT (Model Based Testing) [73, 129], *abstract test sequences* are derived from the specification; such sequences are then realized in tests for the implementation. In order to generate abstract test sequences, the tool used is ATGT [64]. The tool derives from the specification some test goals (called *test predicates*) according to some *coverage criteria* [63], and then generates sequences for covering these goals. For example, the *update rule coverage* criterion requires that each update rule is executed at least once in a test sequence and the update is not trivial (i.e., the new value is different from the current value of the location). The tool uses a classical approach based on model checking for generating tests. The ASMs model is translated in the language of a model checker, and each test goal is expressed as a temporal property (called *trap property*); if the trap property is proved false, the returned counterexample is the *abstract test sequence* covering the test goal (and possibly also other test goals). In order to realize the abstract test sequences into tests for the implementation, a linking between the specification and the implementation is provided. In [21], a technique is proposed to do the linking using Java annotations:

- associate a Java class with the corresponding ASMs model (`@Asm`);
- associate the ASMs state with the Java state:
  - `@FieldToFunction` connects a Java field with an ASM controlled function;
  - `@MethodToFunction` connects a Java pure (i.e., returning a value but not modifying the object state) method with an ASM controlled function;
  - `@Monitored` connects a Java field with an ASM monitored function; such fields represent the inputs of the Java class that take their value from the environment (as monitored functions in ASMs).
- associate the ASM behavior with the Java object behavior; `@RunStep` is used to annotate methods whose execution corresponds to a step of the ASM model.

Given the mapping provided by the Java annotations, abstract test sequences are translated in JUnit tests following the technique described in [22].

---

## 4.2 Compliance of the ASMs process with regulations

In Sect. 4.2.1 and Sect. 4.2.2, it is shown how compliant the ASM-based process is w.r.t. the existing normative in the field of medical software, which activities of the IEC standard can be covered by the use of ASMs, which FDA principles are ensured by the use of ASMs, how the rigour of a formal method such as ASMs can improve the current normative, and what can not be captured by or is out of the scope of ASMs.

---

<sup>2</sup>Note that sequences generated by ATGT could be used to test programs written in any programming language.

### 4.2.1 Compliance of Abstract State Machines process with IEC 62304

Regarding step (5.1) of the IEC 62304 standard, ASMs can supply a precise iterative life cycle model based on model refinement. Life cycle procedures are modeling, validation, verification, and conformance checking, the last applicable also at the maintenance phase. Deliverables are given in terms of a sequence of refined models, each one equipped with validation and verification results. Traceability is given, at each refinement step, by the conformance relation between abstract and refined models. ASMs do not support activities peculiar to risk management, although ASMs tools can be used to predict possible risks by reasoning on models and checking incorrect behaviours or potential faults. Thus, there is not any evidence of a possible use of ASMs in the context of risk management.

Step (5.2) is not covered by the use of ASMs, since it consists of informal requirements definition. Complementary techniques for requirements gathering can be exploited. The use of ASMs, as well as of any other formal method, starts from the results of requirements elicitation. An example is the HMCS description in [100].

Steps (5.3 - 5.4) are covered by the continuous activity of modeling and verifying software requirements along the ASM process till the desired level of refinement, possibly to code level. Already at the ground level, software structure is captured, even if not completely, by the model signature (i.e., domains and functions defined on them), while software behaviour is specified by means of transition rules. Model refinement and decomposition can help to manage the complexity of systems and move from a global view of the system to a component (or unit) view. Design decisions and architectural choices are added along model refinement. The ASM process has no specific techniques for risk analysis required by these steps of the standards; however, several static and dynamic verification techniques can be used for this aim.

Steps (5.5 - 5.7) concern code and testing. Although in ASMs a code prototype could be obtained as a last model refinement step, usually we expect code to be developed by a vendor and implemented by the use of powerful programming techniques and languages. Thus, the ASM process does not fully cover these development steps. However, having executable models available, ASM techniques for conformance checking (model-based testing and runtime verification) are applicable.

Regarding step (5.8), if a device manufacturer adopts the ASM process, demonstration that software has been validated and verified is straightforward, since validation and verification are continuous activities along the process, and conformance checking is possible on the subsequent released versions of the software.

Step (6) concern maintenance process. Every time the model is changed, the ASM process steps should be executed to guarantee the conformance between the model and the executable code.

Regarding step (7), it is not considered in the ASM process.

### 4.2.2 Compliance of Abstract State Machines process with FDA principles

By proposing the ASM process for medical software development, following, a discuss how ASM V&V activities achieve the FDA principles is proposed.

(1) Using ASMs, requirements are specified and documented by means of a chain of models providing a *rigorous baseline for both validation and verification*.

(2) Continuous *defect prevention* is supported. At each modelling level, faults and unsafe situations can be checked. Safety properties are proved on models, while software testing for conformance verification of the implementation is possible.

(3)-(6) The ASM process allows *preparation for software validation and verification* as early as possible, since V&V can start at ground level. These activities are *part of* the process, can be *planned* at different abstract levels, are *documented*, and supported by precise *procedures*, i.e., methods and techniques.

(7) In case changes only regard the software implementation and do not affect the model, our process requires to re-run conformance checking only; in case a software change requires to review the specification at a certain level, then refinement correctness must be re-proved and V&V re-executed from the concerned level down to the implementation.

(8) Regarding *validation coverage*, by simulation and testing, we can collect the coverage in terms of rules or code covered. This can be used by the designer to estimate if the validation activity is commensurate with the risk associated with the use of the software for the specified intended use.

(9) Since V&V are performed by exploiting unambiguous mathematical-based techniques, they facilitate *independent evaluation* of software quality assurance.

(10) The ASM process allows a *device manufacturer to demonstrate that the software has been validated and verified*: if an implementation is obtained as the last model refinement step, it is correct-by-construction due to the proof of refinement correctness; if the code has been developed by a vendor, conformance checking can guarantee correctness w.r.t. a verified model.



---

# CHAPTER 5

---

## Unified Syntax for ASM to Xtext

---

ASMs are a flexible, yet mathematically well-founded method and language for rigorous system engineering. The formalism can be seen as “pseudocode over abstract data”. Although this pseudocode notation is formally defined, in practice many ways exist to encode algebraic concepts and many abbreviations can be used to improve model conciseness and readability. Among the different frameworks for the ASM method (like AsmL, ASM Workbench, ASMGofer, KIV), two of the main ones are **Asmeta** and **CoreASM**. These platforms provide industrial strength tools to specify, verify, simulate, and test ASMs models. However, they implement different dialects of the pseudocode notation and support slightly different extensions of the original definition. Thus, while the availability of multiple support platforms is obviously an advantage, it may also be confusing for new adopters of the method. Moreover, designers cannot share models among the tools (unless a translator or adapter is defined) and thus can not easily take advantage of each tool’s strengths. To overcome these limitations, the idea of a common syntax definition Unified Abstract State Machines (UASM) [18], driven by the community, open to any actors, has grown in the last two years. The main goals are: - provide a stable language kernel for ASMs languages that support the interoperability of tools designed and implemented using this language; - preserve various useful extensions of the different tools. Furthermore, on the one hand, the UASM language should be usable for communication with customers and non-experts, and, on the other hand, precise enough to allow automatic analysis (like type checking, property verification, etc.). The definition of this language started two years ago, but the language was not implemented. For this reason in this thesis, the implementation has been performed using Xtext<sup>1</sup>. **Xtext** is a frame-

---

<sup>1</sup><https://eclipse.org/Xtext/>

work for development of programming languages and domain-specific languages. It follows the Model-Driven Engineering paradigm and uses Metamodel-based approach. The syntax used to define the grammar is similar to the EBNF standard. Once the grammar is defined, it provides many utilities, including the editor (with syntax highlighting, auto-completion, auto-indentation, typechecker), the parser and the validator (for semantic checking). During the implementation in Xtext, we found some incompleteness in the language and some ambiguities have been resolved. The final implementation of UASM grammar in Xtext is shown in the next section, the definitions are followed by some examples<sup>2</sup>.

### 5.1 UASM Header

---

The UASM syntax is defined using EBNF (Extended Backus-Naur form) notation<sup>3</sup>, a metasyntax notation which can be used to express a context-free grammar and it is defined as international standard in ISO-14977. In the following, UASM grammar rules are shown by using the Xtext syntax.

UASM specifications must contain a single ASM structure definition and take the extension `.uasm`. The structure defined in Xtext is divided into three sections: an ID, an Header and a Body.

```
Asm: ('asm' | 'asmmodule') name=ID header=Header body=Body;
```

There exist two types of ASM files:

- **asm**: contains a complete ASM and must contain the command for the execution of the main rule (`'exec' IdRule`);

---

```
asm mainASM
...
rule rule1 =
...
exec rule1
```

---

- **asmmodule**: contains an ASM and must not define a main rule (`exec`). Thus, it is not executable by itself. Any initialization has to be done by using the `initially` keyword in the definition of a function. It is also possible to provide a rule that has to be called from the ASM that uses this module, but one cannot rely on this, since it is up to the user to do this or not. If one cannot avoid this, we recommend to introduce a flag that indicates if the module was properly initialized or not.

---

```
asmmodule moduleASM
...
rule ruleN =
...
```

---

---

<sup>2</sup>The examples can be found at <https://github.com/uasm/uasm-language-examples/tree/master/UNIBG>

<sup>3</sup>The UASM syntax can be found at <https://github.com/uasm/uasm-language-definition>

The ID is the name of the ASM and it is a terminal (literal symbols which may appear in the outputs of the production rules of a formal grammar, the final output string consists only of terminal symbols).

The **Header** includes the imported/exported parts of an ASM from/to another ASM and the imported libraries to manage the elements defined. Although the **Body** contains domains definition, functions definition, rules definition and the main rule ('**exec**' IdRule).

```
Header : {Header} directives+=UseDirective*
directives+=ImportDirective* directives+=ExportDirective*;

Body : {Body} (definitions+=Definition)*(execution=Execution)?;
Execution : ('exec') idRule=IdRule);

Definition returns Definition : (TypeDefinition |
FunctionDefinition | RuleDefinition);
```

Code 5.1 shows a simple example of an .uasm file. It is a counter that increment the value at each step. `counterASM` is the name of the ASM, `counter` is the function updated at each step and `increment` is the name of the main rule.

```
asm counterASM

controlled counter -> INTEGER initially 0

rule increment =
counter := counter + 1

exec increment
```

---

**Code 5.1:** *ASM structure*

## 5.2 Transition Rules Definition

---

In a given state, a transition rule of an ASM produces for each variable assignment an update set for some dynamic functions of the signature. Each rule is identified by the keyword **rule**, the name IdRule (ID terminal), the rule parameters (ParameterDefinition) and the rule definition.

```
RuleDefinition : {RuleDefinition} 'rule' idRule=ID
parameters=ParameterDefinition)? '=' rule=Rule;
```

A simple example of rule definition is shown in Code 5.2. The rule `computeSum` has two parameters `num` and `num2`, both `INTEGER`.

```
controlled sum -> INTEGER
rule computeSum (num in INTEGER , num2 in INTEGER) =
sum := num + num2
```

**Code 5.2:** *Rule definition*

There are different type of rules in UASM.

```
Rule ::= ParBlock | SeqBlock | SeqNext | CaseRule | ChooseRule |
ConditionalRule | ExtendRule | ForAllRule | ImportRule |
IterateRule | LetRule | WhileRule | UpdateRule | SkipRule
| CallRule | LocalRule | PrintRule;
```

Afterwards, the rules are shown.

**ParBlock rule**

Rule is a set of rules (two or more) executed in parallel. The rules can be inserted between '**par - endpar**' keywords or between braces **{}**. The functions values are updated at the end of the state.

```
ParBlock : {ParBlockRule} 'par' rules+=Rule+ 'endpar' | '{'
rules+=Rule+ '}';
```

---

```
rule parRule =
par
rule1
rule2
...
endpar
```

---

```
rule parRule =
{
rule1
rule2
...
}
```

---

**Code 5.3:** *ParBlock rule*

**SeqBlock and SeqNext rule**

Rule is a set of rules (two or more) executed sequentially. In case of **SeqBlock** the rules can be inserted between '**seq - endseq**' keywords or between brackets **[]**. Otherwise, in case of **SeqNext** the rules can be inserted between '**seq - endseq**' keywords and the rules next to the first are preceded by the keyword '**next**'. The functions values are updated when an update rule occurs.

```
SeqBlock : {SeqBlock} 'seq' rules+=Rule+ 'endseq' | '['
rules+=Rule+ ']';
SeqNext : {SeqNext} 'seq' rules+=Rule ('next' rules+=Rule)+
('endseq')?;
```

Code 5.4 and Code 5.5 show an example of sequential rule where rules “rule1, rule2, ...” are executed sequentially in the order they are written.



---

<pre>rule seqRule =   seq     rule1     rule2 ...   endseq</pre>	<pre>rule seqRule =   [     rule1     rule2 ...   ]</pre>
--	---

---

**Code 5.4:** *SeqBlock rule*

---

```
rule seqRule =
  seq
    rule1
  next rule2 ...
  endseq
```

---

**Code 5.5:** *SeqNext rule*

### Case Rule

Depending on the value assumed by the first `Term`, `rulesi` is executed when `Termsi` is equal to `Term`. `otherwiseRule` is executed if `Term` does not correspond to any `Termi`. If `otherwiseRule` is omitted it is assumed as `Skip` as default.

```
CaseRule : {CaseRule} 'case' caseTerm=Term 'of' (terms+=Term
': ' rules+=Rule)+ ('otherwise' otherwiseRule=Rule)?
'endcase';
```

Code 5.6 shows two simple application of case rule. Assuming that `x` is a number, depending on the value assumed by `x` the corresponding rule is executed. In case `x` is equal to 1 `rule1` is executed, in case `x` is equal to 2 `rule2` is executed, otherwise `rule3` is executed. In the second example, since `otherwise` term is omitted, in case `x` is different from 1 and 2 it is assumed that `skip` rule is executed.

---

<pre>controlled x -&gt; INTEGER  rule caseRule =   case x of     1: rule1     2: rule2     otherwise rule3   endcase</pre>	<pre>controlled x -&gt; INTEGER  rule caseRule =   case x of     1: rule1     2: rule2   endcase</pre>
--	--

---

**Code 5.6:** *Case Rule*

### Choose Rule

Choose rule executes `Rule` with `VariableTerm` (set of letters and numbers) chosen in `EnumerableTerm` (`Term` or `Domain`) with `Term` true. If none `VariableTerm`

satisfy Term, ifnone\_Rule is executed. If ifnone\_Rule is omitted it is assumed as Skip as default.

```
ChooseRule : {ChooseRule} 'choose' varTerm+=VariableTerm
'in' term+=EnumerableTerm (',' varTerm+=VariableTerm 'in'
term+=EnumerableTerm)* ('with' termWith=Term)? 'do' rule=Rule
('ifnone' ifnoneRule=Rule )? ('endchoose')?;
```

Code 5.7 shows three examples of choose rule. First example chooses x in the domain D2. If x satisfies the condition  $x < 5$ , rule1 is executed; otherwise if there are no x that satisfy the condition, rule2 is executed. The second example is like the first, but if there are no x that satisfy the condition, it is assumed as Skip as default. The third example chooses x in the interval 0-10 and y in the domain ExampleDomain (contains number in the interval 0-100). If the condition  $x \neq 5$  is satisfied the ParBlock rule is executed, otherwise Skip is executed.

---

```
rule chooseRule =
    choose x in D2 with x<5 do rule1 ifnone rule2 endchoose
```

---

```
rule chooseRule =
    choose x in D2 with x!=5 do rule1 endchoose
```

---

```
domain ExampleDomain of ( INTEGER ) initially {[0 .. 100]}
rule chooseRule =
    choose x in D2, y in ExampleDomain with x!=5 do
        par
            z := x;
            w := x + y;
        endpar
    endchoose
```

---

**Code 5.7:** Choose Rule

### Conditional Rule

Conditional rule executes then\_Rule if Term is true, otherwise else\_Rule is executed. If else\_Rule is omitted it is assumed as Skip as default.

```
ConditionalRule : 'if' condition=Term 'then' thenRule=Rule (=>
'else' elseRule=Rule)? 'endif'?
```

Code 5.8 shows an example where rule1 is executed if  $x > 0$  is true, otherwise rule2 is executed. In the second example if the condition is false, Skip rule is executed.

---

```

controlled x -> INTEGER
rule conditionalRule =
  if x>0 then
    rule1
  else
    rule2
  endif

```

---

```

controlled x -> INTEGER
rule conditionalRule =
  if x>0 then
    rule1
  endif

```

---

**Code 5.8:** *Conditional Rule***Skip Rule**

Skip rule does nothing. This rule is used when the machine does not perform any action

```
SkipRule : {SkipRule} 'skip';
```

**Update Rule**

LocationTerm (function previously declared) is updated with Term value.

```
UpdateRule : {UpdateRule} location=LocationTerm ':=' term=Term;
```

An example is shown in Code 5.2 where function `sum` is updated with the result of `num + num2`.

**Extend Rule**

Extend Rule extends an `ExtendableDomain` with one or more `VariableTerm` and then Rule is executed. The set of `VariableTerm` can be assigned to an `as_VariableTerm` and used as `as_VariableTerm` in Rule execution.

```
ExtendRule : {ExtendRule} 'extend' extendDomain=ExtendableDomain
'with' varTerm+=VariableTerm (',' varTerm+=VariableTerm)* ('as'
varTermOpt=VariableTerm)? 'do' rule=Rule 'endextend'?;
```

In Code 5.9 an extendable domain `userDomain` is extended with the values 5,6 and 7 and then `rule1` is executed. In the second example, the values 5,6 and 7 are assigned to the `VariableTerm NewVar` and it is passed to `rule1` as parameter.

---

```
rule extendRule =
  extend userDomain with 5,6,7 do rule1 endextend
```

---

```
rule extendRule =
  extend userDomain with 5,6,7 as NewVar do rule1(NewVar)
endextend
```

---

**Code 5.9:** *Extend Rule*

### ForAll Rule

ForAll rule executes Rule with VariableTerm chosen in EnumerableTerm with Term true. If none VariableTerm satisfy Term, ifnone\_Rule is executed. If ifnone\_Rule is omitted it is assumed as Skip as default.

```
ForAllRule : {ForAllRule} 'forall' varTerm+=VariableTerm
'in' term+=EnumerableTerm (',' varTerm+=VariableTerm 'in'
term+=EnumerableTerm)* ('with' termWith=Term)? 'do' rule=Rule
('ifnone' ifnoneRule=Rule)? 'endforall'?;
```

In the examples shown in Code 5.10, rule1 is executed forall x in the domain D2 if the condition x<5 is true. If there are no x that satisfy the condition rule2 is executed. In the second example if the condition is false forall x, Skip rule is executed.

---

```
rule forAllRule =
  forall x in D2 with x<5 do rule1 ifnone rule2 endforall
```

---

```
rule forAllRule =
  forall x in [0..10] with x<5 do rule1 endforall
```

---

**Code 5.10:** *ForAll Rule*

### Import Rule

Import rule imports VariableTerm and executes Rule using the imported terms.

```
ImportRule : {ImportRule} 'import' varTerm=VariableTerm 'do'
rule=Rule 'endimport'?;
```

In the example shown in Code 5.11

```
rule importRule =  
  import x do rule1(x) endimport
```

---

Code 5.11: *Import Rule*

### Iterate Rule

Iterate Rule executes Rule until the update set produced is either empty or inconsistent.

```
IterateRule : {IterateRule} 'iterate' rule=Rule 'enditerate'?
```

Code 5.12 execute rule1 until the update set is empty.

```
rule iterateRule =  
  iterate rule1 enditerate
```

---

Code 5.12: *Iterate Rule*

### Let Rule

Let Rule assigns Term to VariableTerm and then executes Rule which contains occurrences of VariableTerm.

```
LetRule : {LetRule} 'let' varTerm+=VariableTerm '=' term+=Term  
(',' varTerm+=VariableTerm '=' term+=Term)* 'in' rule=Rule  
'endlet'?
```

In Code 5.13, rule1 is executed with two parameters which value is set in rule let.

```
rule letRule =  
  let (x=5, y=8) in rule1(x,y) endlet
```

---

Code 5.13: *Let Rule*

### While Rule

While Rule executes Rule while Term is true.

```
WhileRule : {WhileRule} 'while' term=Term 'do' rule=Rule  
'endwhile'?
```

Code 5.14 executes  $x:=x-1$  while  $x$  is bigger than  $y$ .

```
rule whileRule =
  while x>y do x:=x-1 endwhile
```

---

Code 5.14: *While Rule*

### Call Rule

Call Rule calls rule with name `IdRule` (declared in the ASM) and passes parameters (`Term`) if they exist.

```
CallRule : {CallRule} calledRule=IdRule ('(' term+=Term (','
term+=Term)* ')')? )?
```

In Code 5.15, `rule1` (the rule called) is executed if `y>0`.

```
rule rule1 (x in INTEGER) = skip

rule callRule =
  if (y>0) then rule1(y) endif
```

---

Code 5.15: *Call Rule*

### Local Rule

Local Rule defines a local function with ID name, the set of parameters (`ParameterDefinition`), the function Domain and the initial value of the function (`InitialFunctionDefinition`). Rule is executed considering this new declared function.

```
LocalRule : 'local' id+=ID paramdef+=ParameterDefinition?
(('->' | '→') domain+=Domain)? ('initially'
initialDefinition+=InitialFunctionDefinition)? (',' id+=ID
paramdef+=ParameterDefinition? (('->' | '→') domain+=Domain)?
('initially' initialDefinition+=InitialFunctionDefinition)?*
'in' rule=Rule 'endlocal'?;
```

In the example shown in Code 5.16, the local function `x` is declared and used as parameter in `rule1`.

```
...
local x -> INTEGER initially 0 in rule1(x) endlocal
...
```

---

Code 5.16: *Local Rule*

### Print Rule

Print Rule prints out `Term` to the environment.

```
PrintRule : 'print' term=Term
```

Code 5.17 print the value of function x.

```
...
print x
...
```

Code 5.17: *Print Rule*

## 5.3 Type Definition

Type Definition contains the definition of user-named domains: type-domains (`DomainDefinition`) and enumerative domains (`EnumerateDefinition`).

```
TypeDefinition returns TypeDefinition : DomainDefinition |
EnumerateDefinition;
```

### Domains

At the root of enumerative domains and type-domains, there are the following individual components: structured domains, basic domains and extendable domains.

```
Domain : StructuredDomain | BasicDomain | ExtendableDomain;
```

Structure domains are used for building data structures like sets, sequences, bags, maps. Basic domains are reals, integers, strings and extendable domains are a subset of another user-named domain, generic domain or sub-domains of the Agent type-domain.

```
StructuredDomain : {StructuredDomain} type='SET'
('domainSet=Domain')? | type='BAG'
('domainBag=Domain')? | type='LIST'
('domainList=Domain')? | type='MAP' ('domainMap+=Domain',
'domainMap+=Domain')?;
ExtendableDomain : {ExtendableDomain} type='ANY' |
type='AGENT' | type='IdDomain';
BasicDomain : {BasicDomain} typeBasicDom=BasicDomainEnum ;
enum BasicDomainEnum: number='NUMBER' | integer='INTEGER'
| string='STRING' | char='CHAR' | boolean='BOOLEAN' |
rule='RULE';
```

### Enumerative Domain Definition

Enumerative Domain is a domain ID with a defined set of elements `EnumTerm`. Enumerative domain are dynamic and during the ASM execution new elements can be added.

```
EnumerateDefinition : {EnumerateDefinition} 'enum' name=ID '='
'{' enums+=EnumTerm (',' enums+=EnumTerm)* '}';
```

The example in Code 5.18 shows the definition of the enumerative domain `ColorDomain`. At first the values contained in the domain are `RED`, `BLUE` and `GREEN`. In rule `extendDomain` the `ColorDomain` is extended with two values, `YELLOW` and `BLACK`.

```
enum ColorDomain = {RED, BLUE, GREEN}
...
rule extendDomain=
    extend ColorDomain with YELLOW, BLACK do rule1 endextend
...
```

---

**Code 5.18:** *Enumerative Domain*

### Domain Definition

Domain definition defines new user-named domains. `IdDomain` (defined as set of letters and numbers) is the name of the user-named domain, `DomainParameterDefinition` identifies the structure of the elements of the declared domain and `InitialDomainDefinition` is the initial set of values in the defined domain. All domains defined by the user are extendable, new elements can be imported from a possibly infinite reserve by means of extend rules.

The domains can be divided into two groups: tuple domains (domains with more than one field) or single domains (domain with only one field).

```
DomainDefinition : {DomainDefinition} 'domain' name=ID ('of'
domainParamDef=DomainParameterDefinition)?
(initialDefition=InitialDomainDefinition)?;
DomainParameterDefinition : {DomainParameterDefinition} Domain
| ParameterDefinition;
ParameterDefinition : {ParameterDefinition} '(' (idIn=ID ('in'
| '∈') domainIn=Domain | id=ID | domain=Domain) (',' (idIn=ID
('in' | '∈') domainIn=Domain | id=ID | domain=Domain))* ');'
```

In UASM user-named domains are all dynamic, this means that can be extended during the ASM execution. If the values of domains are known a priori, they can be defined in the `InitialDomainDefinition` section. The initialization can be both domain and tuple domain using respectively `Literal` and `TupleLiteral` (records of `Literal`).



```

InitialDomainDefinition : {InitialDomainDefinition}
'initially' '{' (tuple+=TupleLiteral | literals+=Literal) (','
(tuple+=TupleLiteral | literals+=Literal))* '};'
TupleLiteral : {TupleLiteral} '(' literals+=Literal (','
literals+=Literal)+ ')';
Literal : NumberLiteral | BooleanLiteral | Kernelliteral |
StringLiteral | CharLiteral | EnumTerm;

```

Some examples are shown in Code 5.19. `PlayingCards` is a subset of the `INTEGER` domain and identifies the number of one card deck. `BOOK` domain identify a book, it has three fields: the author, the year and the publisher. Initially the domain contains two entities. `listDom` is a domain which elements belong to a list of strings; `Red`, `Yellow` and `Blue` are the initial values.

```

domain PlayingCards of INTEGER initially [0..40]
domain BOOK of (author in STRING, year in INTEGER, publisher in STRING)
initially {"Angelo",2002,"Springer"},{"Elvinia",2010,"Elsevier"}
domain listDom of LIST(STRING) initially ["Red", "Yellow", "Blue"]

```

Code 5.19: *Domains*

## 5.4 Functions definition

There exist six type of functions: controlled, static, derived, monitored, shared and out. Each function may have domain (`Domain`, before declared), set of parameters (`ParameterDefinition`) and initial definition (`InitialFunctionDefinition`); all these elements are not mandatory for each type of function. Only the function type and function name (`IdFunction`) are mandatory in function definition.

```

FunctionDefinition returns FunctionDefinition :
(ControlledFunction | StaticFunction | DerivedFunction |
MonitoredFunction | SharedFunction | OutFunction);

```

### Controlled Function

Controlled functions are read and written by the machine.

```

ControlledFunction : {ControlledFunction} ('controlled'
|('function'|('controlled' ('function') name=IdFunction
from=ParameterDefinition? ((('->' | ('->') to=Domain)?
(('initially' initialDef=InitialFunctionDefinition)?);

```

### Static Function

Static functions do not change the value depending on the current state. Static function must be initialize (`InitialFunctionDefinition`) with the value always assumed during ASM execution.

```
StaticFunction : {StaticFunction} 'static' 'function'?  
name=IdFunction from=ParameterDefinition? (('->' | ('→'  
to=Domain)? ('always' initialDef=InitialFunctionDefinition);
```

### Derived Function

Derived functions return value that depends on the current value of its inputs. The value assumed by the function is defined in `InitialFunctionDefinition`.

```
DerivedFunction : {DerivedFunction} 'derived' 'function'?  
name=IdFunction from=ParameterDefinition? (('->' | ('→'  
to=Domain)? ('=' initialDef=InitialFunctionDefinition);
```

### Monitored Function

Monitored functions are read by the machine and written by the environment. There is not `InitialFunctionDefinition` because the value is read externally at each state.

```
MonitoredFunction : {MonitoredFunction} 'monitored'  
'function'? name=IdFunction from=ParameterDefinition? (('->'  
| ('→') to=Domain)?;
```

### Shared Function

Shared functions are read and written by the machine and the environment.

```
SharedFunction : {SharedFunction} 'shared'  
'function'? name=IdFunction from=ParameterDefinition?  
(('->' | ('→') to=Domain)? ('initially'  
initialDef=InitialFunctionDefinition);
```

### Out Function

Out functions are written by the machine.

```
OutFunction : {OutFunction} 'out' 'function'? name=IdFunction  
from=ParameterDefinition? (('->' | ('→') to=Domain)?  
('initially' initialDef=InitialFunctionDefinition);
```

### Parameter Definition

Parameter definition contains the definition of the function parameters. It can be a single parameter or a set of parameters (see domain definition in Sect. 5.3). Each parameter can be identified by its `Domain`, its name (ID) with a generic domain (any type) or its name with a specific `Domain`.

---

### Initial Function Definition

Initial function definition identifies the values assumed always by static functions, the definition of derived function or the value initially assumed by controlled, shared and out functions.

```
InitialFunctionDefinition : 'from'? term=Term;
```

An example of functions definition and initialization is shown in Code 5.20.

---

```

controlled counter -> INTEGER initially 5
static binary -> INTEGER always 2
derived functionder (x in INTEGER)-> BOOLEAN =
    if x > 0 then
        true
    else
        false
    endif
shared commonfunction -> BOOLEAN initially false
out printResult -> STRING initially "Undefined result"
monitored currentTemperature -> INTEGER

```

---

**Code 5.20:** *Functions definition*

The controlled function `counter` is an integer function and assumes value 5 as initial value. `binary` is an integer static function, this means that the value assumed is always 2 during the execution of ASM. The value assumed by the derived function `functionder` is always a boolean and depends on the current value of `x`; if `x>0` `functionder` is equal to `true`, otherwise is equal to `false`. The boolean shared function `commonfunction` and the string out function `printResult` are initialized to two different values which can be changed during the machine execution. In case of monitored function, initialization is not provided, only the domain can be defined.

## 5.5 Terms

---

Terms are nested expressions which can be evaluated in a state of an ASM. The Terms are recursive expression and can be easily represented using EBNF grammar. Considering Xtext, it does not support left recursion. Left recursion is avoided by adding a return value to each rule. By following the same principle, Xtext allows the definition of the precedence in the operators. In this case the precedence followed is the same as proposed by Java.

```

Term returns Expression : CondTernaryExpression;
CondTernaryExpression returns Expression :
ImpliesExpression ({CondTernaryExpression.cond=current} '?'
then=ImpliesExpression ':' else=ImpliesExpression)*;
ImpliesExpression returns Expression : BitwiseOrExpression
({BinaryExpression.left=current} op=('implies')
right=BitwiseOrExpression)*;
BitwiseOrExpression returns Expression : BitwiseXorExpression
({BinaryExpression.left=current} op=('or')
right=BitwiseXorExpression)*;
BitwiseXorExpression returns Expression : BitwiseAndExpression
({BinaryExpression.left=current} op=('xor')
right=BitwiseAndExpression)*;
BitwiseAndExpression returns Expression : EqualityExpression
({BinaryExpression.left=current} op=('and')
right=EqualityExpression)*;
EqualityExpression returns Expression : RelationExpression
({BinaryExpression.left=current} op=('!='|'=')
right=RelationExpression)*;
RelationExpression returns Expression :
AddExpression ({BinaryExpression.left=current}
op('<'|>'|<='|>='|'memberof') right=AddExpression)*;
AddExpression returns Expression : MultExpression
({BinaryExpression.left=current} op('='|'-')
right=MultExpression)*;
MultExpression returns Expression : PowerExpression
({BinaryExpression.left=current} op('*'|'/'|'mod'|'div')
right=PowerExpression)*;
PowerExpression returns Expression : UnaryExpression
({BinaryExpression.left=current} op('^')
right=UnaryExpression)*;

```

```

UnaryExpression returns Expression : {BooleanNegation} =>
op='not' expression=BasicExpression | {ArithmeticSigned} =>
op='- ' expression=BasicExpression | BasicExpression;

```

### Basic Expression

A basic expression can be a BasicTerm or again a Term.

```

BasicExpression returns Expression : {BasicTermExpr} =>
basicT=BasicTerm | {ParenthesisTermExpr} => '(' term=Term ')';

```

There is a set of BasicTerm in UASM:

```
BasicTerm : LocationTerm | ComprehensionTerm | StructureTerm
| PickTerm | ConditionalTerm | CaseTerm | RuleAsTerm |
ReturnTerm | ForAllTerm | ExistsTerm | LetTerm | Literal |
SizeOfEnumerableTerm
```

### Location Term

Location Term is a specialized function term where `FunctionTerm` is a dynamic function fixed in the ASM definition.

```
LocationTerm : {LocationTerm} function = FunctionTerm | result
= 'result';
```

### Function Term

Function term is a function with `IdFunction` name defined in the `Function-Definition` section. `Term` represents the set of the actual parameters of the function. If the arity of the function is 0, there is no `Term`.

```
FunctionTerm : {FunctionTerm} function=IdFunction ('('
args=Term (',' argsTuple=Term)* ')')?;
```

An example of function term is shown in Code 5.21. The definition of the derived function `functionder` depends on the value assumed by the derived function `addValue`. The element `addValue(y)` in `functionder` is the `FunctionTerm`.

```
derived addValue (x in INTEGER) -> INTEGER =
    x+100
derived functionder (y in INTEGER)-> BOOLEAN =
    if addValue(y) > 0 then
        true
    else
        false
    endif
```

Code 5.21: *Functions definition*

### Structure Term

Structure term identifies four collections:

```
StructureTerm : SetTerm | ListTerm | BagTerm | MapTerm;
```

- Set Term: unordered set of unduplicated `Term` of the same nature

```
SetTerm : {SetTerm} '{' Term (',' Term)* '}' | '{}'
```

- List Term: list of ordered Term of the same nature (duplicated allowed)

```
ListTerm : {ListTerm} '[' Term (',' Term)* ']' | '[' ]';
```

- Bag Term: unordered set of Term of the same nature (duplicated allowed)

```
BagTerm : BagTerm '<' Term (',' Term)* '>' | '<' '>';
```

- Map Term: represents a mapping between an unduplicated key (first Term) and a value (second Term)

```
MapTerm : MapTerm '{' (Term ('->' | '→') Term) (',' Term ('->' | '→') Term)* '}' | '{' ('->' | '→') '};'
```

Code 5.22 shows an example of Set, List, Bag and Map terms.

---

```
Set: {1,8,6,10}
List: [1,6,8,8,9]
Bag: <1,9,5,7,5,6>
Map: {1->5, 3->8, 8->4, 10->8}
```

---

**Code 5.22:** *Structure Term*

### Comprehension Term

Comprehension term (Set, List, Map, Bag or Range) consists of elements that breach of a condition.

```
ComprehensionTerm : SetComprehensionTerm |
ListComprehensionTerm | MapComprehensionTerm |
BagComprehensionTerm | NumberRangeTerm;
```

- Set Comprehension Term: identifies a set of Term of the same nature, such that VariableTerm takes value in EnumerableTerm and WithTerm is satisfied (if WithTerm is omitted it is assumed true).

```
SetComprehensionTerm : {SetComprehensionTerm} '{'
term=Term '|' varTerm+=VariableTerm ('in' | '∈')
enumTerm+=EnumerableTerm (',' varTerm+=VariableTerm
('in' | '∈') enumTerm+=EnumerableTerm)* (('with' | ':')
termWith=Term)? '};'
```

The example in Code 5.23 initializes the domain `setDomain` with values `a + b` that satisfy the condition `a > 5` and `b > a`.

---

```

domain limitDomain of INTEGER initially {0..10}
domain setDomain of SET(INTEGER) initially
  { a + b | a in limitDomain, b in limitDomain with a > 5 and b > a}

```

---

Code 5.23: *Set Comprehension Term*

- List Comprehension Term : identifies a list of Term of the same nature, such that VariableTerm takes value in EnumerableTerm and WithTerm is satisfied (if WithTerm is omitted it is assumed true).

```

ListComprehensionTerm : {ListComprehensionTerm} '['
term=Term '|' varTerm+=VariableTerm ('in' | '∈')
enumTerm+=EnumerableTerm (',' varTerm+=VariableTerm
('in' | '∈') enumTerm+=EnumerableTerm)* (('with' | ':')
termWith=Term)? ']' ;

```

In Code 5.24 the domain listDomain is initialized with the result of double(a) that satisfy the condition  $a > b$ .

---

```

domain limitDomain of INTEGER initially {0..10}
domain listDomain of LIST(INTEGER) initially [double(a) | a in limitDomain,
      b in limitDomain with a > b]
derived double (a in limitDomain) -> INTEGER = ...

```

---

Code 5.24: *List Comprehension Term*

- Map Comprehension Term: identifies a map between two Terms (Term  $\rightarrow$  Term) of the same nature, such that VariableTerm takes value in EnumerableTerm and WithTerm is satisfied (if WithTerm is omitted it is assumed true).

```

MapComprehensionTerm : {MapComprehensionTerm} '{' term1=Term
('->' | '→') term2=Term '|' varTerm+=VariableTerm ('in' |
'∈') enumTerm+=EnumerableTerm (',' varTerm+=VariableTerm
('in' | '∈') enumTerm+=EnumerableTerm)* (('with' | ':')
termWith=Term)? '}' ;

```

The example in Code 5.25 shows the initialization of mapDomain. The elements are pairs of number (key, value) that satisfy the condition  $a > 5$  and  $b = 8$ .

```

domain limitDomain of INTEGER initially {0..10}
domain mapDomain of MAP(INTEGER -> INTEGER) initially
  {a -> b | a in limitDomain, b in limitDomain with a > 5 and b = 8}

```

---

**Code 5.25:** *Map Comprehension Term*

- Bag Comprehension Term: identifies a list of Term of the same nature (also duplicated are admitted), such that VariableTerm takes value in EnumerableTerm and WithTerm is satisfied (if WithTerm is omitted it is assumed true).

```

BagComprehensionTerm : {BagComprehensionTerm} '<'
term=Term '|' varTerm+=VariableTerm ('in' | '∈')
enumTerm+=EnumerableTerm (',' varTerm+=VariableTerm
('in' | '∈') enumTerm+=EnumerableTerm)* (('with' | ':')
termWith=Term)? '>';

```

The domain bagDomain in Code 5.26 is initialized with the values  $a + b * 2$  where the condition  $a = 5$  is true.

```

domain limitDomain of INTEGER initially {0..10}
domain bagDomain of BAG(INTEGER) initially
  < a + b * 2 | a in limitDomain, b in limitDomain with a = 5 >

```

---

**Code 5.26:** *Bag Comprehension Term*

- Number Range Term: identifies a list of Terms that starts from first Term until second Term. If Step\_Term is specified, the values chosen are from first Term until second Term with a constant step.

```

NumberRangeTerm : {NumberRangeTerm} '[' start=Term '..'
end=Term ('step' step=Term)?; ']'

```

The example in Code 5.27 shows the initialization of domain range1 with the values between 0 and 10, while range2 is initialized with even numbers from 0 to 10.

```

domain range1 of INTEGER initially [0..10] //0,1,2,3,4,5,6,7,8,9,10
domain range2 of INTEGER initially [0..10] step 2 //0,2,4,6,8,10

```

---

**Code 5.27:** *Number Range Term*



**Enumerable Term** : can be a Domain or again a Term.

```
EnumerableTerm : Term | dom=Domain;
```

### Pick Term

Pick term chooses a VariableTerm in EnumerableTerm with With\_Term true if it is specified. If With\_Term is not provided it is skip as default.

```
PickTerm : 'pick' varTerm=VariableTerm ('in' | '∈')
enumTerm=EnumerableTerm ('with' term=Term)?;
```

The instruction pick in Code 5.28 takes a value x that belongs to limitDomain if the condition double(x)>10 is true.

```
domain limitDomain of INTEGER initially {0..10}
derived double(a in limitDomain) -> INTEGER = ...
...
pick x in limitDomain with double(x)>10
...
```

**Code 5.28:** *Number Range Term*

### Conditional Term

In conditional term condition, then clause and else clause are mandatory. The term can be written in abbreviated or extended form.

```
ConditionalTerm : 'if' cond=Term 'then' thenTerm=Term 'else'
elseTerm=Term;
```

The conditional term is used in the example in Code 5.29 to define the value assumed by the derived function counterDer.

```
controlled positive -> BOOLEAN initially false
controlled counter -> INTEGER initially 0
derived counterDer -> INTEGER =
  if counter>0 then
    positive := true
  else
    positive := false
```

**Code 5.29:** *Number Range Term*

### Case Term

In case term the corresponding Term is returned, depending on the value assumed by Case\_Term. If no Term has been found, the otherwise Term is returned.

```
CaseTerm : 'case' caseTerm=Term 'of' (term+=Term ':'  
termAction+=Term )+ ( 'otherwise' otherwiseTerm=Term )?  
'endcase';
```

Code 5.30 uses case term to determine the value of the derived function `result`. Depending on the value assumed by `winner` function the value of `result` function is: "WinFirst" if `winner` is 1, "WinSecond" if `winner` is 2, "Draw" otherwise.

---

```
controlled winner -> INTEGER initially 0  
derived result -> STRING =  
  case winner of  
    1: "WinFirst"  
    2: "WinSecond"  
    otherwise: "Draw"  
  endcase
```

---

Code 5.30: *Number Range Term*

### Rule As Term

The rule with id `IdRule` is used as a term when preceded by `@` symbol.

```
RuleAsTerm : {RuleAsTerm} '@' rule=IdRule;
```

### Return Term

The return term returns the `Term` defined in `Rule`.

```
ReturnTerm : {ReturnTerm} 'return' term=Term 'in' rule=Rule;
```

An example is shown in Code 5.31. The term `r` defined in the conditional rule, is assigned to the derived function `functionrecursive`.

---

```
derived functionrecursive(x) =  
  return r in  
    if x < 0 then r := 0  
      else if x < 2 then r := x  
        else r := functionrecursive(x-2) + functionrecursive(x-1)
```

---

Code 5.31: *Number Range Term*

### Let Term

Let term allows to assign a set of `VariableTerm` to a single `Term` and use them simply calling the `Term`.

```
LetTerm : {LetTerm} 'let' varTerm+=VariableTerm '=' term+=Term
(',' varTerm+=VariableTerm '=' term+=Term)* 'in' body=Term;
```

The example in Code 5.32 shows the definition of the derived function `AverageAreaRectangles`. The function computes the average area among two rectangles given the bases and the heights. The areas of rectangles are assigned to two `VariableTerms`, `a1` and `a2`, subsequently the average is computed.

```
derived AverageAreaRectangles (h1 in INTEGER, b1 in INTEGER, h2 in INTEGER,
                               b2 in INTEGER) -> NUMBER =
  let (a1=(h1*b1)/2, a2=(h2*b2)/2) in (a1+a2)/2
```

**Code 5.32:** *Number Range Term*

### Exists Term

Exists term verifies if a `Term` or a set of terms exist (`unique`) and fulfil the condition `'with'` `Term`. The term return a boolean value.

```
ExistsTerm : {ExistsTerm} (('exists' unique='unique') |
'∃' | '∃!') varTerm+=VariableTerm ('in' | '∈') inTerm+=Term
(',' varTerm+=VariableTerm ('in' | '∈') inTerm+=Term)* 'with'
withTerm=Term;
```

Code 5.33 shows an example of exists term. The derived function `existUserColour` returns true if the `userColour` is contained in `SubsetColour` domain, false otherwise.

```
enum SubsetColour = {RED, YELLOW, BLACK}
enum Colour = {RED, YELLOW, BLACK, BROWN, PINK, WHITE, BLUE, ORANGE, GREEN}
derived existUserColour (userColour in Colour) -> BOOLEAN =
  exist a in SubsetColour with a=userColour
```

**Code 5.33:** *Number Range Term*

### Size Of Enumerable Term

Size of Enumerable Term returns the size of the `EnumerableTerm`. The condition of usage is that `EnumerableTerm` is a finite collection of elements.

```
SizeOfEnumerableTerm : SizeOfEnumerableTerm '|'
enumTerm=EnumerableTerm '|';
```

In Code 5.34, the derived function `numberOfSubsetColour` is equal to the number of elements in the enumerable domain `SubsetColour`. The value of the derived function changes because the enumerable domain `SubsetColour` is dynamic and new elements can be added.

```
enum SubsetColour = {RED, YELLOW, BLACK}
derived numberOfSubsetColour -> INTEGER = | SubsetColour |
```

Code 5.34: *Number Range Term*

## 5.6 Header

Header identifies which parts are imported or exported by the ASM. `UseDirective` loads modules (plugins, not other ASMs) with name ID. `ImportDirective` imports in the current ASM domains, functions and/or rules from other ASMs. `ExportDirective` identifies which domains, functions and/or rules are exported from current ASM to other ASMs.

Afterwards, the definition of basic elements is shown.

```
Header : {Header} UseDirective* ImportDirective*
ExportDirective*;
UseDirective : {UseDirective} 'use' id=ID;
ImportDirective : {ImportDirective} 'import' id+=ID ( '(' (
IdDomain | IdFunction | IdRule ) ( ',' ( IdDomain | IdFunction
| IdRule ) ) * ')' )?;
ExportDirective : ExportDirective 'export' id+=ID ( '(' (
IdDomain | IdFunction | IdRule ) ( ',' ( IdDomain | IdFunction
| IdRule ) ) * ')' | '*' )?;
```

## 5.7 Terminals

Terminals are the elementary symbols of UASM. They are String, Char, Number (integer and real) and IDs (sequence of letters and numbers).

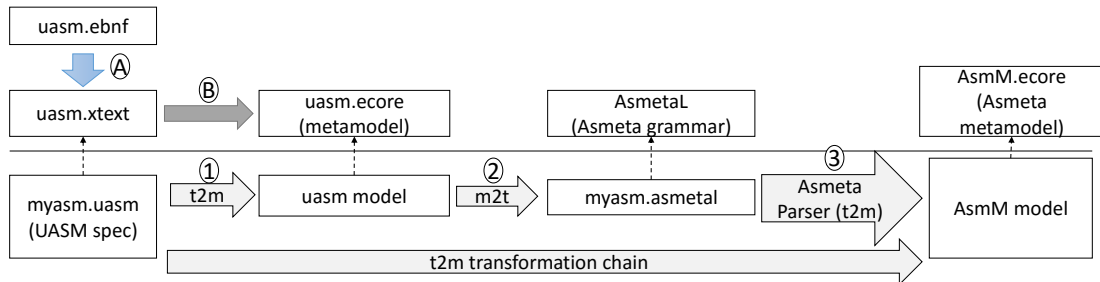
```
terminal ID : '^'? ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' |
'A'..'Z' | '0'..'9')*;
terminal INT returns ecore:EInt: ('0'..'9')+;
String ::= '"' ('\\' | !('\\' | '"')) * '"' | "'" ('\\' |
!('\\' | "')) * "'"
```

Following are shown the definition of the remain elements.

```
BooleanLiteral : {BooleanLiteral} val='true' | val='false'
KernelLiteral : {KernelLiteral} val='undef' | val='self'
NumberLiteral : {NumberLiteral} value=INT | valueDec=TK_FLOAT;
EnumTerm : ID;
IdDomain : ID;
IdFunction : ID;
IdRule : ID;
```

## 5.8 Implementation: from UASM to Asmeta

Currently only the parser and the editor are available for UASM grammar. **Asmeta** framework provides tools (see Sect. 4.1) for validation and verification, but they manipulate **AsmM** models. So, in order to use **Asmeta** tools on UASM specifications, the UASM specifications must to be mapped to **AsmM** models.



**Figure 5.1:** From UASM specification to AsmetaL specification

The process adopted is shown in Figure 5.1. The Xtext grammar (see from Section 5.1 to Section 5.7) has been derived starting from UASM EBNF grammar definition (step A). From Xtext grammar a parser and a metamodel have been obtained automatically (step B). The metamodel is in the form of an EMF (Eclipse Modelling Framework) model in the ecore format and Java APIs are automatically obtained to manage UASM models. Furthermore, an editor with syntax colour highlighting, auto completion and outline view helps the user in writing the specifications.

Given the UASM specification, the transformation to an **AsmM** model (bottom of Figure 5.1) consists in the following steps:

- (step 1) parse the specification with the parser: the parser is automatically generated by Xtext.
- (step 2) produce an **AsmetaL** specification from the Ecore objects derived by Xtext. These objects are iteratively analysed and, based on the type, they are translated to the target language. The translation is not automatically defined by Xtext, but it is defined by the user through a set of translation rules. The translation rules can be defined using Xtend (a typed programming language sitting on top of Java) or Java classes.
- (step 3) obtain an **AsmM** model with the **AsmetaL** parser.

UASM and **AsmetaL** are syntactically similar, but they have some differences<sup>4</sup>.

In Section 5.8.1, an example of translation from UASM to **AsmetaL** is shown. It is a didactic example, but it helps to understand some differences between UASM and **AsmetaL**.

### 5.8.1 Coffee Vending Machine

The machine distributes coffee, tea and milk. The machine accepts only 50 cents and 1 euro. If the user inserts 50 cents the machine distributes milk (if it is

<sup>4</sup>The EBNF definition of **AsmetaL** can be found at [http://fmse.di.unimi.it/asmata/download/AsmetaL\\_EBNF.html](http://fmse.di.unimi.it/asmata/download/AsmetaL_EBNF.html).

available); if the user inserts 1 euro the machine distributes randomly coffee or tea (if they are available). If a drink is distributed his availability is decremented and the money is preserved into the machine. Each step matches with the money insertion. At the beginning the machine has 10 coffee, 10 tea and 10 milk. The machine can contain 25 money maximum. The user decides the money to be insert at each step.

The UASM specification (see Code 5.35) is manually written, while the `AsmetaL` specification (see Code 5.36) is automatically derived using the tool.

---

```
asm coffeeVendingMachine

enum CoinType = { HALF , ONE }
enum Product = { COFFEE , TEA , MILK }
domain QuantityDomain of ( INTEGER ) initially { [ 0 .. 10 ] }
domain CoinDomain of ( INTEGER ) initially { [ 0 .. 25 ] }
controlled available ( Product ) -> QuantityDomain initially 10
controlled coins -> CoinDomain initially 0
monitored insertedCoin -> CoinType

rule r_serveProduct ( p in Product ) =
{
    available ( p ) := available ( p ) - 1
    coins := coins + 1
}

rule Main =
if ( coins < 25 ) then
    if ( insertedCoin = HALF ) then
        if ( available ( MILK ) > 0 ) then
            r_serveProduct ( MILK )
        endif
    else
        choose p in Product with p != MILK and available ( p ) > 0 do
            r_serveProduct ( p )
        endif
    endif
endif

exec Main
```

---

**Code 5.35:** *Coffee Vending Machine UASM specification*

The first difference among the UASM specification and `AsmetaL` specification is the structure. UASM specification does not have constraints in the position of function definition and rule definition. `AsmetaL` is more rigorous. It is divided into three section: the *signature* contains the functions and domains definition, the *definition* section contains the rules implementation and the functions and domains are initialized in the *default init* section. Furthermore `AsmetaL` requires the *Standard Library* which contains definition of basic domains, operators and basic functions.

---

```

asm coffeeVendingMachine

import StandardLibrary

signature:

    enum domain Cointype = {HALF | ONE}
    enum domain Product = {COFFEE | TEA | MILK}
    dynamic domain Quantitydomain subsetof Integer
    dynamic domain Coindomain subsetof Integer
    controlled available: Product -> Quantitydomain
    controlled coins: Coindomain
    monitored insertedCoin: Cointype

definitions:

rule r_serveProduct($p in Product) =
    par
        available ($p) := available ($p) - 1
        coins := coins + 1
    endpar

main rule r_Main =
    if ((coins < 25)) then
        if ((insertedCoin = HALF )) then
            if ((available (MILK) > 0)) then
                r_serveProduct[MILK]
            endif
        else
            choose $p in Product with $p != MILK and available ($p) > 0 do
                r_serveProduct[$p]
            endif
        endif
    endif

default init s0:

    domain Quantitydomain = {(0)..(10)}
    domain Coindomain = {(0)..(25)}
    function available ($productparam0 in Product) = 10
    function coins = 0

```

---

Code 5.36: *Coffee Vending Machine AsmetaL specification*

In the specifications, two enumerative domains are defined *CoinType* and *Product*. In UASM the keyword *enum* is used to identified the enumerative domain, while in AsmetaL two keyowrds are used: *enum domain*. The other difference is in the elements separator, in UASM comma is used, while AsmetaL uses the vertical bar. In both cases enumerative domains are extendible.

Other domains can be defined, in UASM the domains are all dynamic, while in **AsmetaL** they can be dynamic or static. For this reason in the **AsmetaL** specification the domain is preceded by the keywords *dynamic domain*, if *dynamic* is omitted it is considered static. Furthermore in **AsmetaL** the domains are divided in abstract domain (any type of domain) and concrete domain (subset of predefined numerical set e.g. integer, real).

In UASM the keyword used is *of* while in **AsmetaL** is *subsetof*, in both cases the keyword is followed by the name of the concrete domain. In UASM the initialization is performed after the domain definition preceded by the keyword *initially*. While in **AsmetaL** the domains are initialized in the *default init* section.

The functions definition is preceded by the keyword *controlled*, *monitored*, *static*, *derived* or *out* in both cases. After the keyword that identifies the type of function, in UASM the name of the function is defined, followed by the domain and codomain separated by the symbol  $\rightarrow$ . Subsequently, if necessary the function can be initialized using the keyword *initially*. In **AsmetaL**, the function name is declared after the keyword that identifies the type and it is followed by the colon mark. After, the domain and codomain are separated by the symbol  $\rightarrow$ . The function initialization is in *default init* section.

The rules definition has some differences between UASM and **AsmetaL**. UASM rules are preceded by the keyword *rule* and the main rule is defined by writing the name of the rule preceded by the keyword *exec* after the rules are defined. In **AsmetaL**, the name of the rules must start with the characters *r\_* and they are preceded by the keyword *rule*. The exception is for the main rule. The main rule is preceded by the keyword *main rule* and the name starts with the characters *r\_*.

The rules are very similar, the differences are:

- parallel rule: in UASM the parallelism can be represented using the keywords *par - endpar* or using the braces; in **AsmetaL** only the keywords *par - endpar* are admitted.
- variables: in **AsmetaL** the variables name are preceded by the symbol  $\$$ .
- call rule parameters: in UASM the parameters are between round brackets while in **AsmetaL** they are listed in brackets.



---

## Visualization for Abstract State Machines

---

Formal models are in principle accepted as the only way to specify in a precise and rigorous way the informal system requirements: they help to understand what has to be developed and to prove properties already at the early stages of the system development. However, formal specification languages are not widely used in industry, and practitioners largely consider formal methods “too hard to understand and use in practice”. Limiting factors are the lack of *simplicity*, *learnability*, *readability*, *easiness of use* of formal notations [124]. All these qualities are fundamentals to achieve easiness of development and comprehension of models, particularly for large, complex software systems. Requirement models should act as a communication medium among customers, users, designers, developers, and this common understanding is fundamental for the success of the system realization. However, since the mathematical notation is not always intuitive, and the size of the specification often consists of several pages of rules and formulas, model comprehension is threatened. Visualization is considered as a good means for people to communicate and to get a common understanding. Indeed, the use of diagrams and graphical blocks is at the base of the mostly used notations in industry, as FSMs (and their extensions) or UML, the latter nowadays accepted as the industrial standard for system design. However, their shortcomings, as limited expressiveness for FSM w.r.t. other formal notations [41] or semantics lack for UML [45], are well-known. Ever since UML appeared, many modelling approaches have been developed which try to use UML (or one of its profiles or domain-specific UML-like notations) as front-end of the requirements specification and formal notations as back-end of the process, to provide rigour and preciseness to lightweight models and make model validation and verification possible [83, 103, 110, 112, 121].

A concrete textual notation, called `AsmetaL`, to encode the ASM mathematical model as defined in the Lipari Guide [71] has been proposed in [65]. `AsmetaL` has been designed by exploiting the metamodelling paradigm.

The experience in trying to build and read very large system specifications [15, 23] has shown that the complexity of the behaviour being described overwhelms the reader, and most users (even the authors of the specification) need help in navigating and understanding it. This also happened while the development of the Hemodialysis machine case study [16] (see Chapter 9). We tried, at first, to directly specify the ASM models from the textual description of the requirements. Although the refinement process helped us in managing the complexity of the case study, we still had some problems in discussing among us about the solution. So, we started making some drawings, whose notation was inspired by different sources: control flow graphs, UML state machines, sequence diagrams, etc. The lack of a way to graphically represent ASM models was clear. A further observation we have made is that most of the new ASM users start developing ASM models as control state ASMs, a particular frequent class of ASMs – proposed by Börger in [41] – useful to model system modes (or control states). Control state ASMs have an intuitive graphical representation by means of FSM-like state diagrams. However, when the system to model is very complex, the resulting control state is too complicated and fails in achieving its main aim, i.e., easily communicating the behaviour of the system. Moreover, a systematic use of control state ASMs is missing, and there is no algorithmic support to build or reconstruct such machines from models written in textual notations.

Starting from the motivations that (a) formality is important but also understanding and communicating among stakeholders is fundamental, (b) visualization of formal models can surely aid the understanding of model structure and behaviours, (c) visual editing is often used to help designers and developers to graphically build complex models [57], we introduce a graphical notation for ASMs.

### 6.1 A visual notation for Abstract State Machines

---

The proposed visual notation [17] supports basic ASMs and sequential composition, it is defined in terms of a set of construction rules and schemas that give a graphical representation of an ASM and its rules. We assume that the graphical information is represented by a visual graph in which nodes represent syntactic elements (like rules, conditions, rule invocations) or states, while edges represent bindings between syntactic elements or state transitions. We do not introduce a graphical representation for the signature (functions and domains) and properties, since we believe that they can be already easily understood from the textual model. The final goal is to have a graphical visualization derived from textual representation as shown in Figure 6.1.

#### Visualization $\longrightarrow$ From textual to graphical representation

The first usage scenario consists in writing an ASM model in a textual representation (`AsmetaL`) and then derive a graph from it. Such approach can be used when

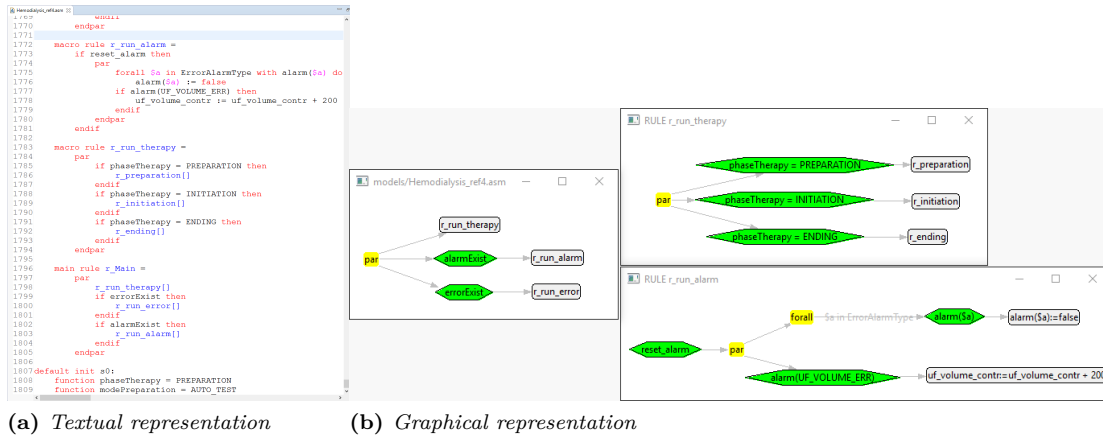


Figure 6.1: Visual notation

the modeller is familiar with the ASM syntax, but (s)he wants to have a graphical representation of the model for its better understanding and communication. If the ASM model is syntactically correct, also the produced graph is correct. In the visualizer, the user can activate some optimizations (see Section 6.3), in order to have different views of the same model: structural (with different levels of optimization), or semantic (behavioural).

## 6.2 Visual Trees

We introduce the relevant concepts which bring to a graphical representation of an ASM model in terms of a navigable forest of tree structures, i.e., a forest of trees connected by navigation links.

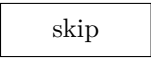
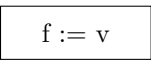

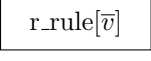
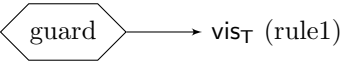
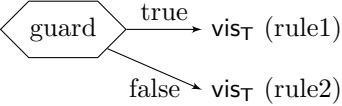
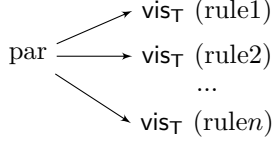
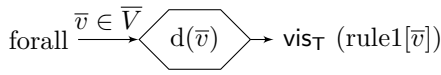
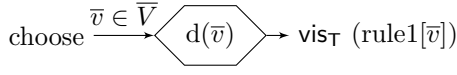
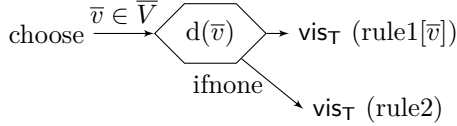
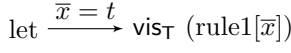
**Definition 6.2.1.** *The visual notation for ASMs is given by the bijective function  $\text{vis}_\tau$  between an ASM rule and a visual tree.*

**Definition 6.2.2.** *The function  $\text{vis}_\tau$  is given by Table 6.1.*

1. For basic rules (update, skip and macro call) the function simply returns a tree with only one node (the root).
2. For compound rules (conditional, block, forall, choose, let), one must apply the schema given in Table 6.1 and recursively call the function  $\text{vis}_\tau$  on component rules.

Table 6.1 describes the semantics of ASM transition rules, and shows the proposed graphical representation and the `AsmetaL` textual notation. The function  $\text{vis}_\tau$  is only based on the syntactical structure of the ASM and it can always be applied. Tree leaves are always skip, update, or call rules, and they are shown in boxes. Note that a call rule invokes a macro rule that has its own tree that, however, is not part of the main tree. At the end, one can obtain a tree for every rule declaration by applying  $\text{vis}_\tau$  to its definition. The visualization of an ASM is given by the forest compound of all the trees of the declared rules. To navigate this visual view, the entry point is the tree for the main rule and, from every call rule, one can navigate to the tree of the invoked macro rule by a virtual *navigation*

Table 6.1:  $\text{vis}_\top$ : Mapping from ASM transition rules to visual trees

Rule	Visual tree	AsmetaL notation
<b>Skip rule</b> do nothing		skip
<b>Update rule</b> update $f$ to $v$		$f := v$
<b>Macro call rule</b> invoke rule $r\_rule$ with arguments $\bar{v}$ (if any)		$r\_rule[]$
		$r\_rule[\bar{v}]$
<b>Conditional rule</b> execute $rule1$ if $guard$ holds, otherwise execute $rule2$ (if given)		if guard then rule1 endif
		if guard then rule1 else rule2 endif
<b>Block rule</b> execute $rule1$ ... $rule_n$ in parallel		par rule1 rule2 ... rule_n endpar
<b>Forall rule</b> execute $rule1$ with all values $\bar{v} \in \bar{V}$ for which $d(\bar{v})$ holds		forall $\bar{v} \in \bar{V}$ with $d(\bar{v})$ do rule1 $[\bar{v}]$
<b>Choose rule</b> execute $rule1$ with a $\bar{v} \in \bar{V}$ for which $d(\bar{v})$ holds. If no such $\bar{v}$ exists, execute $rule2$ (if given)		choose $\bar{v} \in \bar{V}$ with $d(\bar{v})$ do rule1 $[\bar{v}]$
		choose $\bar{v} \in \bar{V}$ with $d(\bar{v})$ do rule1 $[\bar{v}]$ ifnone rule2
<b>Let rule</b> execute $rule1$ substituting $\bar{t}$ for $\bar{x}$		let $(\bar{x} = \bar{t})$ in rule1 $[\bar{x}]$ endlet

*link*, which is not visualized in the graphical representation. By considering the navigation links in the visualization, the resulting structure is a graph, as a macro rule can be called by different call rules.

## 6.3 Visual Patterns

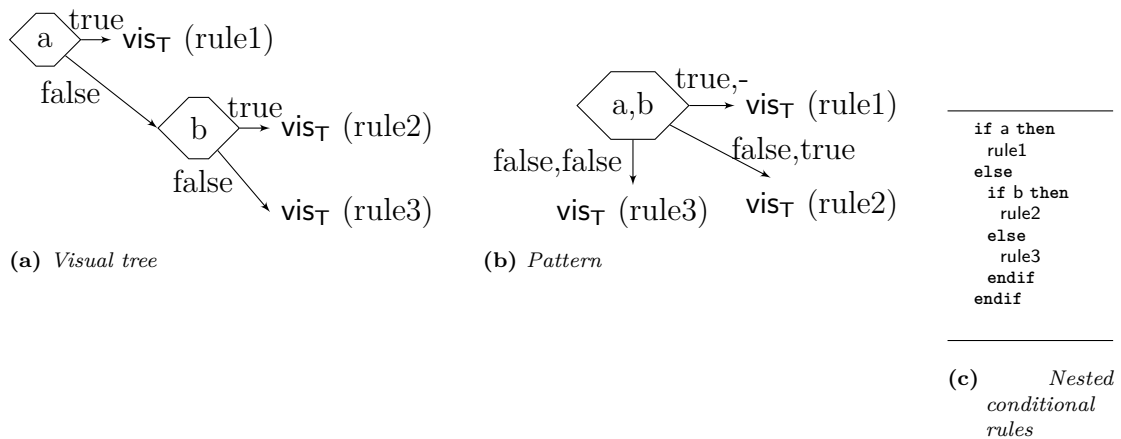
A pattern is a schema of connected tree nodes that is recurring and conveys a *structural* or *semantic* (i.e., behavioral) information. Therefore, identifying a pattern and substituting the entities belonging to it with a simplified structure is of interest.

### 6.3.1 Structural patterns

We identify the following structural pattern that permits to obtain a more compact representation of the model structure.

#### Nested Guards Pattern

The pattern regards the use of *nested conditional rules*. Suppose that you have a conditional rule as shown in Figure 6.2c.



**Figure 6.2:** Structural pattern – Nested guards pattern

By applying the visual trees in Table 6.1, one would obtain the tree shown in Figure 6.2a. However, one can visualize the rule in a more compact way as shown in Figure 6.2b. The pattern is applicable to any depth of nested conditional rules. One just has to collect all the guards  $g_1, \dots, g_n$ , and create only one decision node comprising all the guards separated by commas. The decision node has as many exiting arcs as the number of conditional branches not containing another nested conditional rule, but a different rule  $rule_i$ ; each arc is labeled with the evaluations of the guards that permit to take that particular arc and fire rule  $rule_i$ . Evaluations of guards that are not relevant for the firing of a rule  $rule_i$  are depicted with symbol “-”. The decision node has up to  $n + 1$  exiting arcs. Note that the pattern does not necessarily produce a tree that is more clear to understand, but it always provides a more compact representation of the nested conditional rules. For this reason, we let the modeller decide if (s)he wants to apply it.

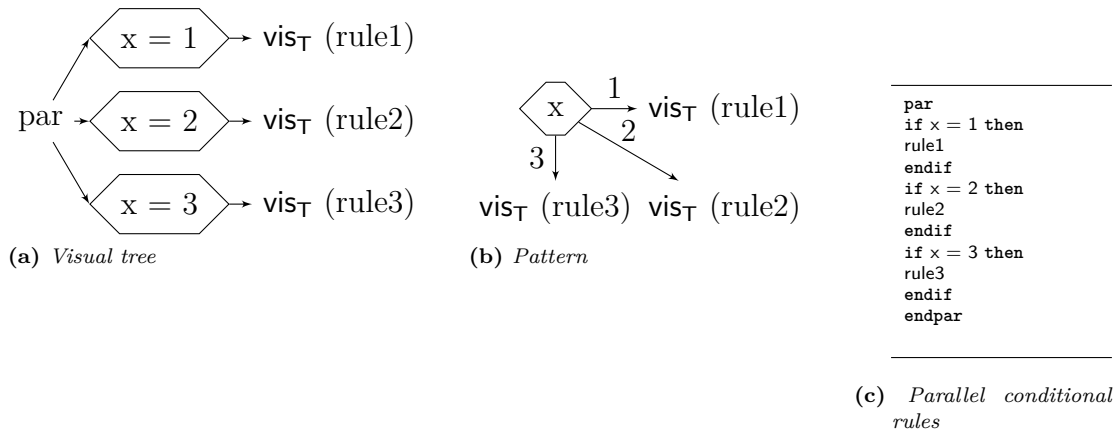
### 6.3.2 Semantic Patterns

Any ASM model can be always represented using visual trees and possibly optimized by applying structural patterns. The resulting tree visualizes the structure of the ASM. However, sometimes it is possible to infer from the model also some hints on the behaviour of the machine. For this reason, we introduce *semantic patterns* that can be applied when it is possible to infer from the model some information on the workflow of the machine. We identify here three semantic patterns: *mutual exclusive guards*, *state*, and *state flow* patterns.

#### Mutual exclusive guards pattern

In case of parallel conditional rules having mutual exclusive guards, it could be useful to represent that the workflow of the machine follows only one of the possible parallel execution paths.

The *mutual exclusive guards pattern* has been defined for this purpose. It is applicable when the rule guards check the current value of a given location that can assume disjoint values. This guarantees mutual exclusion among the guards of the conditional rules. Consider, for example, the ASM rule in Figure 6.3c.



**Figure 6.3:** Semantic pattern – Mutual exclusive guards pattern

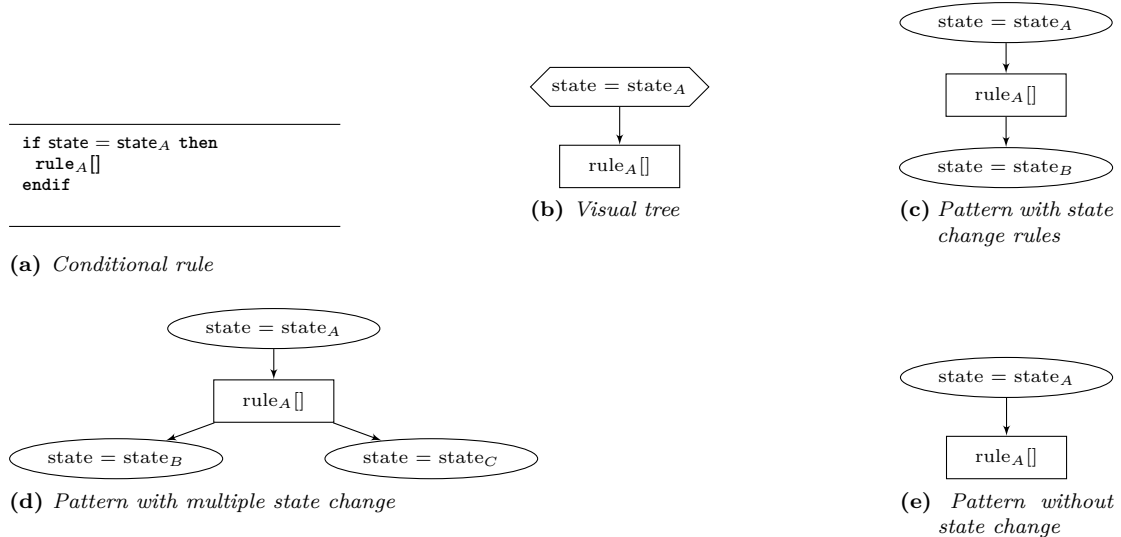
It fires the parallel execution of three conditional rules guarded by the current value of the location  $x$ . Applying the visual tree in Table 6.1 to this rule, we obtain the representation given in Figure 6.3a. However, one can understand that the three conditions on  $x$  are mutually exclusive and, therefore, visualize the rule in a more compact way as in Figure 6.3b, showing that the machine workflow follows only one of the three possible paths<sup>1</sup>.

#### State pattern

Often, it could be desirable to represent the machine behaviour as a flow of activities along a sequence of states of control, i.e., configurations (or *modes*) in which the machine can be. Therefore, we enrich our visual notation with a special node

<sup>1</sup>Note that the pattern can be detected by a simple static analysis of the model because of the particular guard structure we consider. If we would like to handle any type of guard, detecting the pattern would require to use a logical solver.

(an ellipse) representing information about the (control) *state* in which a given rule can be executed. Suppose the model is as shown in Figure 6.4a, where  $\text{rule}_A$  is a macro call rule that might call (directly or indirectly) the update rule  $\text{state} := \text{state}_B$ .



**Figure 6.4:** Semantic pattern – State pattern

Using only the visual trees defined in Table 6.1, the rule would be represented by the schema shown in Figure 6.4b. However, supposing the modeler wants to use the function *state* to identify states of control, if  $\text{rule}_A$  changes the *state* from  $\text{state}_A$  to  $\text{state}_B$ , one can build the graph as shown in Figure 6.4c to explicitly represent the state change. In case  $\text{rule}_A$  can bring to different states (e.g., states  $\text{state}_B$  and  $\text{state}_C$ ), the pattern is as shown in Figure 6.4d. Instead, if rule  $\text{rule}_A$  leaves the mode unchanged, the pattern is as shown in Figure 6.4e. Note that rule  $\text{rule}_A$  will be represented as a macro call rule, if this is not already the case.

### State flow pattern

The definition of the state pattern can be extended to graphically represent a flow of activities along a sequence of control states. Suppose to have the code reported in Figure 6.5a and that  $\text{rule}_A$  contains the update rule  $\text{state} := \text{state}_B$  and  $\text{rule}_B$  contains the update rule  $\text{state} := \text{state}_C$ .

By applying the state pattern explained above, one would obtain the visual graph in Figure 6.5b. However, the evolution of the system from state  $\text{state}_A$  to  $\text{state}_B$  and then to  $\text{state}_C$  can be made explicit, and the graph can be rewritten as in Figure 6.5c. Note that if rule  $\text{rule}_B$  does not update *state*, the flow ends with  $\text{rule}_B$ . Instead, if rule  $\text{rule}_B$  updates *state* to  $\text{state}_A$ , the resulting structure is a graph as shown in Figure 6.5d. Note that if the state flow pattern is applicable, also the mutual exclusive guards pattern is applicable.

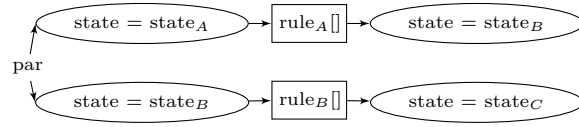
Following, two examples show the application of the state flow pattern. The result of the first example is like the pattern in Figure 6.5c, while the results of the second example is a diagram like the patter in Figure 6.5d.

```

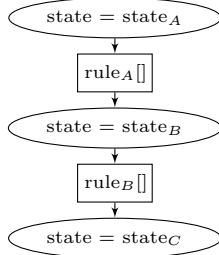
par
  if state = stateA then
    ruleA[]
  endif
  if state = stateB then
    ruleB[]
  endif
endpar

```

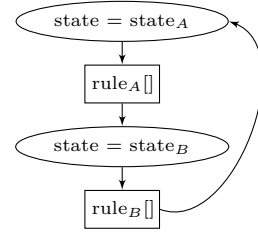
(a) Parallel conditional rules



(b) State pattern



(c) Pattern as tree



(d) Pattern as graph

Figure 6.5: Semantic pattern – State flow pattern

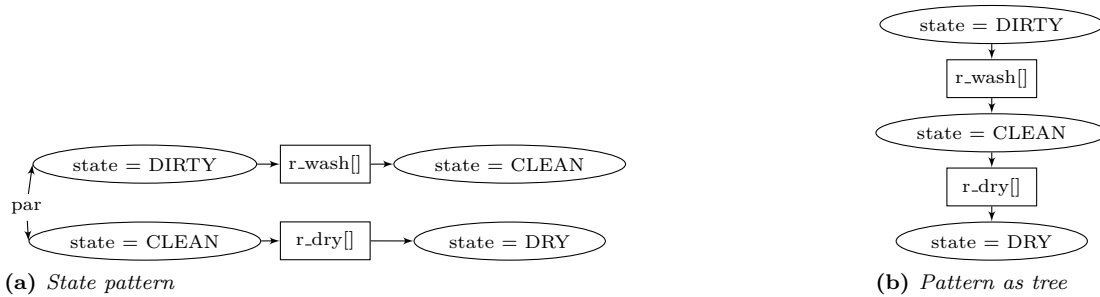
**Example 1: car wash** At the beginning of the washing cycle, the car is DIRTY. The cleaning machine washes it, the car is CLEAN but it still wet. It is DRY only after the drying cycle. The ASM model is shown in Code 6.1. By applying the state flow pattern (see Figure 6.6b), the readability of the state flow is more clear compared to the state pattern shown in Figure 6.6a.

<pre> asm CarWash  signature: enum domain CarState = {DIRTY   CLEAN   DRY}  controlled state: CarState  definitions: rule r_wash = state := CLEAN  rule r_dry = state := DRY </pre>	<pre> main rule r_exec_machine = par   if state = DIRTY then     r_wash[]   endif   if state = CLEAN then     r_dry[]   endif endpar  default init s0: function state = DIRTY </pre>
---	--

Code 6.1: State flow pattenr example 1

**Example 2: industrial machine** An industrial machine can be in three different states: OFF, ON, WORKING. Initially the machine is OFF and the only action admitted moves it to state ON. After that the machine goes in state WORKING, and the next action takes it in state OFF. The model of the system is shown in Code 6.2. By applying the state pattern (see Figure 6.7a) the graph is more complicated and the state flow is less clear than the one obtained using





**Figure 6.6:** Example 1: Semantic pattern – State flow pattern

the state flow pattern (see Figure 6.7b).

```

asm Machine

signature:
enum domain MachineState =
{OFF | WORKING | ON}

controlled state: MachineState

definitions:
rule r_turnOn =
state := ON

rule r_work =
state := WORKING

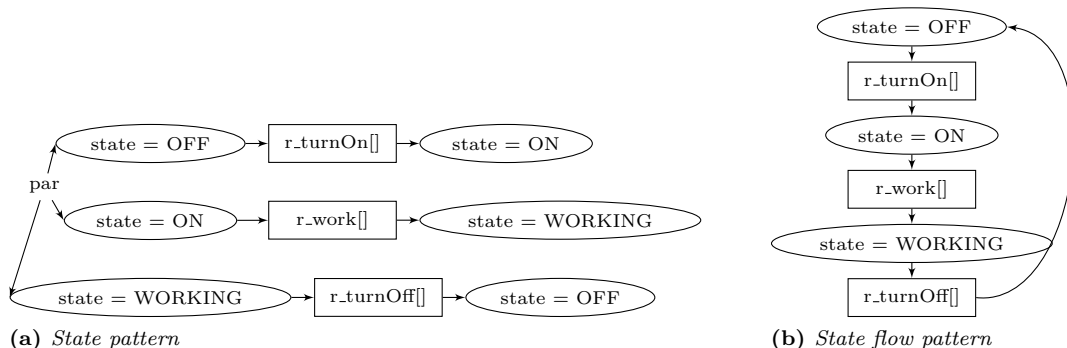
rule r_turnOff =
state := OFF

main rule r_exec_machine =
par
if state = OFF then
r_turnOn[]
endif
if state = ON then
r_work[]
endif
if state = WORKING then
r_turnOff[]
endif
endpar

default init s0:
function state = OFF

```

**Code 6.2:** State flow patter example 2



**Figure 6.7:** Example 2: Semantic pattern – State flow pattern

It is possible to notice from the two examples shown above, that the state flow pattern greatly simplifies the readability of the graphs in case it is applicable to

the model under investigation.

### 6.4 Tool

---

A prototypical tool called `AsmetaVis` has been developed, that permits to represent the visual trees and some of the visual patterns we have presented. The tool permits to obtain the graphical representation of a specification written in `AsmetaL`. The tool is currently able to visualize the ASM in two modes:

- *basic visualization* in which the ASM is visualized using only the visual trees presented in Section 6.2;
- *semantic visualization* in which information on the workflow of the model is visualized using semantic patterns (see Section 6.3.2). Note that the tool automatically identifies the semantic patterns without any hint from the user. It first tries to apply the state and state flow patterns; if these are not applicable, it tries to apply the mutual exclusive pattern.

Figure 6.8 shows the basic and the semantic visualizations applied to the Example 1 (see Section 6.3.2) in `AsmetaVis`.

Due to the complexity of some models, the tool loads the `AsmetaL` model and shows the graph of the main rule. A double-click on a macro call rule node causes the visualization of the corresponding macro rule graph; in this way, we provide the navigation links described in Section 6.1. An example is shown in Figure 6.9 using the Example 1 (introduced in Section 6.3.2). Initially the tool shows the window “models/CarWash.asm” that corresponds to the visualization of the main rule. When the user double-clicks on the call rule “r\_wash”, the corresponding window “RULE r\_wash” is open. The same process is applied for the visualization of the windows “RULE r\_dry”; it is opened when the user double-clicks on the call rule “r\_dry”.

The tool is integrated in the `Asmeta` framework as eclipse plugin<sup>2</sup> and it uses Zest for implementing the visualization features<sup>3</sup>.

### 6.5 Preliminary evaluation of visual notation

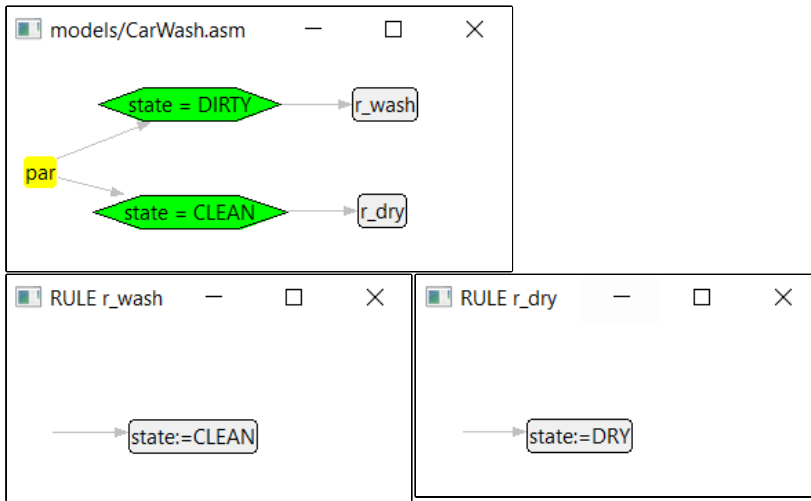
---

We conducted a preliminary experiment to evaluate whether the proposed visual notation can help in understanding a model. We interviewed 15 students who attended a course on formal system modelling and verification at the University of Milan (ten lectures on ASMs), and 11 who attended a course on principles of programming languages at the University of Bergamo (six lectures on ASMs). We took the (last refined) textual model of the hemodialysis case study (see Chapter 9), that consists of 163 macro rules and 1880 lines of code. We gave the textual model to half of the students and its graphical representation to the other half. Then we asked them a question in order to evaluate their understanding of the model (UQ: Which are the phases of the hemodialysis treatment and in which order are they executed?). We measured the time taken for answering the question. After this experiment, we gave them also the other representation (the textual one for those having the graphical one, and vice versa) and we asked them

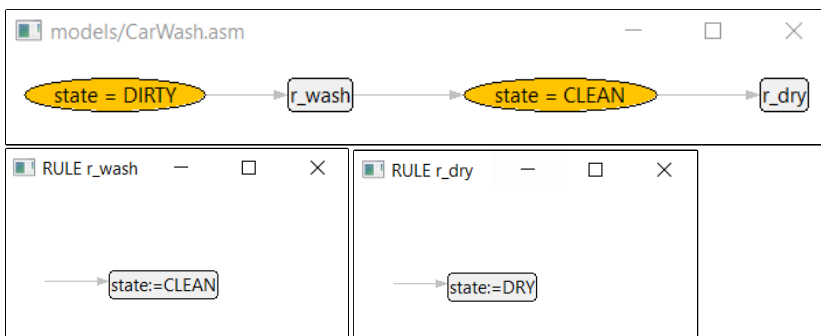
---

<sup>2</sup><http://asmeta.sourceforge.net/download/asmetavis.html>

<sup>3</sup><https://www.eclipse.org/gef/zest/>



(a) Basic visualization



(b) Semantic visualization

Figure 6.8: AsmetaVis tool

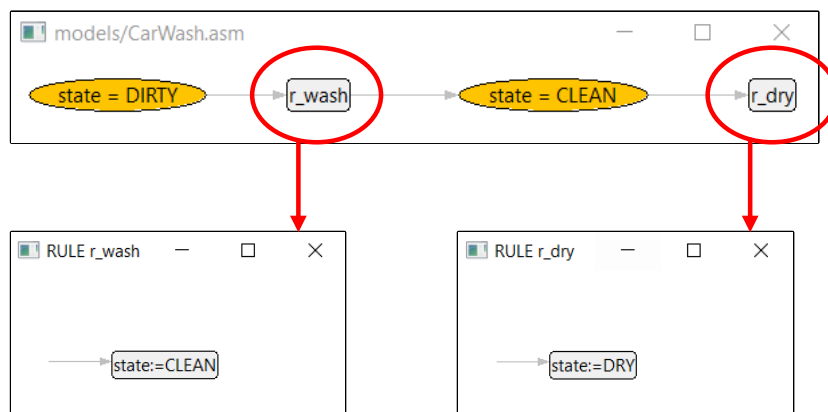


Figure 6.9: Example 1: visual notation using the tool - in red the navigations links

**Table 6.2:** *Experimental results of preliminary evaluation of visual notation*

Group	UQ (% correct answers)	Avg. time (sec)	SQ (% affirmative answers)
Graphical	92.3	135	100
Textual	73.0	226	7.6

to identify the same elements in both representations (we did not count the time in the second phase of the evaluation). Then we asked them a question regarding their satisfaction about the notation they used at the beginning (SQ: Are you satisfied with the notation you used at the beginning?).

Table 6.2 shows the results of the experiment. By UQ, we observe that the graphical notation permits to understand the model semantics better in less time than the textual notation. Regarding the level of satisfaction (SQ), all the students who used the graphical notation were satisfied and they would not have preferred using the textual one. Instead, only 7.6% of those using the textual notation were satisfied and 92.4% of them said that they would have preferred using the graphical one.

---

# CHAPTER 7

---

## Automatic Code Generator

---

In Chapter 4, the **Asmeta** framework has been presented. Given the informal requirements the user derives the models and performs validation and verification activities. The last step of V&V verifies the conformance between the last ASM model and the code (machine code). The code is externally provided or developed by the user considering the last ASM model. Code generation process aims to generate automatically the code from the last formal model. Moreover, it tries to keeping true the system behaviour and the properties verified during V&V. The target language chosen for the first code generator tool from ASM specification is C++, in particular C++ for Arduino devices. In the next sections the transformation process is shown and the design choices are motivated. The starting point of translation process is the formal specification written in UASM (see Chapter 5). UASM has been chosen because it aims to be a stable language kernel for ASMs languages. The tools presented in Chapter 4 process **AsmetaL** models and cannot be used for UASM models. For this reason a translator from UASM to **AsmetaL** has been developed (see Sect. 5.8).

The next sections show the process and the tools used to generate C++ code from UASM specification.

### 7.1 Model transformation

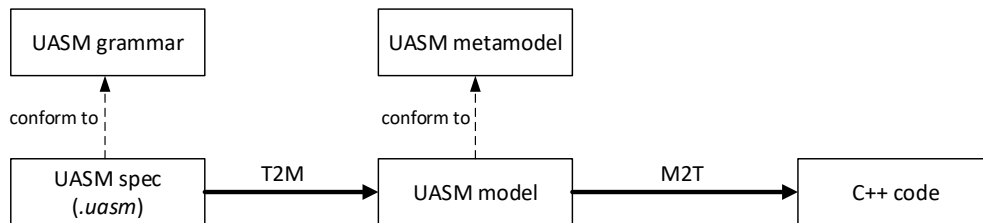
---

The starting point to get the machine code is the formal specification written in UASM conforms to UASM grammar. After that, the specification is translated into UASM model conforms to the meta-model using text-to-model (T2M) transformation. T2M transformation is often called parsing and it is performed by the parser. The parser takes as input a string (for example the text file for the

source code), it verifies if the string is conform to the grammar and creates the corresponding model. In case the text file is not conform to the grammar, an exception is raised and the parsing process is stopped. Starting from model, two solutions can be adopted to perform code generation: model-to-text (M2T) and model-to-model (M2M). The former translates the model into textual representation, e.g. source code of a specific language; the latter translates the starting model into the model of the target textual representation, e.g. the model of C++ language.

### 7.1.1 Model-to-Text

Model-to-text is the most used technique, since it translates the model directly into source code. For example, in the present case (see Figure 7.1) the UASM model is translated into C++ code.



**Figure 7.1:** *M2T Transformation*

There are different technologies to support M2T transformation process, the most common are:

- **Xpand**<sup>1</sup> is based on EMF (Eclipse Modeling Framework) framework and provides code generation facilities like a meta-model (Ecore) to describe the models, a set of Java classes for the components of the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. The main disadvantage of Xpand is that the output generation is slow, the code is not maintained and it is not well tested.
- **Xtend2**<sup>2</sup> is the successor of Xpand based on Java dialect and Xpand functionalities. It has the following advantages: - it is translated into Java - it is faster than Xpand - it is debuggable - it has its own IDE (Integrated Development Environment).

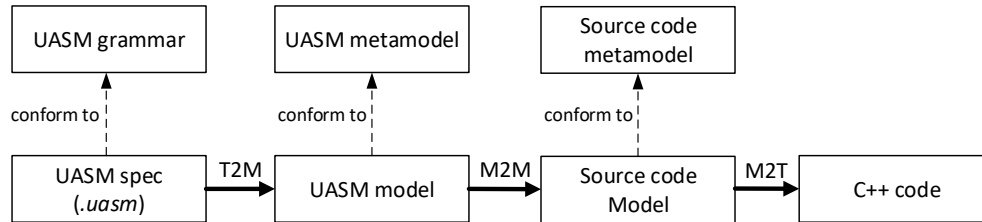
### 7.1.2 Model-to-Model

Model-to-model transforms the model of the specification into the model of the target source code. There are two types of transformations: endogenous and exogenous; the first is between models conform to the same meta-model, while the latter refers to models having different meta-models. The second is the case that matches the present case, because UASM model and source code model of

<sup>1</sup><https://eclipse.org/modeling/m2t/?project=xpand>

<sup>2</sup><http://www.eclipse.org/xtend/>

the target language are conform to different meta-models. As shown in Figure 7.2



**Figure 7.2:** *M2M Transformation*

M2M transformation leads two steps to transform the starting model into runnable code. UASM model is first transformed into the model of the target language and then, through M2T transformation, it is translated into target source code.

There are different technologies to support this process, the most common are:

- **Epsilon**<sup>3</sup> is a family of languages and tools for code generation, model-to-model transformation, model validation, comparison, migration and refactoring that works out of the box with EMF and other types of models.
- **ATL Transformation Language**<sup>4</sup> is a model transformation language. It takes as input the meta-model, then it applies defined transformation rules and produces the output model.
- **Henshin**<sup>5</sup> provides a transformation language for EMF. The transformation language is based on a graphical syntax where every rule is represented by a graphical diagram. The transformation diagrams quickly become extremely complicated.

## 7.2 Microcontrollers

A microcontroller (or MCU – MicroController Unit) is a small computer (SoC) designed to govern the operation of embedded systems. For example, they are used in automotive control systems, implantable medical devices, remote controls, industrial machines, appliances, power tools and toys. Microcontrollers have a small code memory that can be programmed via specific tools. Some of the most relevant aspects that influence the choice between all microcontrollers are: price, programming language and hardware support. Following, the most used microcontrollers are listed:

- **Arduino**<sup>6</sup> is an open source hardware and software project. It is used by hobbyists to perform some experiments and also by professionals to develop prototypes of systems. There are different programming environments, the most used are: the official Arduino IDE, that is recommended for beginners and the Eclipse plugin, which enables developers to deal with more complex

<sup>3</sup><http://www.eclipse.org/epsilon/>

<sup>4</sup><http://www.eclipse.org/at1/>

<sup>5</sup><https://www.eclipse.org/henshin/>

<sup>6</sup><https://www.arduino.cc/>

projects. The language used to program Arduino is C++11, with some additional libraries to manage the hardware. One of the advantages of Arduino is the price, it is cheaper and accessible by all. The base version includes some digital/analog pins and a serial port, but it is possible to extend the hardware by plugging additional shields (e.g. WiFi, bluetooth and motor control).

- **PIC**<sup>7</sup> is a family of microcontrollers made by Microchip Technology, they are derived from the PIC1650 originally developed by General Instrument's Microelectronics Division. PIC microcontrollers are widely used in industry and they can be programmed in assembly or in C, but some of them support C++. PIC provides any kind of desired peripherals and hardware support by paying the related cost.
- **Raspberry** is a computer implemented on a single board and it is used mainly in education. It is comparable with Arduino, the main difference is that Raspberry runs a full Linux distribution (Raspbian) while Arduino runs the program that holds in memory. It supports C++ and Python and can be extended with accessories (e.g. camera module and touch display).
- **PLC** (Programmable Logic Controller) is a digital computer used mainly in electromechanical processes and for safety-critical system because it is sturdy. IEC 61131-3 defines five programming languages for programmable control systems: function block diagram (FBD), ladder diagram (LD), structured text (ST), instruction list (IL), and sequential function chart (SFC). PLCs are very expensive because of their reliability in extreme conditions.

### 7.3 Design choices

---

In the previous sections many technologies, tools and hardware platforms were analysed. In this project, those chosen are shown below.

**Platform and Language** The first choice is the target platform to support. The final decision is Arduino, since an explicit request concerns embedded systems was received. It is very cheap and is widely used in the academic world. In addition, Arduino programs are written in C and C++ languages, which are the most used for embedded systems. Also PLC was evaluated, but it is not convenient because of its high costs and it requires the use of ad hoc programming languages often bound to specific vendors.

**Model transformation** The project is based on the Xtext implementation of UASM. Xtext framework provides a parser that given a textual file containing the ASM specification builds a collections of Java objects using the metamodel. Moreover, Xtext provides a set of Java APIs useful to access and modify the objects. Indeed, it offers an infrastructure for M2M and M2T transformations. As explained in [44], "... it seems easier to start with building a code generator by employing M2T transformation languages - especially when no metamodel for the target language exists. ...". Since in the present case the complete metamodel of C++

---

<sup>7</sup><http://www.microchip.com/design-centers/microcontrollers>



(in particular the metamodel for Arduino) is not available, M2T approach has been adopted. The tool chosen from those listed in Sect. 7.1.1 is **Xtend2** which provides interesting features for code generation and it is fully compatible with Java and Xtext.

## 7.4 Transformation process

---

The transformation process is designed as described in Figure 7.3. One goal of transformation process is an easily reconfigurable translation for the other target languages. For this reason the transformation process is based on black-box, compared to a more reconfigurable pipelined process with intermediate results. Another solution to keep the transformation process easily reconfigurable is to keep separated the code from the hardware configuration. So a different target platform would not affect the code generation but only the HW configuration step. This approach is partially inspired by a similar work about translation from Event-B to C++ [102]. In view of above, the steps implementing the translation are:

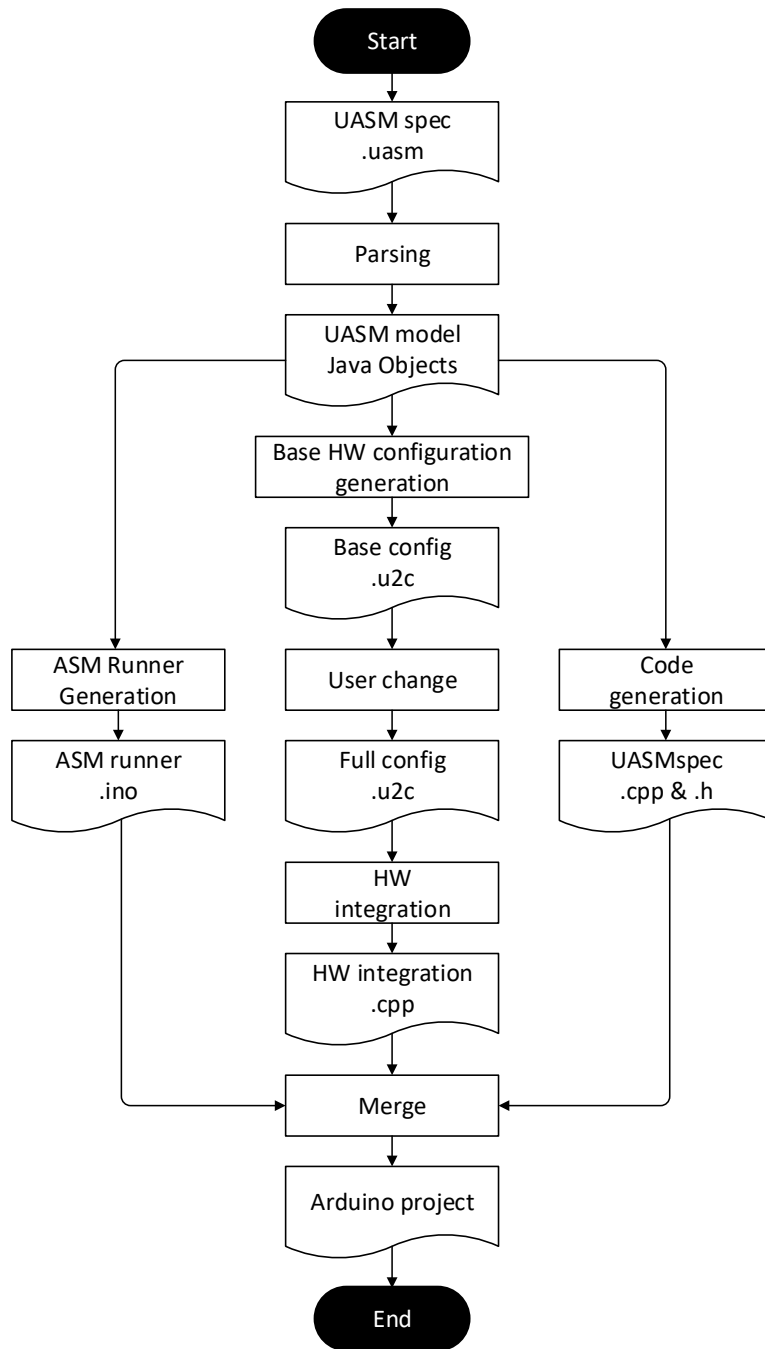
- **Parsing** given a textual UASM specification the parser generates the corresponding syntax tree as Java Objects.
- **Code generation** generates the corresponding C++ code from the parsed UASM specification.
- **Base HW configuration, HW integration and user change** define the hardware-specific aspects e.g. setup, input/output pin connections.
- **ASM Runner generation** writes the Arduino code (*.ino* extension) that contains the loop function, where the ASM main rule is iteratively called.
- **Merge** links the different source code files to create the Arduino project.

It is important to highlight that the result is the source code of the model provided. Therefore the generated code must be completed to implement all the aspects not included in the model.

**Parsing** The generation process, as described in Figure 7.4, starts from the UASM specification. The specification is written in a textual notation conform to the UASM grammar and contains all the information about the model. The UASM parser, provided by Xtext, takes as input the UASM specification and creates the abstract syntax tree of the model as Java Objects. This is easier to be processed compared to the textual representation.

**Code Generation** The code generation step, described in Figure 7.5, generates C++ code starting from the UASM model. M2T transformation is adopted for the code generation step. Once the model is parsed, it is transformed in Java objects that reflects the syntactical structure of the parsed file. The output is the C++ code that can be compiled and executed by any compiler that supports the C++11 (or more recent) standard. The C++ code is composed by the header (.h) and the source (.cpp) files.

**Base HW configuration, HW integration and User Change** The *HW configuration* goes in parallel to the code generation phase and aims to produce the HW-



**Figure 7.3:** Flow Diagram: transformation process from UASM specification to Arduino project.

specific part of the Arduino project which is missing in the automatic C++ code generator. The C++ code containing the *HW integration* is composed by three functions: the setup function, which contains the hardware initial settings, and

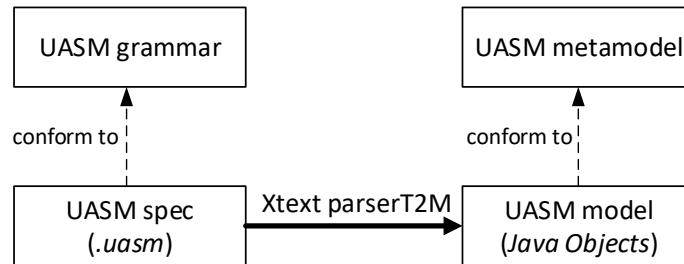


Figure 7.4: Parsing process

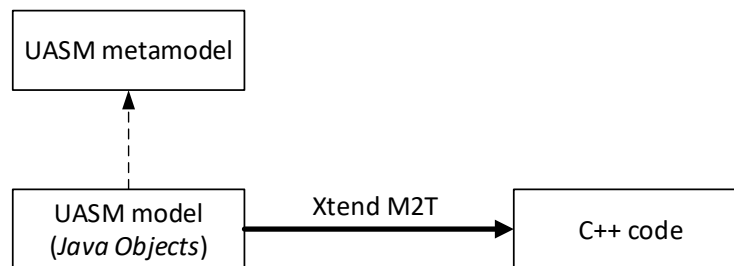


Figure 7.5: Code generator process

the two functions responsible for acquiring inputs and setting outputs. The M2T transformation starts from the UASM model and the tool automatically generates the base `.u2c` configuration file. At this step all functions are defined and they are distinguished between monitored and out functions. Monitored functions are assigned to the inputs while out functions to the outputs. Furthermore, the configuration file contains all hardware settings including Arduino version, which is the machine step delay, the path to the UASM specification and the list of bindings. The base configuration file is the minimal skeleton and the user has to complete this file in *User change* phase. In this phase, the user links monitored and out functions to physical hardware pins and further details can be added. Once the `.u2c` configuration file is complete, the HW integration code is automatically generated.

**ASM Runner generation** The generated C++ code is a complete translation of the ASM specification. However, to run the ASM on an Arduino board, the `loop()` function (a function continuously called) must be implemented. This part is kept in a separated `.ino` file to split the ASM behaviour and the execution policy. The `loop()` function executes iteratively the following methods: - `getInputs()`: reads the data from the input devices like sensors; - `Main()`: contains the behaviour described in the UASM specification; - `UpdateState()`: updates the state at the

end of each loop; - *setOutputs()*: sets the output values like the current state of led.

**Merge** In previous steps, different source codes have been produced. The C++ code containing the translation of the UASM specification, the C++ code for the HW integration and the *.ino* code that executes the ASM. The merge process simply links all the previously generated artefacts together to create the Arduino project. This is done using the include directive of C++ (see Figure 7.6).

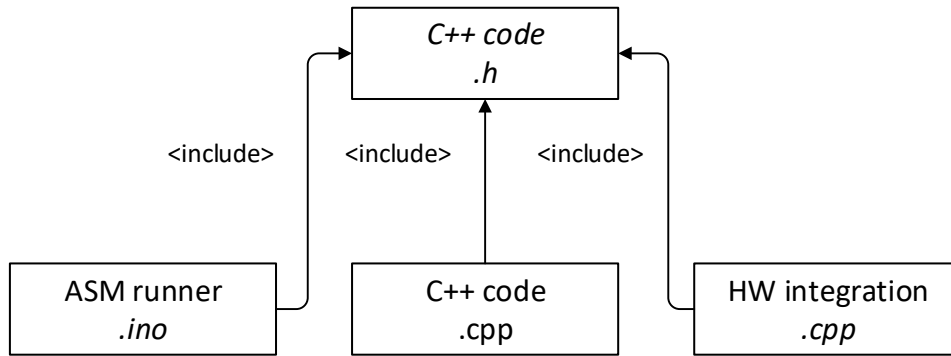


Figure 7.6: Merge process

## 7.5 Tool implementation

---

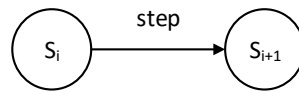
The translation from UASM specification (event-driven language) to C++ (imperative language) is not immediate because they belongs to two different programming paradigms. The tool presented is the first version and not all the functionalities of UASM are translated into C++. By reference to the ASMs classes, basic ASMs and sequential composition are supported. Furthermore, during the translation some compromises have been assessed. Two parameters are considered: correctness and performance. Correctness guarantees that safety and liveness properties still valid after the transformation. Performance could be in conflict with correctness, and for this reason a trade-off must be assessed.

### 7.5.1 Code Generation

The code generation implements one-to-one mapping between UASM constructs and C++ code. The transformation approach wants to assure the correctness of the transformation with respect to the original model. A formal prove of correctness will be provided as future work (see Section III), while in this section the correctness is demonstrated through an informal prove.

We consider a generic transition from state  $S_i$  to  $S_{i+1}$  as shown in Figure 7.7.

A step is the transition from state  $S_i$  to  $S_{i+1}$  obtained performing the following activities: the ASM reads the environment functions, executes the main rule,

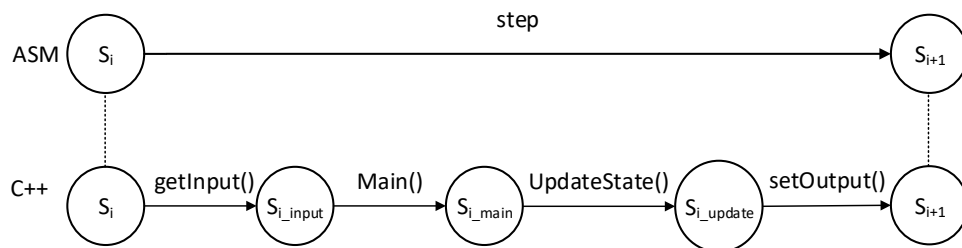


**Figure 7.7:** *ASM state transition*

computes the update set to obtain the state  $S_{i+1}$  and updates the environment variables. In C++ a state consists of set of variables. One of the differences between ASMs and C++ is that in ASMs the functions are updated during the computation of the update set, while in C++ variables are immediately updated. For this reason, functions that could change their value during state  $S_i$ , but the new value is taken into account in the state  $S_{i+1}$ , are translated as an array of two elements. The first represents the value of the functions in the current state, the second represents the value of the functions in the next state. The other difference is about the ASMs step. As mentioned above, it is translated by a set of activities executed sequentially. This behaviour is simulated through a set of C++ methods executed in the following order:

- void getInput(): reads the environment variables
- void Main(): executes the actions defined by the main rule in the ASM models and collects the update set in one element of the array as previously explained.
- void UpdateState(): copies the value of the second element of the array into the first element. This will represent the state  $S_{i+1}$
- void setOutput(): the environment variables are updated.

In this way, each step of the ASM is translated in four intermediate steps in C++. Figure 7.8 and Table 7.1 show the correspondence between ASM step and C++.



**Figure 7.8:** *ASM and C++ correspondence*

In the next paragraphs the translation adopted is shown.

**Minimal ASM skeleton** All the ASMs translated have a common skeleton.

An example of basic UASM model (see Code 7.1) transformation is shown in Code 7.2. Considering Arduino program (see Code 7.3), the C++ methods are executed by *loop()* function and an instance of the machine is globally instantiated.

	ASM	C++
Step	Read environment variables	getInput()
	Execute Main rule	Main()
	Apply update set and update environment variables	UpdateState() setOutput()

Table 7.1: ASM and C++ correspondence

---

```

asm MyAsm
...
<function definitions>
...
<rules definitions>
...

rule Main = ...

exec Main
    
```

---

```

class MyAsm{
private:
<properties definitions>
...
<method definitions>
...
public:
MyAsm(){..}
void getInput(){..}
void Main(){..}
void UpdateState(){..}
void setOutput(){..}
};
    
```

Code 7.1: Minimal UASM model template

Code 7.2: Minimal C++ template

---

```

MyAsm myAsm;
void setup(){..}
void loop(){
    myAsm.getInput();
    myAsm.Main();
    myAsm.UpdateState();
    myasm.setOutput();
}
    
```

Code 7.3: Arduino execution

**Parallelism** Parallelism is a fundamental point of ASMs because the operation done in the same transition are done in parallel, but neither multi-thread nor parallelism is supported by Arduino. Nevertheless it is still possible to run ASMs in a parallel-like way by showing the same copy of the current state to all sequentially-executed threads following the approach proposed by J. Schmid in [114]. ASMs run in discrete steps where each step consists of four operation: acquire inputs, perform the main rule, update the state and release the outputs. The machine state (represented by controlled functions) is modified only during the main rule execution. To simulate parallel execution the state is duplicated: the present state and the future state. Modification made by threads on controlled functions

will affect only the future state, while status readings will refer to the current state. In this way, each thread will see exactly the same value of the current state disregarding the execution order. The modification made by threads will take place only when every thread is done, this means in *UpdateState()* method. This approach guarantees the proper evolving of the machine state, even though it is not a true parallelism. Table 7.2 shows a simple example of parallelism. The example swaps the values of functions *x* and *y*. If we translate this example in C++, the result is not the same. The variables *x* and *y* would contain the value of *y*. For this reason, we introduce an array of two values for each function; the first value corresponds to the value of the function in the current state of the ASM execution and the second value corresponds to the value of the function in the next state of the ASM execution. In this way, when an assignment occurs in C++ we translate it following this rule: we use the next state (the second element of the array) to the left side of the assignment; we use the current state (the first element of the array) to the right side of the assignment; after the execution of the rules in the current state (the *Main()* method in C++) we assign all the values of the next state to the current state. The new current state is used in the next step of execution. We do not consider the case of inconsistent updates because the methodologies applied during the analysis of the ASM model guarantee that the inconsistent updates will not occur. This process is applied also to the other translations in addition to the parallelism.

UASM	Main in C++	UpdateState in C++
<pre> controlled x in INTEGER initially 5 controlled y in INTEGER initially 10 rule parRule = par   x:=y   y:=x endpar </pre>	<pre> int x[2]; int y[2]; void parRule(){   x[1]=y[0];   y[1]=x[0]; } </pre>	<pre> void UpdateState(){   x[0]=x[1];   y[0]=y[1]; } </pre>

**Table 7.2:** *Parallelism: translation in C++*

**Nondeterminism** Nondeterminism in UASM is implemented in either the ChooseRule and PickTerm, where a random element is picked out from a certain EnumerableTerm (set of terms of the same nature). This random choice is performed by generating a random index number that corresponds to the index of the element to be picked inside the EnumerableTerm. Arduino provides functionality for pseudo-random number generation. Functions `random(max)` and `random(min,max)` generate a pseudo random number with uniformly distributed probability within the interval `[min .. max)`. `randomSeed` function must be called at the beginning to make the number generation unpredictable. An example of nondeterminism is shown in Table 7.3.

**Domain Definition** Domains implementing mathematical set (or a subset of them) are implemented using the corresponding data type. These implementations of

UASM	<pre> domain D2 of INTEGER initially {5,9,12} rule chooseRule =   choose x in D2 with x&gt;7 do ruleChoose ifnone ruleIfnone </pre>
C++	<pre> typedef int D2; std::set&lt;D2&gt; D2_elems = {5,9,12} void chooseRule() { std::vector&lt;decltype(D2_elems)::value_type const*&gt; point0; for(auto const&amp; x : D2_elems) if(x&gt;7){   point0.push_back(&amp;x); } int rndm = random() % point0.size(); {   auto x = *point0[rndm];   if(point0.size()&gt;0){     ruleChoose();   } else     ruleIfnone(); } } </pre>

**Table 7.3:** *Nondeterminism: translation in C++*

the mathematical sets is named *BasicDomain* which includes NUMBER, INTEGER, STRING, CHAR, BOOLEAN and RULE. As described in Sect. 5.3, besides BasicDomains there are StructuredDomains, EnumerativeDomains and ExtendableDomains. For StructuredDomains there is a porting of the STL library for Arduino<sup>8</sup> which provides structured data types. Respectively list, set, bag and maps are mapped to the STL equivalents `std::list`, `std::set`, `std::multiset`, `std::map`. EnumerativeDomains are directly mapped to C++ with `enums`. For ExtendableDomains the machine must keep trace of the elements included in the domain and every time an element is added, the include set must be extended. For this reason a set of elements is assigned to each ExtendableDomain. Some examples of translation are shown in Table 7.4.

**Rule** Rules are translated into class methods where every rule is implemented using the C++ basic constructs. The evaluation strategy adopted in ASMs is the pass-by-name, while in C++ the strategy is pass-by-value. Now to guarantee the consistency, we apply the translation only rule calls taking as parameters location variables that are not updated in the rule. The translation of rules into C++ is shown in Table 7.5.

**Terms** In UASM different kind of terms are implemented (see Sect. 5.5), while in imperative language such as C++ only few of them are present. The problem is how to translate UASM terms which are not implemented in C++. For example, consider *CaseTerm*, that depending on the value assumed by a specific term returns a different value. The first approach has been to translate it into the C++ switch-case construct, but in this case the term is translated into rule,

<sup>8</sup>The porting of the STL library it's available at <https://github.com/rpavlik/StandardCplusplus>



UASM	Translation in C++
Basic Domains: corresponding data type in C++	
NUMBER	float
INTEGER	int
STRING	String
CHAR	char
BOOLEAN N	boolean
RULE	Not supported
Structured Domains: STL library	
SET	std::set
BAG	std::multiset
LIST	std::list
MAP	std::map
Enumerative Domains	
enum	enum
Extendable Domains arity 1: implemented with typedef	
domain D1 of INTEGER	typedef int D1
domain D2 of INTEGER initially {5,9,12}	typedef int D2; std::set<D2> D2_elems = {5,9,12}
domain D3 of (x in INTEGER)	Not implemented
domain D4 of (x in INTEGER) initially {5,9,12}	Not implemented
Extendable Domains arity N: the parameters are implemented as tuple	
domain D1 of (INTEGER, INTEGER)	typedef boost::tuple<int,int> D1
domain D2 of (INTEGER, INTEGER) initially {{5,9},{12,10}}	typedef boost::tuple<int,int> D2; std::set<D2> D2_elems = [ ] { std::set<D2> supp = make_set( boost::make_tuple(5,9); boost::make_tuple(12,10); ) return supp; }();
domain D3 of (x in INTEGER, y in INTEGER)	Not implemented
domain D4 of (x in INTEGER, y in INTEGER) initially {{5,9},{12,10}}	Not implemented

Table 7.4: Domain definition: translation from UASM to C++

violating the one-to-one mapping design goal. A better approach is using lambda functions. A lambda function is a function that can be write inline in the source code and call directly without declare it. The main advantages are the flexibility, any kind of function can be defined and the one-to-one mapping from UASM to C++. The disadvantages of this approach are the worsening of readability and efficiency. The structure of lambda function declaration is shown in Code 7.4, while the translation of the terms is shown in Table 7.6.

```
[&]() {
    //place your code here
    return <some_value>;
}
```

**Code 7.4:** *Example of lambda function*

**FunctionTerm** FunctionTerm, depending on the type and on the arity will result in a different translation. Static function with arity zero is translated into constant value, while with non-zero arity is translated into constant array. Dynamic function with zero arity is translated as variable, while function with non-zero arity are translated into map with a n-tuple as key. Derived function is translated into method with zero parameter if the function has zero arity, with n parameters otherwise.

**StructuredTerm** StructuredTerms are identified by comma-separated elements enclosed by brackets inside a UASM model. In C++ these terms are implemented with the container class of the STL library. Unfortunately, there is no standard notation to refer to them in C++ as for UASM language. For this reason structured terms are implemented with lambda functions that create and return the desired term containing the elements. An example is shown in Code 7.5, a *SET* of values is assigned to the output function defined in [SET(INTEGER)] domain.

---

```
out myset of SET(INTEGER)
...
myset := {1, 2, 3}

std::set<int> myset;
...
myset= [](){
    auto x = {1,2,3};
    std::set<int> s(x.begin(),x.end());
    return s;
}();
```

---

**Code 7.5:** *Translation of a Structured Term*

### 7.5.2 Hardware configuration

**Binding** Bindings describe the association between ASM functions and physical pins. Depending on the function domain and the pin type, different binding mode can be realized. Here there is a list of the possible binding mode that are supported:

- *Digital* a binary function is mapped on the HIGH/LOW level of the digital pin.
- *DigitalInverted* the same as Digital, but HIGH and LOW level are inverted.
- *AnalogLinearIn* reads the analog value from the analog port and rescales it over the function domain.
- *AnalogLinearOut* reads the analog value from the analog port and rescales it from the function domain and sets to the analog port.
- *PWM* reads the analog value from the analog port and sets it as PWM modulation of the analog value.
- *UserDefined* none of the above mode was appropriate, the binding is subsequently defined by the user.

**Step time** A limitation about ASM is time modelling because an ASM evolves in discrete steps. However there is no exact correspondence between a machine step and the time. It may be that each step corresponds to an infinitesimal of second, or a fixed timespan for discrete systems. The *stepime* parameter defines the time frame for each machine step.

**Configuration File** In order to generate the complete Arduino project, the information about hardware configuration have to be stored in a configuration file. The file (*.u2c* extension) is automatically generated by the tool with some predefined values, however it must be completed by the user in *User change* step.

### 7.5.3 The Asm2C++ Eclipse Plugin

The *Asm2C++* Eclipse Plugin is realized as push-button transformation, this means that the transformation process, except for the *User Change* step, is completely automatic. The plugin project requires two subprojects: the *uasm2code\_cppgen* and the *uasm2code\_hw*. The first one is responsible to generate the C++ code, while the latter to generate the hardware part. Moreover, the *uasm2code\_hw* is responsible also for automatically generating the configuration file template. Before the execution on the hardware or on the emulator the configuration file has to be completed by the user. Figure 7.9 shows the screenshot of the plug-in. The red circle indicates the transformation button. On the left it is shown the UASM specification, and on the right (a part of) the generated C++ code.

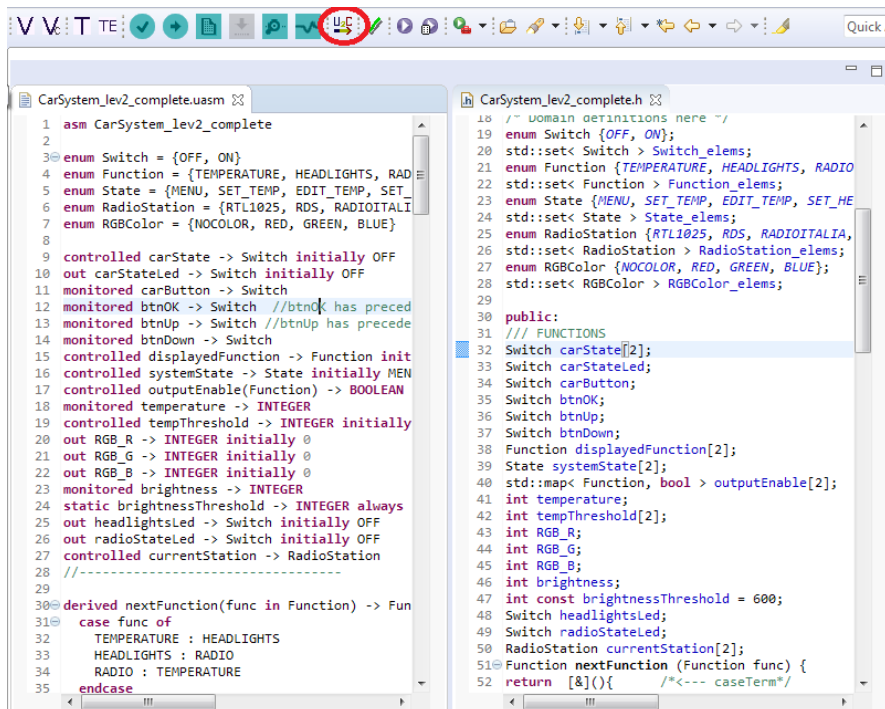
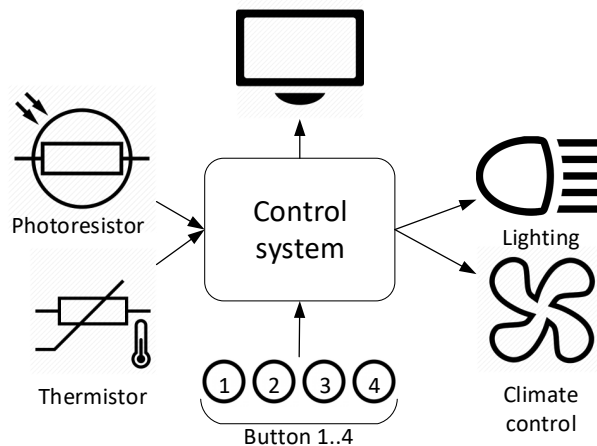


Figure 7.9: Plug-in screenshot

## 7.6 Illustrative example

In this section a case study is presented to show a concrete application of the tool. The purpose is to demonstrate the functionalities by using UASM constructs and Arduino features. The case study is a control system installed in a medical laboratory that contains critical materials. The temperature and the lighting must be controlled otherwise the material inside is compromise (the architecture is shown in Figure 7.10).



**Figure 7.10:** *Control system architecture*

The functionalities implemented by the control system are:

- Switch on/off the system
- Climate control
- Automatic lighting control

Initially the system is off and can be turned on by pressing button 1. Once the system is on, the panel displays the menu which is composed by two voices: climate control and lighting control. The user navigates through the menu voices by pressing button 2 (up) and 3 (down). Once the desired function is reached, the user selects it by pressing button 4. Depending on the selected voice, the corresponding operation is performed. If the user chooses **climate control**, the measured temperature is shown. The user can edit the desired temperature by pressing button 4. At this point, the desired temperature is shown and can be edited using buttons 2 and 3 (respectively up and down). Finally, the value is set by pressing button 4 and the system returns to the menu. Once the climate control is activated, if the measured temperature is greater than the threshold the RGB LED<sup>9</sup> becomes red, if the temperature is less than the threshold the led becomes blue, otherwise the led becomes green. The user can deactivate the climate control by selecting again “climate control” voice from the menu. The display shows a

<sup>9</sup>The led replaced the heat pump behaviour. If the led is green, the heat pump maintains the temperature; if the led is red the heat pump cools the environment; if the led is blue the heat pump heats the environment.

confirmation message, if the user press “yes” (button 2) the control is deactivated, if the user press “no” (button 3) the control remains active and it is possible to edit the desired temperature. If the user chooses **automatic lighting control**, the lighting measured by the photoresistor is shown. The user can edit the desired lighting by pressing button 4. At this point, the desired lighting is shown and can be edited using buttons 2 and 3 (respectively up and down). Finally, the value is set by pressing button 4 and the system returns to the menu. Once the automatic lighting control is activated, if the measured lighting is greater than the threshold the RGB LED<sup>10</sup> becomes red, if the lighting is less than the threshold the led becomes blue, otherwise the led becomes green. The user can deactivate the lighting control by selecting again “automatic lighting control” voice from the menu. The display shows a confirmation message, if the user press “yes” (button 2) the control is deactivated, if the user press “no” (button 3) the control remains active and it is possible to edit the desired lighting. During the operation the following properties must be verified on the model:

- All operations are enabled only if the system is turned on.
- If the temperature/lighting is below the threshold, the corresponding RGB LED is blue (only if the climate/lighting control is enabled)
- If the temperature/lighting is equal to the threshold, the corresponding RGB LED is green (only if the climate/lighting control is enabled)
- If the temperature/lighting is above the threshold, the corresponding RGB LED is red (only if the climate/lighting control is enabled)
- It is possible to deactivate climate/lighting control only if it was activated in the previous step

Subsequently, the process shown in Chapter 4 is applied to the case study. The Arduino code is obtained using code generation plug-in instead of derive the model manually from the specification or catch the code externally.

### 7.6.1 Modelling

Starting from the ground model, through two steps of refinement, the final model is derived. The ground model (see Code 7.6) specification switches on and off the system and its state is described by a binary function (ON/OFF). In the first refinement step (see Code 7.7) the menu is modelled, it is accessible only when the system is turned on. The user navigates through the menu voices and selects the desired functionality. The rule executed when the corresponding voice is selected, is implemented in the next refinement step, at the moment no actions (skip rule) are implemented.

The second (see Code 7.8) and the last refinement (see Code 7.9) steps implement climate control and automatic lighting control. When the user chooses the operation, the corresponding rule is activated as described in the Sect. 7.6.

### 7.6.2 Validation & Verification

At each step of refinement validation and verification activities are performed. Different scenarios have been executed, an example is shown in Code 7.10. The

<sup>10</sup>The led replaced the adjustable lights behaviour. If the led is green, the lights maintain the current lighting; if the led is red the lights decrease the lighting; if the led is blue the lights increase the lighting

```

enum Switch = {OFF, ON}
controlled controlState -> Switch initially OFF
rule r_Main =
  if controlState = OFF then
    r_SwitchOnSystem
  else
    if controlState = ON then
      r_SwitchOffSystem
    else
      skip
    endif
  endif
endif
...

```

---

Code 7.6: *Ground model*

<pre> enum Switch = {OFF, ON} controlled controlState -&gt; Switch   initially OFF enum State = {MENU, SET_TEMP, EDIT_TEMP,   SET_LIGHTS, EDIT_LIGHTS} monitored btnOK -&gt; Switch monitored btnUp -&gt; Switch monitored btnDown -&gt; Switch controlled systemState -&gt; State   initially MENU  rule r_Main =   if controlState = OFF then     r_SwitchOnSystem   else     if controlButton = ON then       r_SwitchOffSystem     else       par         r_Menu         r_SetLights         r_SetTemperature         r_EditTemperature         r_EditLights </pre>	<pre>         r_manageLightsRGBLed         r_manageTemperatureRGBLed       endpar     endif   endif  rule r_Menu =   if systemState = MENU then     if btnOK = ON then       r_selectFunction     else       if btnUp = ON then         ...       else         if btnDown = ON then           ...         endif       endif     endif   endif  rule r_SetTemperature =   skip </pre>
---	--

---

Code 7.7: *First refinement*

scenario turns on the system and scrolls the menu by pressing two times the button 2 and observes the displayed function changing consequently. At the end of the test, the displayed function is the same as the one shown at the beginning.

All the properties are verified as soon as possible in the refinement steps; all of them are verified. For example, the property “ If the temperature/lighting is below the threshold, the corresponding RGB LED is blue (only if the climate/lighting

---

```

monitored temperature -> INTEGER
controlled tempThreshold -> INTEGER
                           initially 20

rule r_SetTemperature =
if systemState = SET_TEMP then
if not(outputEnable(TEMPERATURE)) then
  if btnOK = ON then
    systemState := EDIT_TEMP
  endif
endif
else
  if btnUp = ON then
    par
      outputEnable (TEMPERATURE) :=
                           false
      systemState := MENU
    endpar
  endif
endif
endif

```

Code 7.8: *Second refinement*


---

```

rule r_EditTemperature =
if systemState = EDIT_TEMP then
if btnOK = ON then
  par
    systemState := MENU
    outputEnable (TEMPERATURE) :=
                           true
  endpar
else
  if btnUp = ON then
    tempThreshold := tempThreshold + 1
  else
    if btnDown = ON then
      tempThreshold := tempThreshold - 1
    endif
  endif
endif
endif

```

Code 7.9: *Third refinement*


---

```

scenario navigatemenu
load ControlSystem_complete.asm

  set controlButton := ON;
  step
  check systemState = ON;
  set controlButton := OFF;
  step
  check displayedFunction = TEMPERATURE;
  set btnUp := ON;
  step
  check displayedFunction = LIGHTS;
  set btnUp := ON;
  step
  check displayedFunction = TEMPERATURE;

```

Code 7.10: *Navigate menu scenario*

control is enabled)” is translated in LTL formula as:

---


$$g(((\text{outputEnable(TEMPERATURE)} = \text{true}) \text{ and } (\text{temperature} < \text{temperatureThreshold})) \text{ implies } (\text{RGB.R} = 0 \text{ and } \text{RGB.B} = 255 \text{ and } \text{RGB.G} = 0))$$


---

### 7.6.3 Hardware

In addition to the model, a hardware prototype of the system has been realised to verify the correct functioning of the generated code.

### Components

The components required for inputs are:

- 4 buttons
- 1 photo-resistor for brightness measurement
- 1 thermistor for temperature measurement

The components required for outputs are:

- 1 LCD (Liquid Cristal Display) to show menu
- 1 LEDs for the system state
- 1 RGB LED for climate and lighting control

Other components:

- 1 potentiometer to adjust LCD contrast
- 2 resistors (220  $\Omega$ ) for LED current regulation
- 2 resistors (2  $k\Omega$ ) to calibrate photo-resistor and thermistor
- 4 pull-down resistor (1  $k\Omega$ ), one for each button
- as many wires/jumpers as needed

### HW Integration

Once the model is written, verified and tested, and the hardware is ready to be used, the last thing to do before code generation is to configure the integration file. A base configuration is automatically generated by the plugin, but it needs to be edited to reflect the actual hardware configuration. In addition to bindings, another parameter to be defined is the step time. A too low value would result in a continuously refreshing display, while a too high value would lead to a considerable lag during the user interaction. In this example the value inserted is 200ms, which is fast enough for the user and sufficiently slow for the display. The resulting configuration is shown in Code 7.11.

---

```
{
  "arduinoVersion": "UNO",
  "stepTime": 200,
4  "bindings": [
    {
      "mode": "ANALOGLINEARIN",
      "function": "brightness",
      "pin": "A0",
9     "minval": 0,
      "maxval": 1023
    },
    {
      "function": "temperature",
      "pin": "A1",
      "minval" : 0,
      "maxval" : 1023
    },
    {
      "mode" : "DIGITAL",
      "function" : "btnUp",
      "pin" : "A2"
10  },
    [.. omitted ..]
  ]
}
```

---

Code 7.11: HW configuration

### 7.6.4 From UASM to C++ code

The last step is the code generation [47]. A simple example of the code generation from UASM to C++ is shown in Code 7.12-Code 7.15. It focuses on turn



```

asm ControlSystem
enum Switch = {OFF, ON}
controlled systemState -> Switch
initially OFF
monitored controlButton -> Switch
...

rule r.Main =
if systemState = OFF then
r_SwitchOnSystem
else if controlButton = ON then
systemState := OFF
else
par
r_Menu
r_SetLights
r_SetTemperature
...
endpar
endif
endif
...

```

Code 7.12: *UASM*

```

class ControlSystem{
enum Switch {OFF, ON};
Switch systemState[2];
Switch controlButton;
public:
void getInputs();
void r_Main();
void updateState();
void setOutputs();
...
};

```

Code 7.14: *ControlSystem.h*

```

#include "ControlSystem.h"

// main rule
void ControlSystem::r_Main(){
if (systemState[0] == OFF)
r_SwitchOnSystem();
else if (controlButton == ON)
systemState[1] = OFF;
else{
r_Menu();
r_SetLights ();
r_SetTemperature();
...
}
}

// apply the update set
// to the current state
void ControlSystem::updateState(){
systemState[0]=systemState[1];
}
...

```

Code 7.13: *ControlSystem.cpp*

```

#include "ControlSystem.h"
ControlSystem controlSystem;
...
void loop(){
controlSystem.getInputs();
controlSystem.r_Main();
controlSystem.updateState();
controlSystem.setOutputs();
}

```

Code 7.15: *ASM runner*Figure 7.11: *Snippets from model and code*

on and turn off the system. The ASM is translated in the `ControlSystem` class, where domains, functions and rules become respectively data types, properties and methods. As shown in Code 7.15, the runner cyclically calls four `ControlSystem` methods: 1. Acquire inputs from sensors (`getInputs`) 2. Perform the main rule (`r_Main`) 3. Update the ASM state (`updateState`) 4. Set outputs to actuators (`setOutputs`). Parallel execution is translated as described in [114], where controlled functions are duplicated and the state is updated after the main rule.

Once the code is generated, in order to load the program into the Arduino board, the user must press the "Load" button of the Arduino Eclipse plugin and wait few seconds to have the code built and uploaded into the Arduino code memory.

## Chapter 7. Automatic Code Generator

UASM	Translation in C++
Rule Definition	
<code>rule newrule (x in INTEGER) =</code>	<code>void newrule (int x) {}</code>
Update Rule	
<code>x := 15</code>	<code>x = 15;</code>
Conditional Rule	
<code>if x&gt;0 then   ruleIf [else   ruleElse] endif</code>	<code>if (x&gt;0) then {   ruleIf(); }[else {   ruleElse(); }]</code>
Case Rule	
<code>case color of   BLUE : ruleBlue   RED : ruleRed otherwise   ruleOtherwise] endcase</code>	<code>if (color==BLUE) {   ruleBlue(); } else if (color == RED) {   ruleRed(); } else {   ruleOtherwise(); }</code>
ForAll Rule	
<code>enum EnumDom = { AA , BB } forall x in EnumDom , y in EnumDom   with x = AA do ruleDo</code>	<code>for(auto x: Enum&lt;EnumDom&gt;()) {   for(auto y: Enum&lt;EnumDom&gt;()) {     if(x==EnumDom::AA) {       ruleDo();     }}}</code>
While Rule	
<code>while x&gt;y do ruleWhile endwhile</code>	<code>while (x&gt;y) {   ruleWhile(); }</code>
Call Rule	
<code>ruleCalled(x in INTEGER, y in INTEGER)</code>	<code>ruleCalled(int x, int y);</code>
Skip Rule	
<code>skip</code>	<code>;</code>
PrintRule Rule	
<code>print (x)</code>	<code>println(x);</code>
Seq Rule	
<code>seq   rule1   rule2 endseq</code>	<code>{   rule1();   rule2(); }</code>
Let Rule	
<code>let x=5 in ruleLet endlet</code>	<code>{   auto (x=5);   ruleLet(); }</code>
Extend Rule: Not Implemented	
Iterate Rule: Not Implemented	
Import Rule: Not Implemented	
TurboReturnRule Rule: Not Implemented	
LocalRule Rule: Not Implemented	

**Table 7.5:** Rules: translation from UASM to C++

## 7.6. Illustrative example

UASM	Translation in C++
Conditional Term	
<pre>if x&gt;0 then   true [else   false] endif</pre>	<pre>(x&gt;0) ? true : false</pre>
Case Term	
<pre>case color of   BLUE : 0   RED : 1 otherwise   -1] endcase</pre>	<pre>[&amp;]() {   if (color==BLUE) {     return 0;   } else if (color == RED) {     return 1;   } else {     return -1;   } }()</pre>
Tuple Term	
<pre>(10,15,20,25)</pre>	<pre>boost::make_tuple(10,15,20,25)</pre>
ForAll Term	
<pre>enum EnumDom = { AA , BB } forall x in EnumDom holds x = AA</pre>	<pre>[&amp;]() {   for (auto x: Enum&lt;EnumDom&gt;()) {     if (!(x==EnumDom::AA))       return false;   }   return true; }()</pre>
Exist Term	
<pre>enum EnumDom = { AA , BB } exist x in EnumDom with x = AA</pre>	<pre>[&amp;]() {   for (auto x: Enum&lt;EnumDom&gt;()) {     if ((x==EnumDom::AA))       return true;   }   return false; }()</pre>
Let Term	
<pre>let x=5 in f(x) endlet</pre>	<pre>[&amp;]() {   auto (x=5);   f(x); }()</pre>
Size Of Enumerable Term	
<pre> containerSet  or  containerBag  or  containerMap  or  containerList </pre>	<pre>containerSet.size() or containerBag.size() or containerMap.size() or containerList.size()</pre>
<pre> newDomain </pre>	<pre>newDomain_elems.size()</pre>
Rule As Term: Not Implemented	
Return Term: Not Implemented	
Comprehension Term: Not Implemented	

**Table 7.6:** *Terms: translation from UASM to C++*





**Part III**  
**Case studies**



---

## 3D4Amb: diagnosis and treatment for visual diseases

---

3D4AMB project aims at developing software for diagnosis and treatment of visual diseases. In particular, the focus is on amblyopia, a visual disease that affect young children (see Section 8.1). Other tests have been developed for the measurement of the strabismus and the aniseikonia (the eyes perceive different size of images). In the next sections the project is furthermore explained and in Section 8.3 an application of the process shown in Chapter 4 is applied to the software to perform the diagnosis of amblyopia. This application is not critical (considering the classification of the standard IEC 62304) but it is the first case studies where we applied the ASM process to a medical software using *Asmeta* framework.

### 8.1 Stereoacuity test

---

Having two eyes, as human beings and most animals, located at different lateral positions on the head allows binocular vision. It permits for two slightly different images to be created that provide a means of depth perception. Through high-level cognitive processing, the human brain uses binocular vision cues to determine depth in the visual scene. This particular brain skill is defined as *stereopsis*. Some pathologies, such as blindness in one eye and strabismus, cause a total or partial stereopsis absence. The examination of stereopsis ability can be evaluated by measuring stereoscopic acuity. Stereoscopic acuity, also named stereoacuity, is the smallest detectable depth difference that can be seen by someone with normal two eyes and brain functions. Testing the total or partial loss of stereovision can lead to the detection of visual diseases like *amblyopia*. *Amblyopia*, otherwise known as ‘lazy eye’, is reduced visual acuity that results in poor or indistinct vision

in one eye that is otherwise physically normal. It may exist even in the absence of any detectable organic disease. Amblyopia is generally associated with a squint or unequal lenses in the prescription spectacles. This low vision is not correctable (or only partially) by glasses or contact lenses. Amblyopia is caused by media opacity, strabismus, anisometropia, and significant refractive errors, such as high astigmatism, hyperopia, or myopia. This condition affects 2-3% of the population, which equates to conservatively around 10 million people under the age of 8 years worldwide [131]. If amblyopia is not diagnosed and treated in the first years of life, the lazy eye becomes weaker and the normal eye becomes dominant. Children who are not successfully treated when still young (generally before the age of 7) will become amblyopic adults. The projected lifetime risk of vision loss for an individual with amblyopia is estimated around at least 1-2% [111]. For these reasons, screening for amblyopia in early childhood is done in many countries to ensure that affected children are detected and treated within the critical period. The main goal is to help children to achieve a level of vision in their amblyopic eye that would be useful should they lose vision in their non-amblyopic eye later in life. Stereoacuity is the smallest disparity that can be seen in binocular vision and it is measured in second of arc (arcsec). Stereoacuity tests can be divided into two groups: random dot stereotests and contour stereotests. Random dot stereotests are based on dots patterns arranged randomly with lateral disparity. These tests do not allow monocular vision. TNO, stereotest Lang I and II belong to this group. TNO is a test based on set of sheets with red and green random dots. Using red and green eyeglasses the patient is able to see 3D pictures if he is healthy. This test measures stereoacuity level from 2000 arcsec to 15 arcsec. Lang I and II are based on random dots and they do not need glasses because the dots are displaced to create disparity. These tests are mostly used in young children, but the lower measurable stereoacuity level is 550 arcsec in Lang I and 200 arcsec in Lang II. Contour stereotests are made up by two recognizable images, stagger, and shown to patients using polarized or anaglyph glasses. This category is not sensitive as the former since the images are recognized also monocularly. Titmus stereotest belongs to contour stereotest and is based on vectographic technique. This test measures three levels of stereoacuity: high level around 3600 arcsec, medium level from 400 arcsec to 100 arcsec and low level upto 40 arcsec. There is a test that is a joint of random dot stereotests and contour stereotests, i.e., randot test. It is composed by two tables, the first is based on contour stereotest but background is replaced with dots. This technique prevents the monocular vision. Using this first table the measurable stereoacuity level is from 400 arcsec to 20 arcsec. The second table is based on random dots and it measures stereoacuity from 500 arcsec to 250 arcsec. All these tests can generate a false negative because they have the following disadvantages:

- Shown images are always the same, so the patients can memorize them and give right answers even if they do not see the right image.
- Children can be helped by parents or doctors, so the test is not truthful.

In order to reduce the number of false negative, the policies devised are:

- the shape is randomly chosen every time;
- the user that delivers the test has no clue about which shape is currently



Applications		Technologies		
		3D Active	Anaglyph	CardBoard
Tests	Stereo Acuity Test	X	X	X
	Lancaster/Hess		X	
	Aniseikonia		X	
Treatments	Videogames	Space Invaders	Tetris	Car Racing Cardboard Tetris
	Video Rebalancing	X		X

**Table 8.1:** 3D4AMB applications and technologies

displayed.

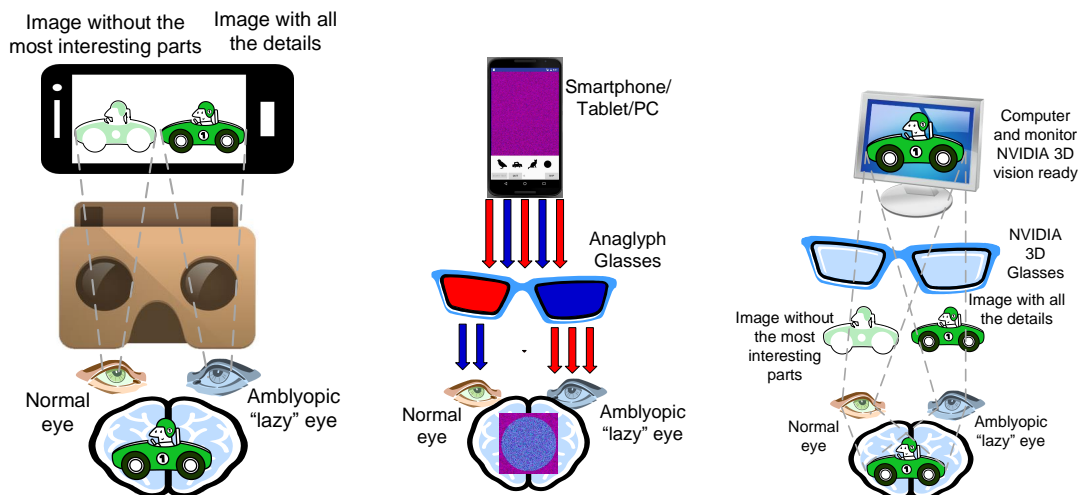
For these reasons, the project 3D4AMB<sup>1</sup> works on developing an efficient and affordable tests and treatments [66], using for instance personal computer and stereoscopic 3D technology as in [59, 61, 62, 130] that permits to show different images to the amblyopic eye and the normal eye.

## 8.2 3D4Amb projects

Currently, the 3D4AMB project uses different 3D technologies and for each technology different applications have been developed (see Table 8.1).

The technologies used in 3D4AMB project are:

- 3D Active
- Anaglyph
- CardBoard



(a) Principle of operation Cardboard (b) Principle of operation Anaglyph (c) Principle of operation 3D Active

**Figure 8.1:** 3D technologies

<sup>1</sup><http://3d4amb.unibg.it/>

### **3DActive**

3D Active (see Figure 8.1c) is based on NVIDIA® 3D Vision™ technology. It requires a standard personal computer with a NVIDIA graphic card (also entry level NVIDIA graphic boards work) and a monitor 3D Vision ready with a refresh rate of 120 Mhz as well as a NVIDA 3D glasses. The monitor alternates images for the two eyes and the glasses are able to synchronize the display of the images shown by the monitor through the synchronized emitter to be connected either PC USB port or directly to the graphic board.

### **Anaglyph**

Anaglyph (see Figure 8.1b) is based on glasses lenses using filters of chromatically opposite colours, typically red and cyan. The images contain three different colours: red, cyan and one visible by both eyes. Common dots are used to merge the images seen by left and right eye. Red dots are seen by the eye with cyan lens, while cyan dots are seen by the other eye.

### **Cardboard**

Cardboard (see Figure 8.1a) is a simple box with two lenses that used in combination with a smartphone, constitutes a simple yet powerful virtual reality viewer. The smartphone must be inserted in the box and the user looks inside in order to see the images displayed by the phone. It permits a stereo vision by sending two different images to the two eyes. It works with different smartphones and can be easily adapted to be used by children.

### **Diagnosis and Treatments**

The applications are divided in two categories: diagnosis and treatments. Diagnosis are applications that verify if patient suffering from diseases, treatments are applications used to cure the diseases. Stereoacuity test measure the visual acuity, the total or partial lost of acuity cause diseases like amblyopia. This test has been developed for 3D active (Figure 8.2a), anaglyph glasses (Figure 8.2b) and cardboard [40] (Figure 8.2c)<sup>2</sup>. The Lancaster/Hess test (see Figure 8.3a) measures strabismus in the nine diagnostic positions of gaze, the result is a grid (one for each eye) that shown the mobility of the eye in different position (see Figure 8.3b). Aniseikonia (see Figure 8.3c) measures if there is a significant difference in the perceived size of images, the difference is measure in percentage respect to the size perceived by the other eye. These two test have been developed only for anaglyph technology.

Treatments are applications that help people to cure the diseases, in particular our applications cure the amblyopia. 3D system provides different images to the two eyes of the same scene with viewing angles slightly out of phase, that correspond to the different points of view of left and right eye. For treatment applications, the principle of the system that shows different images to the two eyes is used. Two different images are sent to the two eyes: to the amblyopic eye we will show

---

<sup>2</sup>Note that the images shown in Figure 8.2 have picture guess coloured otherwise it would not be possible to see it.

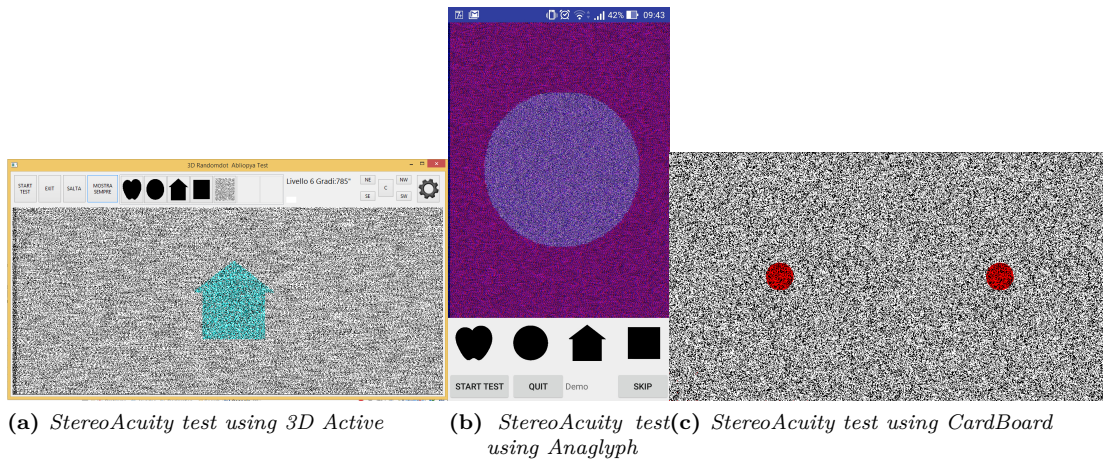


Figure 8.2: StereoAcuity tests

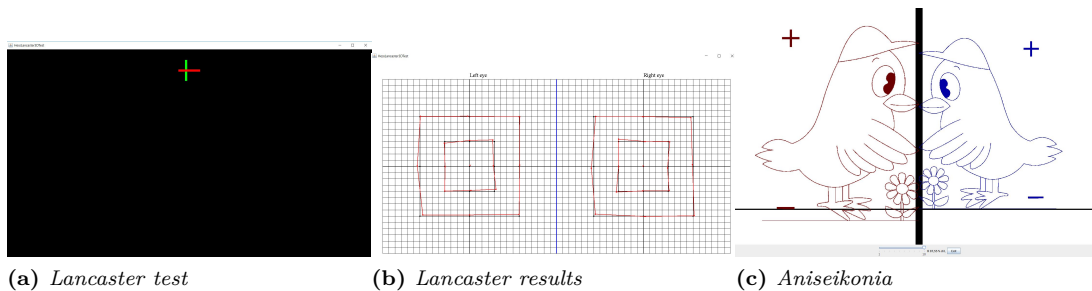
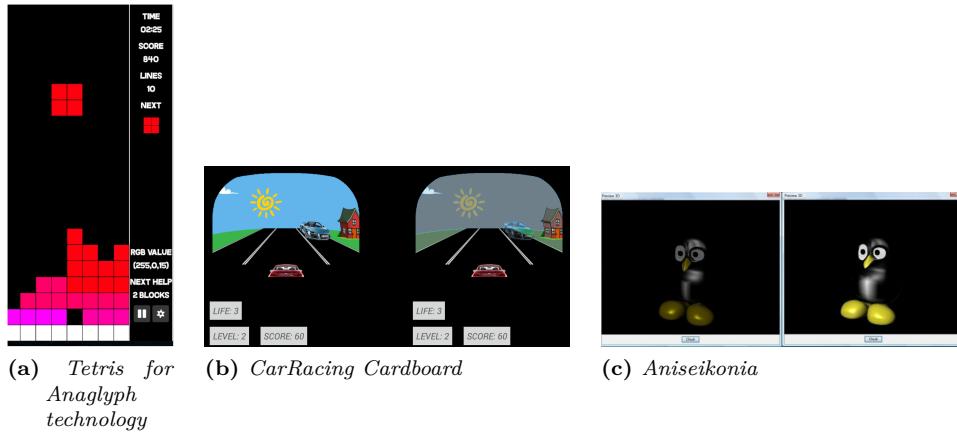


Figure 8.3: Lancaster test and Aniseikonia measurement

the most interesting part of the frame of the clip (or game), while to the not amblyopic eye (or good) we will show the least interesting part. In this way, the user uses more the amblyopic eye to piece together the complete scene. For 3D Active, two applications have been developed: Space Invaders and video rebalancing (see Figure 8.4c). The first is a game and the application shows to the amblyopic eye the more interesting parts needed to win. The second shows e.g. video, film, cartoon and send to the amblyopic eye the complete scene, while to the non amblyopic eye send the scene without all details. The same principle is applied to the version for CardBoard. Tetris has been developed for anaglyph (see Figure 8.4a) and CardBoard. For both technologies the application shows the most important part to the amblyopic eye and some part of the game to the non amblyopic eye. The same principle has been applied to Car Racing Cardboard (see Figure 8.4b).

### 8.3 StereoAcuity Test

StereoAcuity test aims to substitute the standard random dot stereotests. Stereoacuity test has been developed for different platforms: Java applications for pc using 3D Active and anaglyph technologies, Android applications for anaglyph and CardBoard technologies. The goal is to perform stereotests in order to reduce the number of false negative. All these applications share the core component called



**Figure 8.4:** *Treatments for visual diseases*

SAM (Stereoscopic Acuity Measurer). SAM decides the stereo depth of the image to be shown to the patient, when to stop the test, and to provide the final exam result (stereoacuity certification). To certify the patient's stereoacuity, different stereo random dot images are shown at different levels of difficulties. The test starts at level  $N$ , that is the easiest level. Every time a patient recognizes the shown image, the level decreases (i.e., the test becomes more difficult) until it reaches level 1, which is the most difficult level. A level is passed if the patient recognizes three times the shown images. When the patient answers incorrectly, (s)he can try another time at the same level. If (s)he fails again, the level increases (i.e., the test becomes easier), and (s)he cannot be tested at that level any more: (s)he can try to be certified at the upper level. If the patient fails twice the  $N$ th level, the test stops with no certified level. Besides recognizing images, a user can EXIT the test or SKIP an image. The SKIP answer is treated like the wrong answer: if the patient skips twice at the same level, the level increases. The SAM component eventually stops: 1) by certifying the patient at level  $i$ , if the patient has been able to recognize three images at level  $i$ , but has failed at level  $i-1$  (if any); 2) without certification, if the patient has been unable to complete the test or the doctor has quit the exam.

### 8.3.1 Modelling and refinement

The SAM component has been modelled using five levels of refinement. The starting ground model captures the core behaviour of the component; the further requirements have been considered incrementally along the subsequent refined models till a level detailed enough to be checked for conformance w.r.t. the code<sup>3</sup>.

#### Ground model

In the ground model, we abstract from the explicit answers of the patient, and we consider only if (s)he successfully recognizes an image (regardless of the specific image). We, therefore, model the reaction of the software component to a user

<sup>3</sup>Complete models are available at <http://fmse.di.unimi.it/sw/MEMOCODE15.zip>

input which can be a correct/incorrect patient's answer or the doctor request to quit the system without any certification. Therefore, in the ground model, the patient is certified at a given level if (s)he recognizes the image once. The system can be in three different configurations: in state **Test** when the patient is doing the test, in (the final) state **Uncertified** when the patient is not able to complete the test and the system is not able to certify the patient, in (the final) state **Certified** when the patient finishes the test at a given level and can be certified at that level. In state **Test** (also initial), the system checks for the user answer. If the doctor wants to **EXIT**, the test is stopped and the system goes in **Uncertified** state. Otherwise, the system checks if the patient's answer is correct or not. If the answer is **RIGHT** and the patient is in level 1, (s)he has completed the test. The system provides the corresponding certification and moves to **Certified** state. Otherwise, the software checks if the patient has previously failed a level (i.e., a **level** has been **increased before**). If this is the case, the software certifies the patient at the current level and moves to **Certified** state. Otherwise, the level is decreased and the system returns to **Test** state. If the answer is **WRONG** and the patient is doing the test at the maximum level, the system stops the test and moves to **Uncertified** state. Otherwise, the level testing is increased (**Increase level**) and the system returns to **Test** state.

#### First refinement

At this level of refinement, we add the **SKIP** option. At each level the patient can skip the answer once. If (s)he skips the answer twice at the same level, the level is increased. If the user inserts **SKIP** for the first time, the system memorizes that a **SKIP** has been made and returns to **Test** state. In case it is the second skip at the same level, the system behaves as in presence of a wrong answer in the previous model.

#### Second refinement

At this level of refinement, we model the requirement that the patient can give a wrong answer only once; thus, on the second mistake at the same level, the level is increased. When the patient gives the **WRONG** answer for the first time at the same level, the system memorizes that an error has been made and returns to **Test** state. Otherwise, if it is the second error at the same level, the system behaves as in presence of a wrong answer in the previous models.

#### Third refinement

At this level of refinement, we specify the requirement that the patient has to recognize the shown image three times at the same level to get certification at that level. When the patient gives the **RIGHT** answer, the system checks for the current level under test. If it is level 1 and the patient has already answered correctly twice (i.e., the current answer is the third correct image recognition), the system issues her/him a certification and moves to the **Certified** state. Otherwise, the number of images recognized at that level is incremented and the patient continues the test. The same behaviour is performed when the patient is not in level 1 but

(s)he has already incremented the level (i.e., (s)he made two skips or two errors in a lower level).

#### **Fourth refinement**

This model refines **RIGHT** and **WRONG** answers with the images name. Images are randomly chosen and the system checks whether the patient correctly recognizes the shown image or not. When the patient answers **SKIP** or the doctor selects the **EXIT** answer, the behaviour is the same as in the previous refinements. In case the answer is an allowed image name, the system checks whether it is the right one. If the answer is correct, the system behaves as previously modelled in case of right answer, otherwise it has the same behaviour as in case of a wrong answer (see previous refinements).

### **8.3.2 Validation & Verification of the SAM**

In this section we apply Validation & Verification activities presented in Chapter 4 to the StereoAcuity test case study.

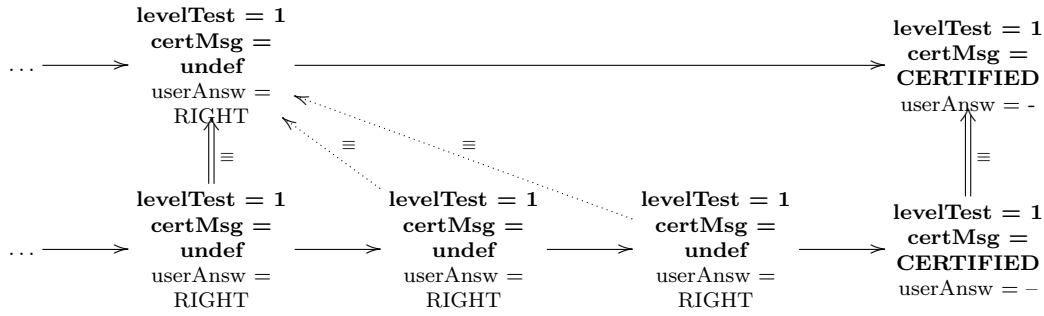
#### **Proving refinement**

Each step of refinement is proved correct [41] using the SMT-based tool **ASM-RefProver**. We check a notion of refinement, called *stuttering refinement*, more restrictive than the definition given in [41]. If we consider as conformance relation between abstract and refined states the equality of functions **levelTest** (representing the current level) and **certMsg** (representing the status of the certification), all our refinement steps are correct. Let us consider the third refinement in which is modelled the fact that the certification is granted only when three images are recognized at the same level. The proof of correct refinement in case of (a) wrong answer, (b) right answer to decrease the level, (c) skip answer, and (d) skip command, is straightforward (since there is surely an abstract run equal to the refined run). It is also shown that the runs in which a given level is certified (i.e., three shown images are recognized) have corresponding runs in the abstract machine. A refined state in which the shown images have been recognized once or twice is stuttering conformant with the abstract state in which no image has been recognized yet. Figure 8.5 shows an example of refined run certifying level 1 and a corresponding abstract run.

Similar arguments for proving refinement correctness can be done for all the refinements. Note that the proof of the refinement correctness is completely automatic: the designer must only indicate which are, in the two models, the “locations of interest” [41] that are involved in the conformance relation.

#### **Simulation**

Simulation is a validation activity by using the **AsmetaS** simulator. It allows interactive simulation and random simulation. In interactive simulation, at each step the user is asked for the values of the monitored functions, whereas in random simulation the simulator itself randomly chooses the values for monitored



**Figure 8.5:** Stuttering refinement from the second refinement model to the third refinement model – Example of refined run

functions. Moreover, the `AsmetaS` simulator allows to check if some invariants are satisfied during simulation. For example, if we consider the following invariant:

---

```
(certMsg=CERTIFIED and not loop) implies
levelCertificate=1
```

---

It states that if the patient has been `CERTIFIED` and (s)he has never increased the level (i.e., function `loop` is false), (s)he is definitely certified at level 1.

### Scenario Validation

Simulation is useful in the earlier stages of model development when it is easy to follow the machine executions due to the compactness of the model. When the ASM becomes more complex and large or the simulations become repetitive, `AsmetaV` is a useful tool to support the user in validation. Such scenarios have been executed whenever the models are modified to check that the behaviour of the ASM was not altered (in a kind of regression testing). Moreover, the user constructs some scenarios for simulating different refined models; indeed, a scenario for an abstract model  $M$  should be correct also for a refined model  $\widetilde{M}$  if the scenario is related to the elements of  $M$  that have not been refined in  $\widetilde{M}$ . For example, the scenario that simulates the certification of first level without any error has to be the same in the ground model, first, and second refinement. Indeed, first and second refinements add the counting of the wrong answers and the possibility of giving the skip answer; however, the handling of correct answers is the same in the three models.

### Model review

*Model review* is an automatic validation technique that verifies some general properties that any model should guarantee such as *completeness*, *minimality*, and *consistency*. The `AsmetaMA` tool [20] allows *automatic* review of ASMs. Typical vulnerabilities and defects that can be introduced during the modeling activity using ASMs are checked as violations of suitable *meta-properties* (*MPs*, defined in [20] as CTL formulae). An inconsistent update (meta-property MP1 in [20]), for example, is a signal of a real fault in the model; the presence of functions that are never read nor updated (meta-property MP7 in [20]), instead, may simply indicate that the model is not minimal, but not that it is faulty. `AsmetaMA` tool found

that, in the fourth refinement, function `expAnsw` (specifying the expected answer) could not take values `SKIP`, `EXIT` (violation of meta-property MP6 in [20]). This violation is not a signal of a real fault in the model, but of a bad choice at design time. Indeed, in the models (starting from the ground model), the same domain `Answers` has been used for modelling the patient's choices (i.e., `RIGHT` and `WRONG` till the third refinement, the image names in the fourth refinement) and the commands to skip a step and to exit the test (i.e., `SKIP` and `EXIT`). In the fourth refinement, the random choice of an image is chosen by the machine and the chosen value is stored in `expAnsw`: since for the function codomain the same domain `Answers` is used for the patient's choice, the model must avoid to select `SKIP` and `EXIT` as chosen images. A better design choice would have been to separately model the available images and the test execution commands. In a preliminary version of our ground model, instead, `AsmetaMA` found a real fault. We wrongly wrote `levelTest < maxLevel` instead of `levelTest > maxLevel` in the guard of a conditional rule and this did not allow the else branch of the conditional rule to be ever executed (violation of meta-property MP3 in [20]). These small examples show that model review is a quite powerful push-button validation activity that can be used since the first stages of model development to find faults (e.g., inconsistencies) and/or stylistic defects (e.g., minimality violations) of our models.

### Property verification

In this section we use model checking (by means of the `AsmetaSMV` tool) for verifying *application-dependent* properties, i.e., properties specific to our case study. Some have been derived directly from the requirements of the system, others have been added during the verification activity for increasing the requirements completeness. We here report some CTL properties common to all models. For each developed model, we have also specified more specific properties regarding the requirements considered by that model. As first property, we check that it is always possible to terminate a test (note that the boolean function `test` is true when the test is running).

---

`af (not test)`

---

Moreover, we check that if the test is terminated, it cannot start again (for us, a test corresponds to a run of the ASM).

---

`ag (not test implies ag (not test))`

---

The decision whether or not to certify a level for a patient can only be made when the test is finished. Therefore, we verify that, during the test, the message containing the certification decision (i.e., function `certMsg`) is undefined and becomes defined when the test is finished.

---

`ag (test implies isUndef(certMsg))`  
`ag (isDef(certMsg) implies not test)`

---

Note that the previous two properties were also specified as invariants for simulation<sup>4</sup>.

---

<sup>4</sup>The tool `AsmetaSMV`, for each invariant  $\varphi$ , automatically creates the CTL property  $\mathbf{AG}(\varphi)$ .



Then we check that both decisions can be taken:

---

```
ef (certMsg = CERTIFIED)
ef (certMsg = NOTCERTIFIED)
```

---

We further check that, once a decision has been taken (either CERTIFIED or (NOT-CERTIFIED)), it cannot be changed:

---

```
ag (certMsg = CERTIFIED implies
ag (certMsg = CERTIFIED))
ag (certMsg = NOTCERTIFIED implies
ag (certMsg = NOTCERTIFIED))
```

---

### 8.3.3 Scenario and test generation

We have exploited model checking tools for identifying interesting runs of the models by introducing *trap properties*, which are not actual system properties to be verified. A trap property has form *never*( $\phi$ ), where  $\phi$  is a predicate over the state that we want to *cover* with a system run, and *never* is translated to a corresponding model checker operator. If a state  $S$  satisfying  $\phi$  exists, the trap property is false and the returned counterexample is a trace leading to  $S$ . Such counterexamples are used in the scenario-based validation for constructing execution scenarios (see Sect. 8.3.2) and in the test framework for testing the implementation (see Sect. 8.3.3). Some trap properties we have specified are:

---

```
never (certMsg = NOTCERTIFIED)
never (certMsg = CERTIFIED and
levelCertificate = 1)
...
never (certMsg = CERTIFIED and
levelCertificate = 6)
```

---

By means of these properties, we are able to generate a scenario (test) in which the exam is terminated without certifying the patient and six different scenarios (tests) in which the patient is certified in one of the six levels.

#### Conformance checking

In this section, it is shown how the formal specifications is used for testing the Java implementation of SAM. First of all, the implementation has to be linked with the formal specification: in order to do this, we use a set of Java annotations originally introduced in the runtime verification framework CoMA [21]. Such annotations link both the data part (some fields and pure methods of the implementation are linked to functions of the specification) and the behaviour (the execution of some selected methods corresponds to a step of the ASM). In the original version of CoMA, only one class can be connected with the formal specification. In this work, CoMA is extended to handle more complex programs: if a program is composed of several classes, the tester writes a wrapper class and connects it with the specification. In our case, SAM is composed of the main `3DStereoLogic` class, and the `DepthCertifier` class, which is internally used by `3DStereoLogic`. A wrapper is used as shown in Figure 8.6. Code 8.1 reports its Java implementation annotated for testing and connected to the last refined model. For example, for the data part, the field `outMessage` is connected to the ASM function `certMsg` (in

the implementation, the certification result is shown as an output message) and the pure method `getCurrentLevel` is connected to the ASM function `levelTest`; for the behavioural part, the execution of method `chooseShape` corresponds to an ASM step.

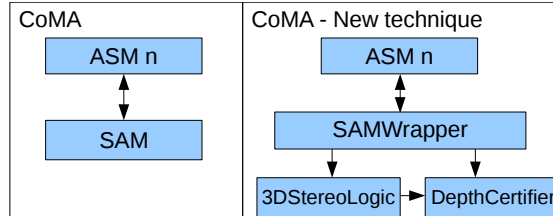


Figure 8.6: Wrap class for testing

```

import org.asmeta.monitoring.*;

@Asm(asmFile = "models/SAM-ref4.asm")
public class SAMWrapper {
    @FieldToLocation(func = "certMsg")
    OutMessage outMessage;
    @FieldToLocation(func = "test")
    boolean test;
    @Monitored(func = "expAnsw")
    ShapeW chosenShape;
    @Monitored(func = "userAnsw")
    AnswersW userAnswer;
    3DStereoLogic logic = new 3DStereoLogic(5);

    @StartMonitoring
    public SAMWrapper() {
        logic.startTest(new ExperimentSetupData(6), false);
        test = (certifier.getMode() == TEST);
    }

    @RunStep
    public void chooseShape() { ... }

    @MethodToFunction(func = "levelTest")
    int getCurrentLevel() {
        return certifier.getCurrentDepth();
    }
}
    
```

Code 8.1: Java code of SAM Wrapper

To produce tests from formal specifications, we use the ATGT tool [64] which is based on the model checker SPIN. The produced tests are *abstract* and they need to be concretised in tests for the implementation (e.g., JUnit tests): in order to do this, we use the technique introduced in [22] that exploits the linking provided by the Java annotations. In a preliminary version of the Java implementation, we found an error related to the counting of the recognized images. The requirements prescribe that a level  $i$  is certified if the patient recognizes three times the shown images, *without changing the level between two recognitions*. Thus, the number of recognized images in a level is reset when the patient change the level. The preliminary Java implementation, instead, did not reset the counters for the different levels: a patient could be wrongly certified at level 2 by recognizing the image at level 2, moving to level 1, failing twice at level 1, and guessing twice at level 2 (i.e., recognizing three times at level 2 but moving to level 1 between two recognitions). We have tested the implementation (whose size is 2.6 KLOCs) with three different test suites: one we generated manually by reasoning on the requirements (*Handmade*), one generated by ATGT using a Breadth-First-Search (*BFS*) approach in generating the tests, and the last one generated by ATGT using a Depth-First-Search (*DFS*) approach. Table 8.2 reports the obtained results in terms of number of tests, global test length, and achieved coverage. The coverage is split between the two main components of the program SAM. We can observe that the handmade test suite has the minimum number of test instructions, but also obtains the lowest coverage. DFS obtains better coverage than

	#tests	total length of tests	Achieved coverage (%)	
			3DStereoLogic	DepthCertifier
Handmade	15	872	62.1	60.8
BFS	14	1181	62.5	69.3
DFS	8	1688	64.4	83.4

**Table 8.2:** *Code coverage*

BFS, although both test suites have been built for covering the same set of test predicates over the ASM; this is due to the fact that a depth-first approach usually builds longer tests that are able to exercise code instructions not reached by the usually shorter tests generated by a breadth-first approach. Finally, we observe that the advantage of using an automatic approach becomes significant when the program under test is particular complex (as the `DepthCertifier` component); for simple programs (as the `3DStereoLogic` component), instead, the handmade approach can still obtain good results. Note that the linking of the implementation with its formal specification can also be used for checking the conformance at runtime [21] (in a kind of *online testing*).



---

## Hemodialysis machine case study

---

The hemodialysis machine case study has been proposed at ABZ 2016 conference<sup>1</sup>. Given the document of requirements and the design of an hemodialysis machine, the goal is to provide a model of the system using formal methods. Starting from the model, validation and verification techniques are applied to guarantee the correctness of the model compared to the document of requirements. In this case study, we have applied the process defined in Chapter 4 using the tools provided by Asmeta framework.

### 9.1 Requirements

---

Kidney diseases are increasing every year due to:

- diabetes
- high blood pressure
- autoimmune diseases
- genetic diseases
- nephrotic syndrome
- urinary tract problems.

Sometimes the kidney can stop working very suddenly. It is called acute kidney injury and its common causes are:

- heart attack
- illegal drug use and drug abuse
- not enough blood flowing to the kidneys
- urinary tract problems.

---

<sup>1</sup><http://www.cdcc.faw.jku.at/ABZ2016/program/>

When the patient is nearing kidney failure, he/she will need dialysis or a kidney transplant to survive. Hemodialysis is a treatment that use a machine to clean the blood. The machine transported the blood to and from the patient and filters wastes, salts and fluid from the blood. The connection between venous and the machine is done with a vein created surgically. Following, the hemodialysis machine architecture and the therapy execution are briefly shown, the complete description can be found in [100].

### 9.1.1 Hemodialysis machine architecture

The hemodialysis machine architecture is shown in Figure 9.1 and it is composed of components described in the next paragraphs.

**Extracorporeal Blood Circuit (EBC)** The extracorporeal system consists of components that transport the blood from and to the patient. Arterial blood pump pumps the blood from the patient to the machine. Heparin pump inject the heparin to avoid the blood coagulation previously the blood enters into the arterial chamber, where it is accumulated before get in the dialyser. When the blood comes out of the dialyser, it is accumulated into venous chamber before pumped it back to the patient. Sooner the blood is reinfused into the patient, the Safety Air Detector (SAD) detects air bubbles in the blood to prevent the patient death.

**Dialyser** The dialyser is a component of the machine that filters the blood to remove waste. It is composed by a set of semipermeable hollow fibres encased in a plastic canister. The dialyser is used to correct the concentration of water-soluble substances in the patient blood before delivering it back. The blood is separated from the dialysing fluid (DF) by a semipermeable membrane that permits a bidirectional diffusive transport and ultrafiltration (UF). The process allows the diffusion of waste from the blood to the DF and the nutrients from the DF to the blood.

**Balance chamber** The balance chamber is a closed system that consists of two chambers, each with a flexible membrane, allowing it to fill the chamber from one side while an identical volume is emptied to the other side. Therefore the outlet fluid volume is equal to the input fluid volume. Each membrane has a magnetic sensor which reads the membrane position and controls the opening and the closing of each sub compartment. The control of the dialysate volume is also carried out by the balance chamber. The difference between used dialysate and fresh dialysate is the ultrafiltration volume, which is removed from the blood side of the dialyser. Ultrafiltrate removal is carried out by the UF pump.

**DF preparation** In bicarbonate dialysis, which is the most common procedure, concentrate preparation consists of mixing the heated and degassed water with bicarbonate concentrate and acid concentrate. The accuracy of dialysate concentration is controlled by conductivity sensors. If the concentration is incorrect, the dialyser will be bypassed.

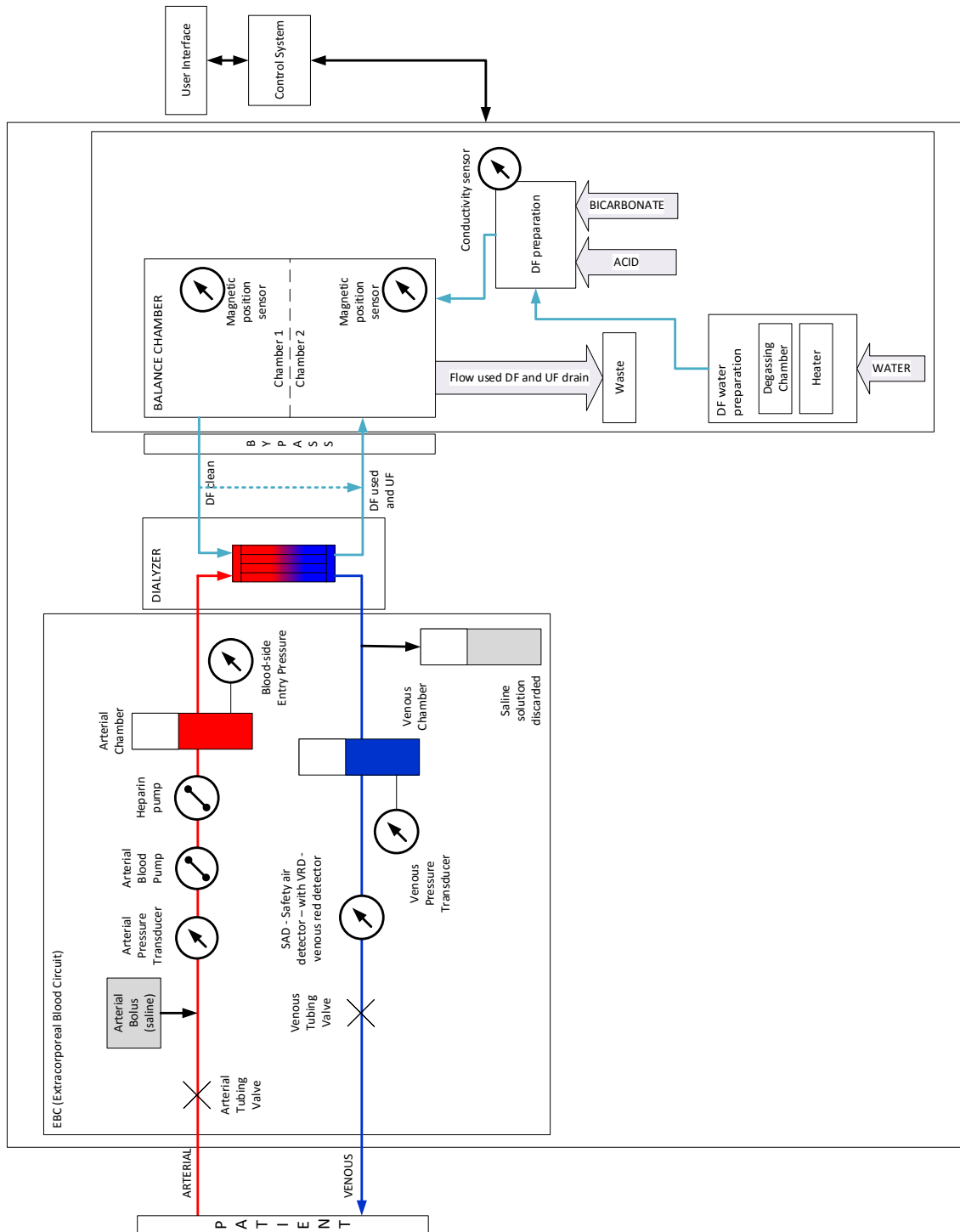


Figure 9.1: Hemodialysis machine: schema

**Water preparation** Purified water, coming from the reverse osmosis system, has to be degassed and tempered to a predetermined temperature, which is set by the user (usually 37°C), before the concentrate is prepared. A degassing chamber and a heater are integrated in the system.

**Bypass** Bypass is activated when the DF conductivity or temperature goes beyond permissible limits. The dialyser is separated from the DF without interrupting DF preparation, when temperature and conductivity are within the limits the bypass is deactivated.

**User interface** The user interface is a display panel that provides communication between the machine and the user. On the display it is possible to visualize all the dialysis parameters and relevant information about the procedure and alarm conditions. By touching the icons on the screen, the user can input all the parameters for the treatment such as: dialysis time, UF volume and heparin pump flow. Several profiles for the procedure can be selected and set via the interface.

**Control system** The control system is divided into two parts: The top level control system connects the interface with the user and transmits data to and from other modules. The low level control system controls and monitors the machine and its functions and also communicates with the top level control system. Both systems operate independently of each other.

### 9.1.2 Hemodialysis therapy

During the therapy the machine extracts the blood through an arterial access. The dialyser separates the metabolic waste products from the blood. At the end the clean blood is pumped back to the patient. A therapy session is divided into three phases: 1. Preparation 2. Initiation 3. Ending .

**Preparation phase** The first operation executed by the machine is an automatic test to check all functions. After that, the concentrate for the therapy is connected and a nurse sets all rinsing parameters. The tubing system is connected to the machine and it is filled with saline solution. Afterwards, the nurse prepares the heparin pump and inserts the treatment parameters. At the end of preparation phase the dialyser is connected to the machine and rinsed with saline solution.

**Initiation phase** During the initiation phase, the patient is connected to the machine. The patient is connected arterially and the tubes are filled with blood until it is detected by VRD (Venous Red Detector) sensor (VRD detects the blood and it is installed with the SAD sensor). Subsequently the patient is connected venously and the therapy can start. During the therapy the blood is cleaned and the following operations are performed:

- monitoring the blood pressure limits
- set the treatment at minimum UF rate
- run heparin bolus
- infuse sodium chloride



- switch the dialysis mode into bypass (the dialyser is temporary disconnected from the DF)

When the therapy is completed an acoustic signal is heard.

**Ending phase** The patient is disconnected arterially and saline solution is infused. When the solution is infused completely, the patient is disconnected venously. After that the dialyser and the cartridge are emptied. Finally, an overview of the therapy is shown.

## 9.2 Modelling by refinement

---

In this section, the application of the ASM-based development process of the Hemodialysis machine case study (HMCS) is exemplified. The ASM method can be used to support in a formal way the activities required by steps 5.3 and 5.4 of the IEC 62304 standard and allows early and continuous V&V activities as required by the FDA principles. In modelling the HMCS, we proceeded through refinement. At the highest level of abstraction, the *ground model* gives the overall abstract view of the whole device that goes through three phases: the PREPARATION of the device, the execution (or INITIATION) of the therapy, and the termination (or ENDING) of the process. Then, each of these phases has been refined. The first refinement regards PREPARATION phase, the second refinement models the device operating in INITIATION phase, and the third refinement specifies the behaviour in ENDING phase. Each refinement step models all the (possible) activities performed in the phase and all the controls with related errors and alarms. A peculiarity of the case study is that the device behaviour is clearly divided in phases and each phase is characterized by different activities (or sub-phases). To identify which activity is currently executed, a function that assumes the value corresponding to the sub-phase is introduced. For example, during the PREPARATION phase the nurse performs different activities and to identify which one is currently executed a function `prepPhase` assumes the corresponding value. If one of these sub-phase consists of a set of actions, a new function is introduced. Table 9.1 summarizes the phases and sub-phases of the case study.

In the next sections each refinement step is shown together with validation and verification activities.

### 9.2.1 Ground model

As said before, the ground model simply describes the transitions between the phases constituting a hemodialysis treatment, without any additional detail.

Code 9.1 shows the ground model written using the `AsmetaL` syntax. Depending on the current `phase`, the machine executes a corresponding rule:

- `r_run_preparation` models all the activities needed to prepare the hemodialysis device for the treatment. This rule is refined in the first refinement step (see Sect. 9.2.2).
- `r_run_initiation` models the hemodialysis therapy, when the patient is connected to the device and her/his blood is cleaned. The rule is refined in the second refinement step (see Sect. 9.2.3).

phase			
PREPARATION	prepPhase AUTO_TEST CONNECT_CONCENTRATE SET_RINSING_PARAM		
	TUBING_SYSTEM	tubingSystemPhase CONNECT_AV_TUBES CONNECT_ALL_COMP SET_SALINE_LEVELS INSERT_BLOODLINES PRIMING CONNECT_AV_ENDS	
	PREPARE_HEPARIN		
	SET_TREAT_PARAM	treatmentParam BLOOD_CONDUCTIVITY BIC_AC BIC_CONDUCTIVITY DF_TEMP DF_FLOW UF_VOLUME THERAPY_TIME MIN_UF_RATE MAX_UF_RATE MAX_AP DELTA_AP PERC_DELTA_TMP LIMITS_TMP MAX_TMP EXTENDED_TMP MAX_BEP STOP_TIME_H BOLUS_VOLUME_H RATE_H ACTIVATION_H SYRINGE_TYPE	
	RINSE_DIALYZER	rinsePhase CONNECT_DIALYZER FILL_ART_CHAMBER FILL_VEN_CHAMBER FILL_DIALYZER	
INITIATION	initPhase		
	CONNECT_PATIENT	patientPhase CONN_ART START_BP BLOOD_FLOW FILL_TUBING CONN_VEN END_CONN	
	THERAPY_RUNNING	therapyPhase START_HEPARIN	arterialBolusPhase WAIT_SOLUTION SET_ARTERIAL_BOLUS_VOLUME CONNECT_SOLUTION RUNNING_SOLUTION
		THERAPY_EXEC	
		THERAPY_END	
ENDING	endingPhase		
	REINFUSION	reinfusionPhase REMOVE_ART CONN_SALINE START_SALINE_INF CHOOSE_NEXT_REINF_STEP RUN_SALINE_INF START_SALINE_REIN RUN_SALINE_REIN REMOVE_VEN	
	DRAIN_DIALYZER EMPTY_CARTRIDGE THERAPY_OVERVIEW		

Table 9.1: Hemodialysis device phases

- `r_run_ending` models the terminal part of the therapy, when the patient is disconnected and the device is cleaned for subsequent treatments. The rule is refined in the third refinement step (see Sect. 9.2.4).

Since the textual representation could be difficult to understand when the model becomes complicated, the `AsmetaVis` tool (see Chapter 6) is used. The tool allows two visualization techniques: *basic visualization* (shows the syntactical structure of the ASM in terms of a tree) and *semantic visualization* (represents the ASMs extrapolating part of the behaviour from the model). Figure 9.2 shows the basic visualization of the rule `r_run_dialysis` of the model in Code 9.1. Depending on the value assumed by `phase` function, the corresponding rule is executed.

Figure 9.3 shows the semantic visualization of the rule `r_run_dialysis`. The system starts in the `PREPARATION` phase and moves to the `INITIATION` phase by executing rule `r_run_preparation`, from which it moves to the `ENDING` phase by

```

asm HemodialysisGround

signature:
  enum domain Phases =
    {PREPARATION | INITIATION |
    ENDING}
  controlled phase: Phases

definitions:
  macro rule r_preparation =
    phase := INITIATION

  macro rule r_initiation =
    phase := ENDING

  macro rule r_ending =
    skip

  main rule r_Main =
  par
    if phase = PREPARATION then
      r_run_preparation[]
    endif
    if phase = INITIATION then
      r_run_initiation[]
    endif
    if phase = ENDING then
      r_run_ending[]
    endif
  endpar

  default init s0:
  function phase = PREPARATION
  
```

Code 9.1: Ground model

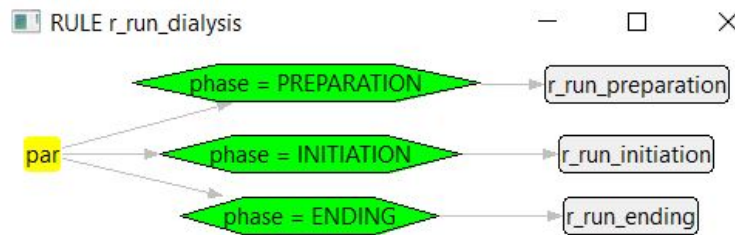


Figure 9.2: Ground model basic visualization

the execution of rule `r_run_initiation`. In the ENDING phase, rule `r_run_ending` is executed, and does not modify the phase. The simple visual inspection was sufficient to give us confidence that the model correctly evolves through the three top-level phases.

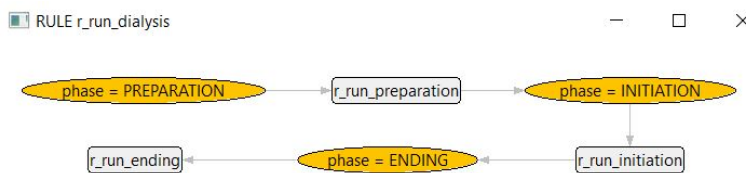


Figure 9.3: Ground model semantic visualization

Figure 9.4 shows the content of the rules `r_run_preparation`, `r_run_initiation` and `r_run_ending`. Since the ground model describes the transition between phases, the rules simply contain the update of the `phase` function to the next value.

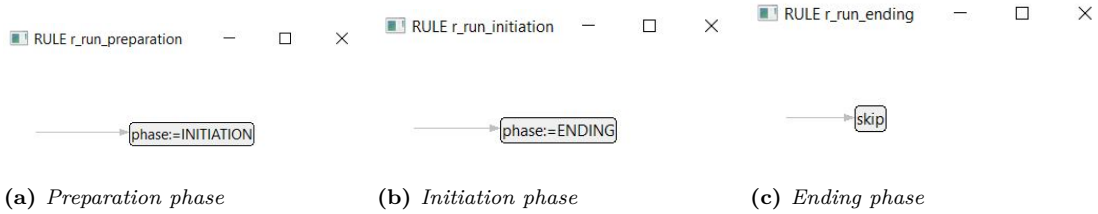


Figure 9.4: Ground model phases

### 9.2.2 First refinement: preparation phase

The first refinement extends the ground model by refining the PREPARATION phase. As shown in Figure 9.5, the preparation consists in a sequence of activities, specified by function `prepPhase`. For each value of `prepPhase`, a given rule performs some actions related to the device preparation and updates `prepPhase` to the next value.

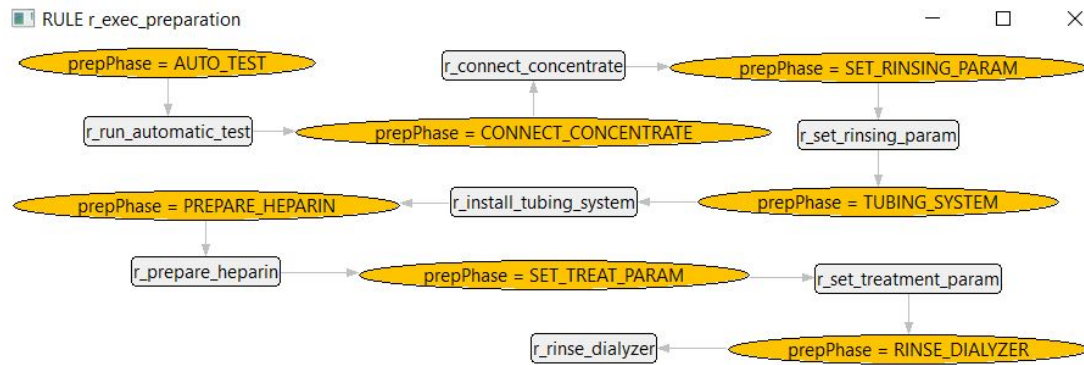


Figure 9.5: Preparation rule: semantic visualization

The overall behaviour is not clear if we consider the code of preparation phase (see Code 9.2) instead of the graphical representation. It is not easy to understand which is the correct sequence of rules execution because is not clear which is the next value assumed by `prepPhase` function.

As shown in Table 9.1, some phases are further divided in sub-phases. For example, Figure 9.6 shows how phase `SET_TREATMENT_PARAM` is specified by the treatment parameter (function `treatmentParam`). Also in this case, the sub-phases are executed in sequence.

The correctness of each refinement step has been proved with the tool `AsmRef-Prover` that checks a particular kind of refinement called *stuttering refinement* [24]. The first refined model is a correct stuttering refinement of the ground model, using as conformance relation the equality on the `phase` function. Figure 9.7 shows the correspondence of a refined run with an abstract run. We can see that, in the run of the ground model (abstract run), the machine goes from a state in which `phase` is `PREPARATION` to a state in which is `INITIATION` in one step. Instead, in the run of the refined model (refined run), there is a sequence of intermediate states in which `phase` remains in `PREPARATION`. These states are all compliant

```

asm HemodialysisGround

signature:
enum domain Phases =
{PREPARATION | INITIATION | ENDING}
enum domain PrepPhase = {RINSE_DIALYZER | SET_TREAT_PARAM | PREPARE_HEPARIN
| TUBING_SYSTEM | SET_RINSING_PARAM | CONNECT_CONCENTRATE | AUTO_TEST}

controlled phase: Phases
dynamic controlled prepPhase: PrepPhase

definitions:
...
macro rule r_exec_preparation =
  par
    if prepPhase = AUTO_TEST then
      r_run_automatic_test []
    endif
    if prepPhase = CONNECT_CONCENTRATE then
      r_connect_concentrate []
    endif
    if prepPhase = SET_RINSING_PARAM then
      r_set_rinsing_param []
    endif
    if prepPhase = TUBING_SYSTEM then
      r_install_tubing_system []
    endif
    if prepPhase = PREPARE_HEPARIN then
      r_prepare_heparin []
    endif
    if prepPhase = SET_TREAT_PARAM then
      r_set_treatment_param []
    endif
    if prepPhase = RINSE_DIALYZER then
      r_rinse_dialyzer []
    endif
  endpar

default init s0:
function phase = PREPARATION

```

Code 9.2: First refinement - Preparation rule

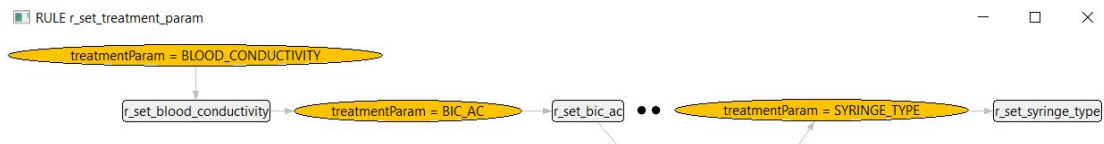


Figure 9.6: Treatment parameters

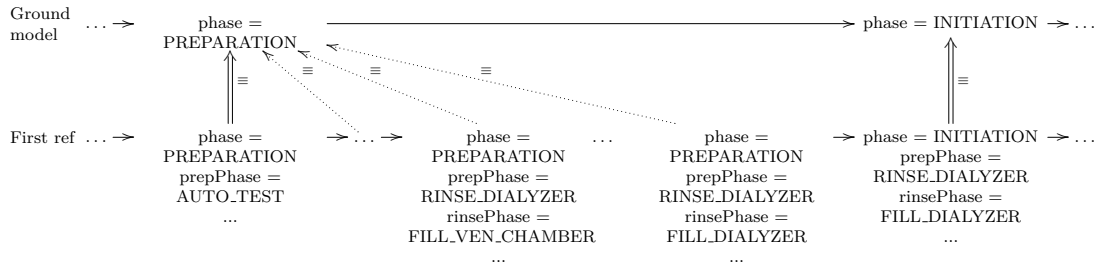


Figure 9.7: Hemodialysis case study – Relation between a refined run and an abstract run

with the first abstract state. The state in the refined run in which `phase` becomes `INITIATION` is compliant with the second abstract state. Furthermore, during preparation activities, the machine performs checks and manages errors if there were. If a violation occurs, the model tracks the error and activate an alarm. When the user confirms the alarm, the alarm stops. The error still active until the problem is resolved. For example, Figure 9.8 shows how the machine checks the upper limit of the temperature. If the temperature exceeds the upper threshold, the error and the alarm are raised. After that, the alarm is disabled by the user and the error is automatically reset when the temperature goes below the upper limit.

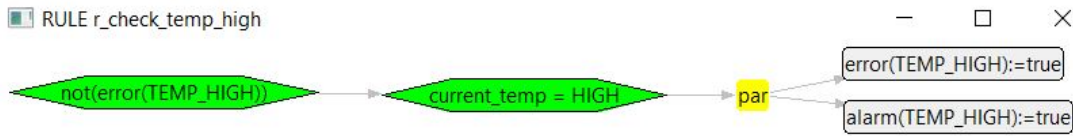


Figure 9.8: Check temperature upper limit

### Validation and Verification

Starting from the first refinement, *model review* is applied using `AsmetaMA` [20] tool. Common vulnerabilities and defects that can be introduced during ASM modeling are checked as violations of suitable *meta-properties* (*MPs*, defined in [20] as CTL formulae). In this model, we found that controlled function `machine_state` was initialized but never updated (violation of meta-property MP7). Although this is not a real fault of the model, it could make the model less readable, since a reader may expect an update of the function (since it is controlled). Declaring the function static made clear that, in this refinement step, the function is not updated. When modelling other case studies [15, 23, 26], we extensively used *interactive simulation* by means of the simulator `AsmetaS` [65] that allowed us to observe some particular system executions. In this case study, we could largely reduce the effort spent in simulation, since by semantic visualization we could get a feedback regarding the control flow similar to that provided by simulation. Figure 9.9 shows a simulation trace of the current model. Transitions between phases can be discovered also through simulation, but in a less direct way than with semantic visualization (see Figure 9.5). A further advantage of semantic

<pre> Insert a boolean constant for auto_test_end: true &lt;State 0 (monitored)&gt; auto_test_end=true &lt;/State 0 (monitored)&gt; &lt;State 1 (controlled)&gt; alarm(DF_PREP)=false alarm(SAD_ERR)=false alarm(TEMP_HIGH)=false dialyzer_connected_contr=false error(DF_PREP)=false error(SAD_ERR)=false error(TEMP_HIGH)=false phase=PREPARATION prepPhase=CONNECT_CONCENTRATE preparing_DF=false signal_lamp=GREEN &lt;/State 1 (controlled)&gt; </pre>	<pre> Insert a boolean constant for conn_concentrate: true &lt;State 1 (monitored)&gt; conn_concentrate=true &lt;/State 1 (monitored)&gt; &lt;State 2 (controlled)&gt; alarm(DF_PREP)=false alarm(SAD_ERR)=false alarm(TEMP_HIGH)=false dialyzer_connected_contr=false error(DF_PREP)=false error(SAD_ERR)=false error(TEMP_HIGH)=false phase=PREPARATION prepPhase=SET_RINSING_PARAM preparing_DF=true signal_lamp=GREEN &lt;/State 2 (controlled)&gt; </pre>
---	--

**Figure 9.9:** *Simulation trace of first refinement model*

visualization is that it also shows the rule that changes a given phase. However, simulation shows ASM states, whereas semantic visualization only shows control states given by the value of the phase function. Therefore, if we are interested in observing the exact ASM runs, we still have to use simulation. Instead, if we are only interested in knowing how the machine evolves through its phases, the semantic visualization is enough. Instead of interactive simulation, we mainly performed *scenario-based validation* [46] that permits to automatize the simulation activity, so scenarios can be rerun after specification modifications. In scenario-based validation the designer writes a scenario (using the textual notation *Avalla*) specifying the expected behaviour of the model; scenarios are similar to test cases. The tool *AsmetaV* reads the scenario and executes it using the simulator *AsmetaS*. We wrote several scenarios for the different refinement steps. We discovered that such scenarios had several common parts, since they had to perform the same actions and same checks in different parts of their evolution. Code 9.3 shows an example of scenario for the first refined model reproducing the whole therapy process. We defined the block `initStatePrep`, since its instructions regarding the

<pre> scenario completeTherapyRef1  load HemodialysisRef1.asm  begin initStatePrep   check phase = PREPARATION;   check prepPhase = AUTO_TEST;   check rinsingParam = FILLING_BP_RATE;   ... end //initStatePrep  begin preparationPhase   begin automaticTest     set auto_test_end := true;   step end //automaticTest </pre>	<pre> begin connectConcentrate   check prepPhase = CONNECT_CONCENTRATE;   check signal_lamp = GREEN;   set conn_concentrate := true;   step end //connectConcentrate ... end //preparationPhase  check phase = INITIATION; check bp_status = STOP; check bp_status_der = STOP; step  check phase = ENDING; step </pre>
---	--

**Code 9.3:** *Scenario for the first refinement*

initial state will be reused in scenarios written for other refinement steps. We also defined the block `preparationPhase` containing instructions related to the `PREPARATION` phase. Such block is further divided in sub-blocks (e.g., `automaticTest`),

since some scenarios will reuse the whole block `preparationPhase`, while others will reuse only some sub-blocks and redefine some others. Once a modeller is confident enough that the model correctly reflects the intended requirements, heavier techniques can be used for property verification. The case study document [100] reports a list of safety requirements (divided between *general* (S1-S11) and *software* (R1-R36) requirements) that must be guaranteed. We have specified them as LTL properties and verified using the tool `AsmetaSMV` [19] that translates `AsmetaL` models to models of the model checker `NuSMV`. Each requirement has been proved as soon as possible in the chain of refinements, i.e., in the model that describes the elements involved in the requirement. At this refinement step, we were able to express only 13 of the 47 requirements; these are software requirements regarding the flow of bicarbonate concentration into the mixing chamber, the heating of the dialysing fluid, and the detection of safety air conditions. Requirement R20 states that “if the machine is in the preparation phase and performs priming or rinsing or if the machine is in the initiation phase and if the temperature exceeds the maximum temperature, then the software shall disconnect the dialyser from the DF and execute an alarm signal.” The requirement has been formalized in LTL as follows.

---

```
//R20
g((phase = PREPARATION and dialyzer_connected_contr and prepPhase = RINSE_DIALYZER and not error(
  TEMP_HIGH) and current_temp = HIGH) implies x(error(TEMP_HIGH) and alarm(TEMP_HIGH) and not
  dialyzer_connected_status))
```

---

Note that some requirements are strictly related and somehow redundant and, therefore, can be verified together with only one property. This is the case of requirements R18 and R19, and requirements R23-R32. Requirements R18 and R19 specify conditions about the concentrate:

- “During preparation of the DF in the bicarbonate mode, if *acid* concentrate is provided instead of bicarbonate concentrate, then the software shall detect the mix-up of concentrates and disconnect the dialyser from the DF and execute an alarm signal.”
- “During preparation of the DF in the bicarbonate mode, if *acetate* concentrate is provided instead of bicarbonate concentrate, then the software shall detect the mix-up of concentrates and disconnect the dialyser from the DF and execute an alarm signal.”

These two requirements are verified together because it is enough to verify that the sensor detects bicarbonate, otherwise raise an alarm.

---

```
//R18–R19
g((phase = PREPARATION and prepPhase = RINSE_DIALYZER and dialyzer_connected_contr and not error(
  DF_PREP) and preparing_DF and not detect_bicarbonate) implies x(error(DF_PREP) and alarm(DF_PREP)
  and not dialyzer_connected_status))
```

---

Concerning requirements R23-R32, they define conditions relative to safety air detector (SAD) sensor:

- “If the machine is in the preparation phase and performing rinsing of the EBC or if the machine is connecting the patient or if the machine is in the initiation phase or if the machine is in the reinfusion process, then the software shall monitor the flow through the SAD sensor and if the flow through the SAD



sensor exceeds 1200 mL/min, then the software shall stop the BP and execute an alarm signal.”

- “If the flow through the SAD sensor is in the range of 0 to 200 mL/min, then the software shall use an air volume of 0.2mL as limit for air detection by the SAD sensor.”
- “If the flow through the SAD sensor is in the range of 200 to 400 mL/min, then the software shall use an air volume of 0.3mL as limit for air detection by the SAD sensor.”
- “If the flow through the SAD sensor is greater than 400 mL/min, then the software shall use an air volume of 0.5mL as limit for air detection by the SAD sensor.”
- “The software shall update the air volume detected by the SAD sensor every 1mS.”
- “If the machine is in the preparation phase and performing rinsing of the EBC and if the software detects that the air volume exceeds the air volume limit depending on the actual flow through the venous blood line, then the software shall stop the blood flow and execute an alarm signal.”
- “During the application of arterial bolus, if the software detects that the air volume exceeds the air volume limit depending on the actual flow through the venous blood line, then the software shall stop the blood flow and execute an alarm signal.”
- “While connecting the patient, if the software detects that the air volume exceeds the air volume limit depending on the actual flow through the venous blood line, then the software shall stop the blood flow and execute an alarm signal.”
- “During the initiation phase, if the software detects that the air volume exceeds the air volume limit depending on the actual flow through the venous blood line, then the software shall stop the blood flow and execute an alarm signal.”
- “During the reinfusion process, if the software detects that the air volume exceeds the air volume limit depending on the actual flow through the venous blood line, then the software shall stop the blood flow and execute an alarm signal.”

---

```
//R23-R32
g((phase = PREPARATION and prepPhase = TUBING_SYSTEM and passed1Msec and currentSAD !=
  PERMITTED and current_air_vol != PERMITTED and not error(SAD_ERR)) implies x(error(SAD_ERR) and
  alarm(SAD_ERR)))
```

---

Requirements R20 and R23-R32 are related both to the preparation and initiation phases; R23-R32 also consider the ending phase. At this level of refinement, these properties can only check the preparation phase; they will be refined in the second refinement to take into consideration the initiation phase (see Sect. 9.2.3), and in the third refinement for considering the ending phase (see Sect. 9.2.4).

### 9.2.3 Second refinement: initiation phase

The second refinement extends the first refinement by refining the INITIATION phase. As shown in Table 9.1, the phase is further divided in two phases (recorded

by function `initPhase`): the connection of the patient (`CONNECT_PATIENT`) and the running of the therapy (`THERAPY_RUNNING`).

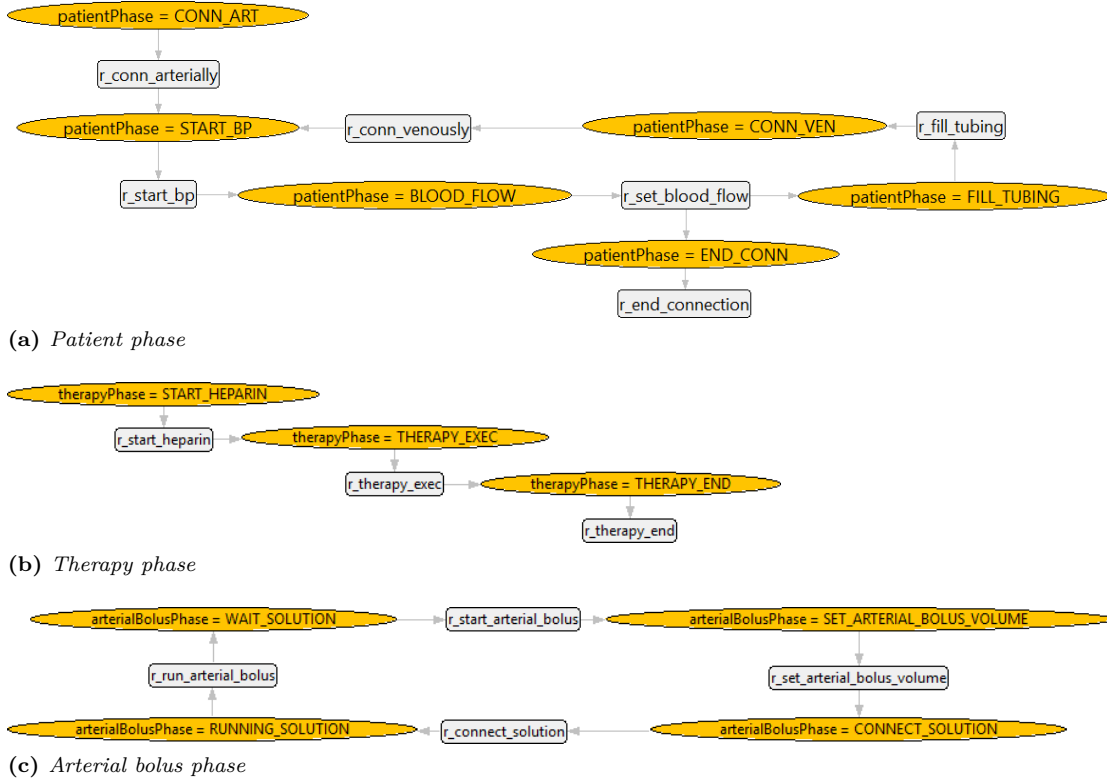


Figure 9.10: Second refinement – Semantic visualization

As shown in Figure 9.10a, function `patientPhase` indicates in which step the patient is during the connection. The patient is initially connected arterially; then the blood pump is activated to extract the blood from the patient (in state `BLOOD_FLOW`). In this state, rule `r_set_blood_flow` can follow two different paths:

- In the first execution of the rule, `patientPhase` is updated to `FILL_TUBING`. Then, the operator sets the blood flow and the blood pump stops when the blood fills the tubes between the patient and the dialyser. After this, the patient is connected venously and the blood pump is restarted to fill the tubes between the dialyser and the patient vein.
- In the second execution of the rule (when the patient has been venously connected), `patientPhase` is updated to `END_CONN`. Then, the therapy can start (i.e., `initPhase` goes to `THERAPY_RUNNING`).

Note that, in this case, the semantic visualization is not sufficient to completely understand the model behaviour, since the path taken after the rule execution can only be discovered by simulation. The therapy status is specified by function `therapyPhase` (see Figure 9.10b) that, when in `THERAPY_EXEC`, is further specified by `arterialBolusPhase`, whose semantic visualization is shown in Figure 9.10c. Such sub-phase consists in the infusion of saline solution and it is activated by the operator. `arterialBolusPhase` is initially in state `WAIT_SOLUTION` until the operator presses the start button. After that, the doctor sets the volume of

---

```

scenario completeTherapyRef2

load HemodialysisRef2.asm

begin initStateInit
execblock completeTherapyRef1.initStatePrep;

check patientPhase = CONN_ART;
check arterialBolusPhase = WAIT_SOLUTION;
...
end

execblock completeTherapyRef1.preparationPhase;

begin initiationPhase
begin patientConnection
check phase = INITIATION;
check initPhase = CONNECT_PATIENT;
check patientPhase = CONN_ART;
set art_connected := true;
step
...
end
end

check phase = ENDING;
step

```

---

Code 9.4: Scenario for the second refinement

the saline solution, the solution is connected to the machine, and the infusion starts. When the predefined volume is infused, the `arterialBolusPhase` returns to `WAIT_SOLUTION` state until the operator restarts again the saline infusion. Also in this case, semantic visualization does not allow to fully understand the machine behaviour: the initial state of the graph in Figure 9.10c can only be discovered through simulation. As required by the modelling process (see Figure 4.2), before any further validation and verification activity of the requirements, it is necessary to guarantee correctness of the refinement step. This has been carried out, similarly to what described at the end of Sect. 9.2.2, by means of the `AsmRefProver`.

### Validation and Verification

By model review, some locations were *trivially updated* (meta-property MP4), i.e., that the value of the location before the update was always equal to the new value. This means that the update is not necessary. Removing trivial updates is important because the reader may have the feeling that the ASM is modifying its state when it is not. The trivial updates were related to `signal_lamp` when updated to `GREEN`, and `error(UF_DIR)` and `error(UF_RATE)` when updated to true. The update of `signal_lamp` was indeed unnecessary and we removed it; the updates of `error(UF_DIR)` and `error(UF_RATE)`, instead, were not correct since the locations had to be updated to false and so we fixed the fault. Moreover, we found that functions `bf_err_ap_low`, `reset_err_pres_ap_low` were never updated (meta-property MP7): this was due to a wrong guard in a conditional rule. This shows that model review is also useful in detecting behavioural faults. We found that also locations `error(ARTERIAL_BOLUS_END)`, `error(UF_BYPASS)`, and `error(UF_VOLUME_ERR)` were never updated. This is due to the fact that functions `error` and `alarm` share the domain `AlarmErrorType` representing the different alarms; for each alarm there is an error, except for `ARTERIAL_BOLUS_END`, `UF_BYPASS`, and `UF_VOLUME_ERR`. Therefore, locations `error(ARTERIAL_BOLUS_END)`, `error(UF_BYPASS)`, and `error(UF_VOLUME_ERR)` are actually unnecessary. It is possible to declare two different domains for errors and alarms, but we think that the specification would have been less clear and it would have been more difficult to keep consistent the values of errors and alarms. Therefore, we ignored the meta-property violation and we left the model as it is. We also wrote some scenarios for this refinement step, as the one shown in Code 9.4. The scenario reuses blocks

## Chapter 9. Hemodialysis machine case study

---

`initStatePrep` and `preparationPhase` defined in scenario `completeTherapy-Ref1` for the first refined model (see Code 9.3). At this modelling level, it is possible to prove 23 more safety requirements. They concern patient connection, infusion of the saline solution when the patient is connected to the extra-corporeal blood circuit, pressure during the therapy, dialysing fluid temperature, heparin infusion, air detected in the blood and ultrafiltration process. Among these, we realized that some were not correctly described in [100]. For example, S1 states that “arterial and venous connectors of the EBC are connected to the patient simultaneously”. The corresponding LTL property is as follows

---

```
g(art_connected_contr iff ven_connected_contr)
```

---

However, the property is false because the patient is connected *before* to the arterial connector and *then* to the venous connector. Other requirements are instead ambiguous and so we had problems in formalizing them. For example, S5 states that “the patient cannot be connected to the machine outside the initiation phase, e.g., during the preparation phase.” We did not know how to interpret “be connected”: as the patient status of being attached to the machine, or as the atomic action performed by the operator of connecting the patient to the machine? The former interpretation would require to prove the following property:

---

```
g((art_connected_contr or ven_connected_contr) implies phase = INITIATION)
```

---

that, however, is false. Indeed, the patient can be attached to the machine also outside the `INITIATION` phase. The former interpretation, instead, would require to prove the two following properties:

---

```
g((not art_connected_contr and x(art_connected_contr)) implies phase = INITIATION)
g((not ven_connected_contr and x(ven_connected_contr)) implies phase = INITIATION)
```

---

that are actually both true. It may be the case that this interpretation is not correct; this is a clear example of ambiguous requirement that would need a clarification from the stakeholders. Such clarification would also be needed for requirements S2, S3, S6 and R16, for which we were not able to provide a satisfactory formalization. Requirement S4 verifies that “when the patient is connected to the EBC, the saline solution in the EBC is replaced with blood. The BP is stopped when either the VRD detects blood or the BP has transported a predefined volume.”

---

```
//S4
g((phase = INITIATION and initState = CONNECT_PATIENT and patientPhase = FILL_TUBING and not(
  err_patient_conn) and not(error_bp) and (blood_on_VRD or conn_infuse_set_volume = PERMITTED)) implies x
  (bp_status_der = STOP and patientPhase = CONN_VEN))
```

---

Requirement R1 refers to arterial bolus applications during which “the software shall monitor the infusion of saline into the patient and if the infused volume exceeds 0.4 l, then the software shall stop the blood flow and execute an alarm signal.”

---

```
//R1
g((phase = INITIATION and initState = THERAPY_RUNNING and therapyPhase = THERAPY_EXEC and
  arterialBolusPhase = RUNNING_SOLUTION and not(error_therapy) and not(error_bp) and not(error(
  ARTERIAL_BOLUS)) and current_art_bolus_volume = HIGH) implies x(error(ARTERIAL_BOLUS) and alarm(
  ARTERIAL_BOLUS)))
```

---

Requirements R2-R4 specify properties about the blood pump during initiation phase. The requirements state:

- R2: “the software shall monitor the blood flow in the EBC and if no flow is detected for more than 120 s, then the software shall stop the BP and execute an alarm signal.”
- R3: “if the machine is not in bypass, then the software shall monitor the blood flow in the EBC and if the actual blood flow is less than 70% of the set blood flow, then the software shall execute an alarm signal.”
- R4: “the software shall monitor the rotation direction of the BP and if the software detects that the BP rotates backwards, then the software shall stop the BP and execute an alarm signal.”

The properties defined for requirements R2 and R4 include the verification of requirements R14 and R17 respectively. Requirements R14 and R17 express the same requirement in a sub-phase considered by R2 and R4.

---

```
//R2 and R14
g((phase = INITIATION and bp_status_der = START and not(error(BP_NO_FLOW)) and (not(detected_blood_flow
) and passed120Sec)) implies x(error(BP_NO_FLOW) and alarm(BP_NO_FLOW)))
//R3
g((phase = INITIATION and bp_status_der = START and machine_status_der = MAIN_FLOW and not(error(
BP_LESS_FLOW)) and current_bp_flow = TOLOW) implies x(error(BP_LESS_FLOW) and alarm(
BP_LESS_FLOW)))
//R4 and R17
g((phase = INITIATION and bp_status_der = START and not(error(BP_ROTATION_DIR)) and bp_rotates_back)
implies x(error(BP_ROTATION_DIR) and alarm(BP_ROTATION_DIR)))
```

---

Requirements from R5 to R11 regard the value assumed by arterial and venous pressure. The values must remain within the limits according to these conditions:

- R5: “During initiation, if the software detects that the pressure at the VP transducer exceeds the upper pressure limit, then the software shall stop the BP and execute an alarm signal.”
- R6: “During initiation, if the software detects that the pressure at the VP transducer falls below the lower pressure limit, then the software shall stop the BP and execute an alarm signal.”
- R7: “During initiation, if the software detects that the pressure at the AP transducer exceeds the upper pressure limit, then the software shall stop the BP and execute an alarm signal.”
- R8: “During initiation, if the software detects that the pressure at the AP transducer falls below the lower pressure limit, then the software shall stop the BP and execute an alarm signal.”
- R9: “While connecting the patient, if the software detects that the pressure at the VP transducer exceeds +450mmHg for more than 3 s, then the software shall stop the BP and execute an alarm signal.”
- R10: “While connecting the patient, if the software detects that the pressure at the VP transducer falls below the defined lower pressure limit for more than 3 s, then the software shall stop the BP and execute an alarm signal.”
- R11: “While connecting the patient, if the software detects that the pressure at the AP transducer falls below the lower pressure limit for more than 1 s, then the software shall stop the BP and execute an alarm signal.”

---

```
//R5
```

## Chapter 9. Hemodialysis machine case study

---

```
g((phase = INITIATION and initPhase = THERAPY_RUNNING and therapyPhase = THERAPY_EXEC and not(
  error_therapy) and bp_status_der = START and ap_limits_set and vp_limits_set and not(error(INIT_VP_UP))
  and current_vp = HIGH) implies x(error(INIT_VP_UP) and alarm(INIT_VP_UP)))
//R6
g((phase = INITIATION and initPhase = THERAPY_RUNNING and therapyPhase = THERAPY_EXEC and not(
  error_therapy) and bp_status_der = START and ap_limits_set and vp_limits_set and not(error(INIT_AP_UP))
  and current_ap = HIGH) implies x(error(INIT_AP_UP) and alarm(INIT_AP_UP)))
//R7
g((phase = INITIATION and initPhase = THERAPY_RUNNING and therapyPhase = THERAPY_EXEC and not(
  error_therapy) and bp_status_der = START and ap_limits_set and vp_limits_set and not(error(INIT_VP_LOW))
  and current_vp = TOOLOW) implies x(error(INIT_VP_LOW) and alarm(INIT_VP_LOW)))
//R8
g((phase = INITIATION and initPhase = THERAPY_RUNNING and therapyPhase = THERAPY_EXEC and not(
  error_therapy) and bp_status_der = START and ap_limits_set and vp_limits_set and not(error(INIT_AP_LOW))
  and current_ap = TOOLOW) implies x(error(INIT_AP_LOW) and alarm(INIT_AP_LOW)))
//R9
g((phase = INITIATION and initPhase = CONNECT_PATIENT and not(err_patient_conn) and not(error_bp) and
  bp_status_der = START and not(error(CONN_VP_UP)) and current_vp = HIGH and passed3Sec) implies x(
  error(CONN_VP_UP) and alarm(CONN_VP_UP)))
//R10
g((phase = INITIATION and initPhase = CONNECT_PATIENT and not(err_patient_conn) and not(error_bp) and
  bp_status_der = START and not(error(CONN_VP_LOW)) and current_vp = TOOLOW and passed3Sec)
  implies x(error(CONN_VP_LOW) and alarm(CONN_VP_LOW)))
//R11
g((phase = INITIATION and initPhase = CONNECT_PATIENT and not(err_patient_conn) and not(error_bp) and
  bp_status_der = START and not(error(CONN_AP_LOW)) and current_ap = TOOLOW and passed1Sec)
  implies x(error(CONN_AP_LOW) and alarm(CONN_AP_LOW)))
```

---

Two of the four requirements defined for patient connection (R14 and R17) have been verified together with previous requirements. Requirement R15 states that “while connecting the patient, the software shall monitor the filling blood volume of the EBC and if the filling blood volume exceeds 400 mL, then the software shall stop the BP and execute an alarm signal. The blood volume can be detected by measuring the pump rotations with the speed sensor of the BP.”

```
//R15
g((phase = INITIATION and initPhase = CONNECT_PATIENT and bp_status_der = START and not(
  err_patient_conn) and not(error_bp) and not(error(FILL_BLOOD_VOL)) and current_fill_blood_vol = HIGH)
  implies x(error(FILL_BLOOD_VOL) and alarm(FILL_BLOOD_VOL)))
```

---

Regarding the water used in dialyser fluid, requirement R21 is introduced during initiation phase. It states that “if the machine is in the initiation phase and if the temperature falls below the minimum temperature of 33°C, then the software shall disconnect the dialyser from the DF and execute an alarm signal.”

```
//R21
g((phase = INITIATION and not(error(TEMP_LOW)) and current_temp = TOOLOW) implies x(error(TEMP_LOW)
  ) and alarm(TEMP_LOW)))
```

---

During heparin infusion (R22) “the software shall monitor the anticoagulant flow direction and if the reverse direction is detected, then the software shall stop the blood flow and the anticoagulant flow, and execute an alarm signal.”

```
//R22
g((phase = INITIATION and initPhase = THERAPY_RUNNING and therapyPhase = THERAPY_EXEC and not(
  error_therapy) and heparin_running and not(error(HEPARIN_DIR)) and reverse_dir_heparin) implies x(error(
  HEPARIN_DIR) and alarm(HEPARIN_DIR)))
```

---

Requirements R33-R36 express conditions about ultrafiltration process. The requirements state:

- R33: “The software shall monitor the net fluid removal rate in the balance chamber and if the net fluid removal rate exceeds a safe upper limit, then

the software shall stop flow from and to the dialyser and execute an alarm signal.”

- R34: “If the machine is in the initiation phase and net fluid removal is enabled, then the software shall monitor the rotation direction of the UFP and if backward rotation of the UFP is detected, then the software shall put the machine in bypass and execute an alarm signal. The backward delivered volume shall not exceed 400 mL.”
- R35: “If the machine is in the initiation phase and net fluid removal is enabled, then the software shall monitor the net fluid removal volume and if the net fluid removal volume exceeds (UF set volume + 200 mL), then the software shall put the machine in bypass and execute an alarm signal. When the alarm is acknowledged by the user, then the software shall increase the UF set volume by 200 mL.”
- R36: “If the machine is in the initiation phase and net fluid removal is enabled and if the bypass valve is opened, then the software shall stop the DF flow to and from the dialyser and execute an alarm signal.”

---

```
//R33
g((phase = INITIATION and initPhase = THERAPY_RUNNING and therapyPhase = THERAPY_EXEC and not(
  error_therapy) and bicarbonate_status_der and not(error(UF_RATE)) and current_UF_rate = HIGH) implies x(
  error(UF_RATE) and alarm(UF_RATE)))
//R34
g((phase = INITIATION and initPhase = THERAPY_RUNNING and therapyPhase = THERAPY_EXEC and not(
  error_therapy) and bicarbonate_status_der and not(error(UF_DIR)) and uf_dir_backwards) implies x(error(
  UF_DIR) and alarm(UF_DIR)))
//R35
g((phase = INITIATION and initPhase = THERAPY_RUNNING and therapyPhase = THERAPY_EXEC and not(
  error_therapy) and bicarbonate_status_der and not(alarm(UF_VOLUME_ERR)) and current_UF_volume =
  HIGH) implies x(alarm(UF_VOLUME_ERR)))
//R36
g((machine_status_der = BYPASS) implies not(df_flow_state))
```

---

Moreover, properties related to requirements R20 and R23-R32 are refined to take into consideration also the initiation phase.

---

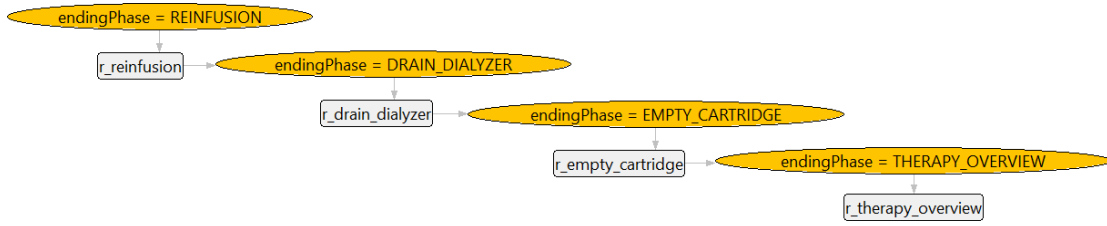
```
//R20 updated
g(((phase = INITIATION and not error(TEMP_HIGH) and current_temp = HIGH) or (phase = PREPARATION
  and dialyzer_connected_contr and prepPhase = RINSE_DIALYZER and not error(TEMP_HIGH) and
  current_temp = HIGH)) implies x(error(TEMP_HIGH) and alarm(TEMP_HIGH) and not
  dialyzer_connected_status))
//R23–R32 updated
g((((phase = PREPARATION and prepPhase = TUBING_SYSTEM) or (phase = INITIATION and bp_status_der =
  START)) and (passed1Msec and currentSAD != PERMITTED and current.air_vol != PERMITTED and not
  error(SAD_ERR))) implies x(error(SAD_ERR) and alarm(SAD_ERR)))
```

---

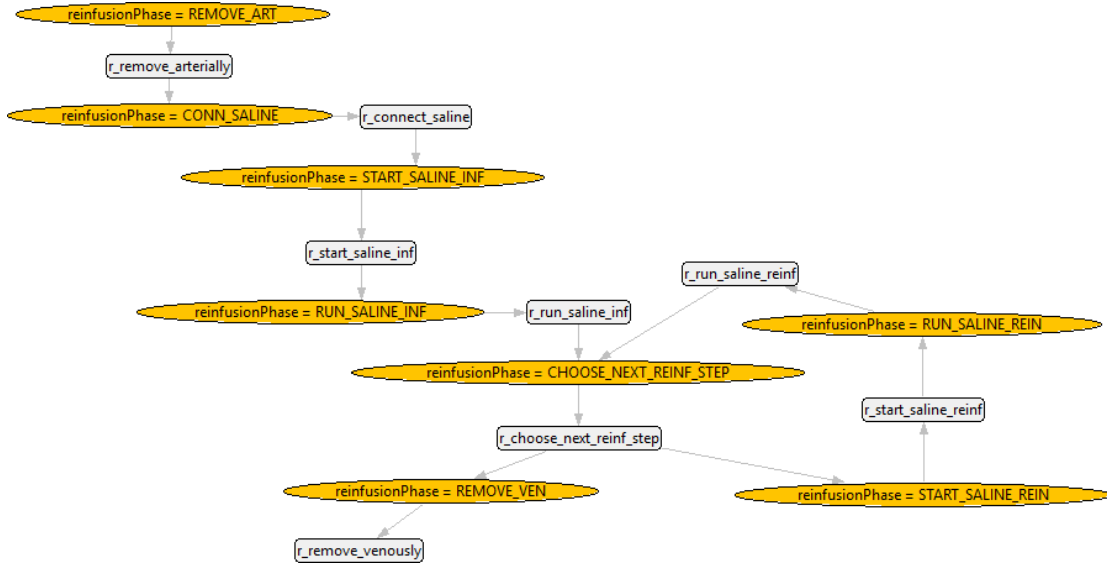
### 9.2.4 Third refinement: ending phase

The third refinement extends the second refinement by refining the `ENDING` phase. As shown in Figure 9.11a, the ending consists in a sequence of activities (specified by function `endingPhase`).

When in `REINFUSION`, the phase is further refined by `reinfusionPhase`, whose semantic visualization is shown in Figure 9.11b. The reinfusion consists in an initial sequence of activities for starting the infusion of the saline solution, followed by a loop in which the doctor performs the solution reinfusion. Rule `r_choose_next_reinf_step` is responsible for deciding the loop termination: either going to



(a) *Ending phase*



(b) *Reinfusion phase*

**Figure 9.11:** *Third refinement – Semantic visualization*

START\_SALINE\_REIN (i.e., the operator decides to continue the reinfusion) or to REMOVE\_VEN (i.e., the operator disconnects the patient). To complete modelling at this level, `AsmRefProver` was used to prove that this model is a correct stuttering refinement of the second refinement.

### Validation and Verification

Model review did not find any meta-property violation in this model. For this model, we wrote scenarios for reproducing the occurrence of some errors. Code 9.5 shows the scenario that triggers error `TEMP_HIGH` related to high temperature of the dialyser fluid during the preparation phase. When the temperature exceed the upper limit, the model raises an error and an alarm. The alarm is reset by the user, while the error is reset when the current value of the temperature is within the limits. We can see that in this scenario we reused some sub-blocks of block `preparationPhase` defined in scenario `completeTherapyRef1` (see Code 9.3) as, for example, `automaticTest` and `connectConcentrate`. We did not use the whole block because the instructions related to the dialyser rinsing had to be changed in order to trigger the error. Indeed, blocks defined in scenario `completeTherapyRef2` and `completeTherapyRef3` are used. In this refinement step, we were able to prove three more requirements. The first is a general re-



---

```

scenario installTubingTempHigh

load HemodialysisRef3.asm

execblock completeTherapyRef3.initStateEnd;

execblock completeTherapyRef1.automaticTest;
execblock completeTherapyRef1.connectConcentrate;
execblock completeTherapyRef1.setRinsingParam;
execblock completeTherapyRef1.installTubingSystem;
execblock completeTherapyRef1.prepareHeparin;
execblock completeTherapyRef1.setTreatmentParam;

check rinsePhase = FILL_ART_CHAMBER;
set current_temp := HIGH;
set reset_alarm := false;
step
check error(TEMP_HIGH) = true;
check alarm(TEMP_HIGH) = true;
set arterial_chamber_filled := false;
step
check rinsePhase = FILL_ART_CHAMBER;
set arterial_chamber_filled := true;
step
set reset_alarm := true;
step
check alarm(TEMP_HIGH) = false;
set current_temp := PERMITTED;
step
check dialyzer_connected_status = true;
check error(TEMP_HIGH) = false;
step
...

execblock completeTherapyRef2.initiationPhase;
execblock completeTherapyRef3.endingPhase;

```

---

**Code 9.5:** *Scenario for the third refinement – Triggering of error TEMP\_HIGH*

quirements about blood pump, it states that “Once “empty dialyser” has been confirmed, the BP cannot be started anymore.”

---

```

//S11
g(empty_dialyzer implies g(bp_status_der = STOP))

```

---

The other two requirements specifies conditions about venous pressure and arterial pressure during reinfusion phase:

- R12: “if the software detects that the pressure at the VP transducer exceeds +350mmHg for more than 3 s, then the software shall stop the BP and execute an alarm signal.”
- R13: “if the software detects that the pressure at the AP transducer falls below –350mmHg for more than 1 s, then the software shall stop the BP and execute an alarm signal.”

---

```

//R12
g((phase = ENDING and endingPhase = REINFUSION and not(error_rein_press) and not(error_bp) and
  bp_status_der = START and not(error(REIN_VP_UP)) and current_vp = HIGH and passed3Sec) implies x(
  error(REIN_VP_UP) and alarm(REIN_VP_UP)))
//R13
g((phase = ENDING and endingPhase = REINFUSION and not(error_rein_press) and not(error_bp) and
  bp_status_der = START and not(error(REIN_AP_LOW)) and current_ap = TOOLOW and passed1Sec) implies
  x(error(REIN_AP_LOW) and alarm(REIN_AP_LOW)))

```

---

Moreover, we could refine the property related to requirements R23-R32 in order to take into consideration also the ending phase.

---

```

//R23–R32 updated
g((((phase = PREPARATION and prepPhase = TUBING.SYSTEM) or (phase = INITIATION and bp_status_der =
  START) or (phase = ENDING and endingPhase = REINFUSION and not error_rein_press and bp_status_der
  = START)) and (passed1Msec and currentSAD != PERMITTED and current_air_vol != PERMITTED and
  not error(SAD_ERR))) implies x(error(SAD_ERR) and alarm(SAD_ERR)))

```

---

Note that there are four requirements that we do not consider in our work. They are related to aspects that we do not capture because of abstraction, requirements S7-S10 refer to continuous values of the blood flow rate, while we have discretized them.

### 9.3 Conformance checking

The implementation of the hemodialysis device is not accessible. Therefore, a prototypical implementation in Java of the hemodialysis device software has been built, in order to show the last part of the ASM-based process, i.e., the *conformance checking* between the implementation and the specification) and how the process can help to perform the activities required by steps 5.5-5.7 of the IEC 62304 standard, as well as step 5.8 on subsequent releases of medical software. Techniques of conformance checking are also useful for validation coverage and re-verification of software upon changes, as required by the FDA principles. The implementation faithfully reflects (or, at least, it should) the case study requirements; however, some components (e.g., the connection with the hardware) have not been implemented and so they have been substituted with mock objects. Part of the Java implementation is shown in Code 9.6.

```

import org.asmeta.monitoring.*;

@Asm(asmFile = "models/HemodialysisRef3.asm")
public class HemodialysisMachine {
    HemodialysisMachinePanel dialog;

    @FieldToFunction(func = "phase")
    Phases phase = Phases.PREPARATION;
    @FieldToFunction(func = "preparing_DF")
    boolean preparing_DF = false;
    @FieldToFunction(func = "initPhase")
    InitPhase initPhase = InitPhase.CONNECT_PATIENT;
    ...
    @Monitored(func = "interrupt_dialysis")
    boolean interrupt_dialysis = false;
    @Monitored(func = "error_heparin_resolve")
    boolean error_heparin_resolve = false;
    @Monitored(func = "blood_conductivity")
    int blood_conductivity = HIGH;
    ...
}

public HemodialysisMachine() {
    dialog = new HemodialysisMachinePanel(this);
    dialog.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
    dialog.setVisible(true);
}

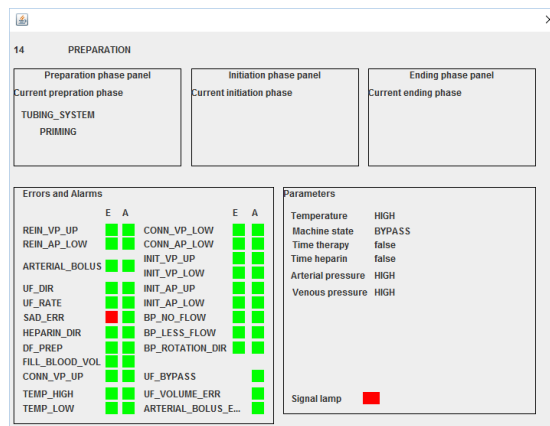
@RunStep
public void execDialysis() {
    if (phase == Phases.PREPARATION) {
        ...
        dialog.updateGUI();
    }
}

@MethodToFunction(func="error")
public boolean error(AlarmErrorType aet){
    return error[aet.ordinal()];
}
}

```

**Code 9.6:** Java implementation of the HMCS

Note that the implementation has many details that are not present in the specification, as the graphical user interface written in Swing (shown in Figure 9.12).



**Figure 9.12:** Hemodialysis device program GUI

Conformance checking can be done *offline* (i.e., *before* the deployment) by MBT

(Model Based Testing) or *online* (i.e., *after* the deployment) by RV (Runtime Verification). In this application the former approach has been applied to the case study. In MBT [73,129], *abstract test sequences* are derived from the specification; such sequences are then concretized in tests for the implementation. In order to generate abstract test sequences, we use the tool ATGT [64]. The tool first derives from the specification some test goals (called *test predicates*) according to some *coverage criteria* [63], and then generates sequences for covering these goals. For example, the *update rule coverage* criterion requires that each update rule is executed at least once in a test sequence and the update is not trivial (i.e., the new value is different from the current value of the location). The tool uses a classical approach based on model checking for generating tests. The ASM model is translated in the language of a model checker, and each test goal is expressed as a temporal property (called *trap property*); if the trap property is proved false, the returned counterexample is the *abstract test sequence* covering the test goal (and possibly also other test goals). For this work, ATGT has been extended in order to work with the model checker NuSMV. For the case study, we used the structural coverage criteria presented in [64]; ATGT built 980 test predicates and generated 183 tests for covering them (in around 2 hours). Note that each generated test can cover more than one test predicate and we avoid generating tests for already covered test predicates. An example of test predicate for the update rule coverage criterion is:

---

```
phase = INITIATION and initPhase = THERAPY_RUNNING and therapyPhase = THERAPY_EXEC and
interrupt.dialysis and therapyPhase != THERAPY_END
```

---

requiring to observe a state in which the update rule of `therapyPhase` to `THERAPY_END` fires and `therapyPhase` is not already equal to `THERAPY_END`. Note that a sequence covering this test predicate is guaranteed to exist if the specification has been previously checked with model review and no violation of MP4 (requiring that all the update rules are not always trivial) occurred. The sequence covering the predicate is shown in Figure 9.13. We can see that the test predicate holds in

```
- State 1 -
phase = PREPARATION
initPhase = CONNECT_PATIENT
therapyPhase = START_HEPARIN
interrupt_dialysis = FALSE
...

...
- State 46 -
phase = INITIATION
initPhase = CONNECT_PATIENT
therapyPhase = START_HEPARIN
interrupt_dialysis = FALSE
...

...
- State 53 -
phase = INITIATION
initPhase = CONNECT_PATIENT
therapyPhase = START_HEPARIN
interrupt_dialysis = FALSE
...

...
- State 54 -
phase = INITIATION
initPhase = THERAPY_RUNNING
therapyPhase = START_HEPARIN
interrupt_dialysis = FALSE
...

...
- State 55 -
phase = INITIATION
initPhase = THERAPY_RUNNING
therapyPhase = THERAPY_EXEC
interrupt_dialysis = TRUE
...
```

**Figure 9.13:** *Abstract test sequence*

the last state of the sequence. In order to concretize the abstract test sequences into tests for the implementation, we need to provide a linking between the specification and the implementation. In [21], a technique is proposed to do the linking using Java annotations. The different annotations are:

- associate a Java class with the corresponding ASM model (`@Asm`);

## Chapter 9. Hemodialysis machine case study

- associate the ASM state with the Java state:
  - `@FieldToFunction` connects a Java field with an ASM controlled function;
  - `@MethodToFunction` connects a Java pure (i.e., returning a value but not modifying the object state) method with an ASM controlled function;
  - `@Monitored` connects a Java field with an ASM monitored function; such fields represent the inputs of the Java class that take their value from the environment (as monitored functions in ASMs).
- associate the ASM behavior with the Java object behavior; `@RunStep` is used to annotate methods whose execution corresponds to a step of the ASM model.

Given the mapping provided by the Java annotations, the abstract test sequences are translated in JUnit tests following the technique described in [22]. For example, Code 9.7 shows the JUnit test corresponding to the sequence shown in Figure 9.13.

```

@Test
public void test() {
    HemodialysisMachine sut = new HemodialysisMachine();
    // check conformance
    assertEquals(Phases.PREPARATION, sut.phase);
    assertEquals(InitPhase.CONNECT_PATIENT, sut.initPhase);
    assertEquals(TherapyPhase.START_HEPARIN, sut.therapyPhase);
    ...
    // set monitored
    sut.interrupt_dialysis = false;
    ...
    // perform step
    sut.execDialysis();
    ...
}

sut.execDialysis();
// check conformance
assertEquals(Phases.INITIATION, sut.phase);
assertEquals(InitPhase.THERAPY_RUNNING, sut.initPhase);
assertEquals(TherapyPhase.THERAPY_EXEC, sut.therapyPhase);
...
// set monitored
sut.interrupt_dialysis = true;
// perform step
sut.execDialysis();
}

```

Code 9.7: JUnit test

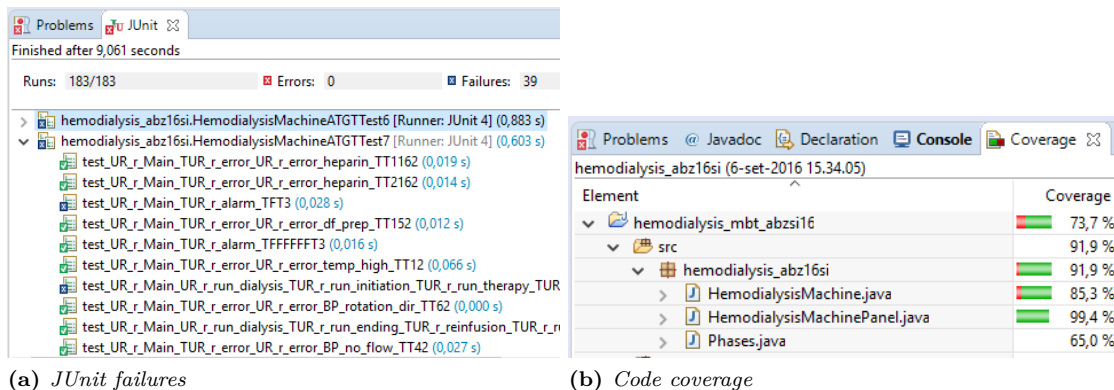


Figure 9.14: JUnit testing results

Each test is built by creating the initialization of the Java class and then, for each state of the corresponding abstract test sequence:

- update the fields annotated with `@Monitored` to the values of the corresponding ASM functions. In the example, field `interrupt_dialysis` is updated to the value of the homonymous ASM function.
- invoke the method annotated with `@RunStep`. In the example, method `execDialysis()` (see Code 9.6) is executed.

- add JUnit `assert` commands that check that the Java state is conformant with the ASM state; they check that the values of the fields annotated with `@FieldToFunction` and the values returned by the methods annotated with `@MethodToFunction` are equal to the values of the corresponding ASM functions. In the example, fields `phase`, `initPhase`, `therapyPhase`, ... are linked with `@FieldToFunction` and checked during conformance checking.

We run all the 183 tests (divided in multiple JUnit files) and we actually found some conformance violations (i.e., some tests failed. See Figure 9.14a). We analysed the failing tests and we discovered that the authors writing the implementation misunderstood some requirements. Consequently, the errors were fixed. The coverage obtained by the tests is shown in Figure 9.14b: our tests were able to cover more than 90% of the Java code. Although the obtained coverage is already high, we found that some parts of the code are not covered, since the structure of the code in some parts is different from the structure of the specification. Using structural coverage criteria for test generation may not guarantee the full coverage.

## 9.4 Related Work

---

The hemodialysis machine case study has been developed by other four groups of researchers.

The authors of paper [75] model the system using iUML-B notation and then they verify the behaviour using the deductive theorem proving and model checking. Furthermore, the model is validated using the animation technique. The final model is translated into a sequential programming language code using a semi-automatic process. The limitation of this approach is that the translator translates only a subset of the B syntax.

The solution presented in [33] applies Hybrid Event-B to explicitly focus on continuously varying state evolution. This approach distinguishes between discrete and continuous elements of hemodialysis machine but the tool do not support all features of the notation.

The authors of paper [67] show a solution based on Circus. The solution specifies the concurrent and parallel aspects of the system and it focuses on timing properties. The model presented is not verified because there are not tools that support Circus specification. The model is translated in Communicating Sequential Processes (CSP) and the specification is verified using CSP model checker.

The solution in paper [60] is based on the graphical notation Algebraic State-Transition Diagrams (ASTD) and Event-B. The graphical notation simplifies the validation of the model, while the verification is performed using the theorem prover and model checker of Event-B.

## 9.5 Conclusions

---

The hemodialysis machine case study is a complex application of ASMs using `Asmeta` framework. The notation used is easy to understand and the visualization tool helps the developer to follow better the behaviour of the machine. `Asmeta` framework provides tools to perform simulation, model review, model checking,

and conformance checking. It gives a grasp on the notion of correctness far better than the approaches which are comprised of only a subset of the employed analysis techniques in this work. Another advantage of **Asmeta** framework is to allow the modelling of the systems using the refinement technique. It helps to manage the complexity of the development process in case of complex system (as the hemodialysis machine).

---

---

## Conclusions and Future Work

---

In this thesis, we have proposed a formal method to perform validation and verification of ASMs models. Furthermore, a code generator from ASMs to C++ for Arduino platform is introduced. Subsequently, the process is applied to two case studies: stereoacuity software and hemodialysis machine. Due to the criticality of medical software we have identified standards and regulations for the certification of the software. Moreover, before starting the work on this topic, we applied SLR to understand the current applications of formal methods to medical devices. The works presented in this thesis have been published in different forms in journal papers, conference papers, and book chapters. The list of publications is reported in Appendix B.

In the next sections, we summarise the contribution of the thesis and describe some possible future works.

### Contributions of the thesis

---

In Part I, we described the regulations for medical device software certification (see Chapter 2) and which is the state of the art of formal methods to certify software in medical devices (see Chapter 3).

In Chapter 2, we presented standards and guidelines applicable to produce certifiable software. It is the responsibility of competent authorities to release the certification based on the software documentation and on the results of some specific tests.

In systematic literature review (see Chapter 3), 71 publications are analysed. Quantitative and qualitative analysis (see Sect. 3.1.2) are performed to provide information that can help researchers working within this domain. We found that the number of publications per year is still growing due to the critical issue of the topic. Due to the novelty of the topic, a lot of authors that have published only once and only a few authors have published more than two papers. The communities have applied several formal methods, the most used are automata, Event-B, Z and EFSM (Tables 3.3). The notations are applied to distinct case studies which results are summarized in Table 3.4. Each notations is supported by

tools to perform different activities which are summarised in Table 3.5. Analysing the table, the most used tools are B tools, MATLAB and Simulink, and UPPAAL.

Chapter 4 is a background chapter. We proposed a process to model, to validate and to verify a system using ASMs. After that, we shown the compliance of ASM process with the standard and regulation, presented in Chapter 2, to develop medical software: IEC62304 and FDA principles. The main advantages offered to the standards are: (i) an iterative software life cycle; (ii) models as a rigorous means for safety properties assurance; (iii) validation and verification performed continuously along the software life cycle, and always aimed at defect prevention; (iv) software quality evaluation performed in an objective and repeatable manner; and (v) demonstration that software has been validated and verified.

In ASMs community, there are different notations and tools to perform modelling, validation and verification, but it is not possible to interoperate with the tools. We tried to deal with this problem by implementing a common language UASM (see Chapter 5) that the community can use to share models. For this reason, we have developed a parser and an editor for UASM models. Furthermore, we have developed the translator from UASM to *AsmetaL* that allow to the user of UASM to use also the tools of *Asmeta* framework to perform different analysis.

A tool for visualization of ASMs models is presented in Chapter 6 to improve the readability and understandability of ASMs models. We have proposed a graphical notation for ASMs, and we have defined visual patterns that capture, in a concise way, different recurring ASMs rule patterns. The representation concerns only the transition rules and not the signature of the model.

In Chapter 7 we have presented a code generator to translate formal specification to code automatically. The tool generates C++ from formal specifications written in UASM dialect. The source code is obtained from requirement models by applying a set of M2T transformations. Since the analysis tools accept formal specifications written in *AsmetaL*, the translator from UASM to *AsmetaL* is used for the V&V activities. In this way, the specification can be analysed using *Asmeta* framework tools and subsequently it can be translated into C++ code.

In Part III, we presented two case studies: a medical software used for measuring the stereoacuity and a hemodialysis machine. For both case studies, we followed the process presented in Chapter 4; starting from the specification we applied validation (simulation and model review) and property verification (model checking) by using model refinement. Finally, the conformance of the implementation w.r.t. the specification can be checked through testing and runtime verification.

### Future work

---

We here describe some of the possible improvements of the works presented in this thesis.

**Systematic Literature Review** The work presented in this thesis analyses papers until the end of 2016. In the results of the analysis the newest papers are missed. The next step is to integrate continuously the new papers into the SLR process defined in Figure 3.1. The results will help to monitor the current state



of the art about the application of formal methods to medical devices. We will also investigate new techniques to include the major number of papers about the topic of the SLR.

**Standards and regulations** Standards and regulations are continuously updated to fit the current state of the medical software that is continuously in evolution. We plan to monitor the evolution of the standards.

**ASMs** The ASM process has been compared with the requirements of the standards. The next step will be to better define how the ASMs can be integrated to make them completely conform to the regulations. Furthermore, it is interesting to understand how ASMs can help the user during the software certification process.

**Visualization for ASMs** Currently the tool translates the textual representation of the ASMs into the graphical representation making available some patterns. As future work, we plan to identify new visual patterns and to define visual trees for all the turbo rules. Regarding the tool, we plan to implement a second usage: from graphical to textual representation. It consists in graphically specifying the ASMs by drawing the graph. In this way, the modeller can focus on the high level structure of the model. Once the modeller has produced the graph, a **translator** can translate the graph in an AsmetaL textual model. Then, the AsmetaL parser may find some faults that passed undetected during graph validation.

**Code generation** At the moment the generator misses the translation of some rules and constructors, the future work will be to complete the tool. Furthermore, we plan to extend the tool with an automatic test cases generator. From the ASMs specification, a series of tests can be automatically generated and they could be executed on the Arduino board. This would test both the system and the translation from the specification to the code. We plan to work on the validation of the translation itself. We have tried to define our mapping from specification concepts (like state and rules) to source code as clear as possible and we are convinced of its correctness, i.e., that the generated code preserves the same behaviour as described by the original specification. As future work, we plan to work on proving the correctness of the code transformation. At present in Chapter 7, an informal prove of the transformation correctness is shown. Considering code generator, we are planning to develop a code generator from ASMs to Java by defining a mapping of ASMs elements to Java language constructs.

**Case study** FDA provides a complete description of a infusion pump case study<sup>2</sup>, in order to allow the application of different approaches, increase user awareness, proactively facilitate device improvements, and publish new guidance for industry. We want to apply the approach presented in Chapter 4 to check how

<sup>2</sup><http://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/default.htm>

## Chapter 9. Hemodialysis machine case study

---

it behaves in a real application and we will compare the results with the other available in literature.

---

# APPENDIX *A*

---

## IEC and ISO deliverables

---

IEC and ISO do not develop only standards, but also other documents that users can follow as a guide.

### IEC deliverables

---

IEC Technical Committees and Subcommittees (TCs/SCs) develop International Standards and other type of publications. They are divided into two categories:

- Normative publications: provide agreements on the technical description about product characteristics;
- Informative publications: provide information such as procedures or guidelines.

Furthermore the IEC publishes the following deliverables:

- Technical Specifications (TS)
- Publicly Available Specification (PAS)
- Amendments
- Technical Corrigenda
- Interpretation Sheets
- Technical Report (TR)
- Guides

**International Standards** is a normative document which has been approved by IEC National Committee members and it is adopted by international standards organization.

**Technical Specifications (TS)** has same details and information as International Standards, but have not yet passed through all approval stages or the subject under question is still under development.

**Publicly Available Specification (PAS)** is a publication responding to an urgent market need.

**Amendments** is a normative document and alters and/or adds technical provisions in an existing International Standard.

**Technical Corrigenda** is issued to correct a technical error or ambiguity in an International Standard, TS, PAS or TR; or to correct information that has become outdated since publication.

**Interpretation Sheets** provides a quick formal explanation to an urgent request by users.

**Technical Report (TR)** is an informal document that contains data different from those published in International Standard (e.g. data obtained from a survey, data of work, state of the art in relation to standards).

**Guides** give rules, orientation, advice or recommendations relating to international standardization and conformity assessment.

### ISO deliverables

---

ISO technical committees deliver International Standards and the following deliverables:

- New work item Proposal (NP)
- Committee draft (CD) or Publicly Available Specification (PAS)
- Technical Specification (TS)
- Technical Report (TR)
- Final Draft International Standard (FDIS)
- International Workshop Agreement (IWA)
- Guide

**ISO Standards** are normative documents, which have been approved by the ISO membership and members of the responsible committee.

**Committee draft (CD) or Publicly Available Specification (PAS)** PAS is a normative document representing the consensus within a working group. It shall be review at least every three years and, after six years, PAS shall either be converted into an International Standard or be withdrawn.

**Technical Specification (TS)** is a normative document representing the technical consensus within an ISO committee. This document is drawn up in cases in which there was insufficient support for the publication of a standards. TS shall be review at least every three years and, after six years, it shall either be converted into an International Standard or be withdrawn.

**Technical Report (TR)** is an informative document containing information of a different kind from that normally published in a normative document.

**International Workshop Agreement (IWA)** is an ISO document produced through workshop meeting(s).

**Guides** provide guidance to technical committees for the preparation of standards.

---

## Publications

---

1. **A Low-cost Virtual Reality Game for Amblyopia Rehabilitation**  
(co-authors: A. Gargantini, F. Terzi, M. Zambelli)  
in *Proceedings of the 3rd 2015 Workshop on ICTs for Improving Patients Rehabilitation Research Techniques*, ACM, (2015): 81–84, 978-1-4503-3898-1  
**Abstract:** The paper presents the design and development of a mobile application realizing a video game that aims at treating amblyopia by using a Google Cardboard. Google Cardboard is a low cost device able to reproduce virtual reality by means of a smartphone. The proposed video game engaged the patient in a car racing game and it displays the same image to the eyes, but with some differences that stimulate the lazy eye more than the normal eye.
2. **A Mobile Application for the Stereoacuity Test**  
(co-authors: A. Gargantini, A. Vitali)  
in *Digital Human Modeling. Applications in Health, Safety, Ergonomics and Risk Management: Ergonomics and Health: 6th International Conference, DHM 2015, Held as Part of HCI International 2015*, LNCS, Springer International Publishing, (2015): 315–326, 978-3-319-21070-4  
**Abstract:** The research paper concerns the development of a new mobile application emulating measurements of stereoacuity using Google Cardboard. Stereoacuity test is based on binocular vision that is the skill of human beings and most animals to recreate depth sense in visual scene. Google Cardboard is a very low cost device permitting to recreate depth sense of images showed on the screen of a smartphone. Proposed solution exploits Google Cardboard to recreate and manage depth sense through our mobile application that has been developed for Android devices. First, we describe the research context

as well as the aim of our research project. Then, we introduce the concept of stereopsis and technology used for emulating stereoacuity test. Finally, we portray preliminary tests made so far and achieved results are discussed.

### 3. **A preliminary systematic literature review of the use of formal methods in medical software systems**

(co-authors: A. Gargantini, A. Mashkoor)

in *Industrial Proceedings of the 23rd European & Asian System, Software & Service Process Improvement & Innovation (EuroAsiaSPI2 2016)*, (2016): 9.15–9.23, 978-87-998116-6-3

**Abstract:** The use of formal methods is often recommended to guarantee the provision of necessary services and to assess the correctness of critical properties, such as safety, security and reliability, in medical and healthcare systems. Several research groups have proposed and applied formal methods related techniques to the design and development of medical software and systems. However, a systematic and inclusive survey with some form of analysis is still missing in this domain. For this reason, we have collected the relevant literature on the use of formal methods to the modelling, design, development, verification and validation of medical software systems. We apply the well-known systematic literature review technique and we run several queries in order to obtain information that can be useful for people working in this area. We present some research questions and the data answering these questions. We also discuss some limitations of the adopted approach and how to address these issues in order to have a comprehensive survey.

### 4. **Asm2C++: a tool for Code Generation from Abstract State Machines to Arduino**

(co-authors: M. Carisconi, A. Gargantini, A. Mashkoor)

in *NASA Formal Methods Symposium (NFM) 2017*

**Abstract:** This paper presents **Asm2C++**, a tool that automatically generates executable C++ code for Arduino from a formal specification given as Abstract State Machines (ASMs). The code generation process follows the model-driven engineering approach, where the code is obtained from a formal abstract model by applying certain transformation rules. The translation process is highly configurable in order to correctly integrate the underlying hardware. The advantage of the **Asm2C++** tool is that it is part of the **Asmeta** framework that allows to analyze, verify, and validate the correctness of a formal model.

### 5. **Formal validation and verification of a medical software critical component**

(co-authors: P. Arcaini, A. Gargantini, A. Mashkoor, E. Riccobene)

in *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference*, ACM, (2015): 80–89, 978-1-5090-0237-5

**Abstract:** Medical device software malfunctioning can lead to injuries or death for humans and, therefore, its development should adhere to certification standards. However, these standards establish general guidelines on the use of common software engineering activities without any indication re-

---

garding methods and techniques to assure safety and reliability. This paper presents a formal development process, based on the Abstract State Machine method, that integrates most of the activities required by the standards. The process permits to obtain, through a sequence of refinements, more detailed models that can be formally validated and verified. Offline and online testing techniques permit to check the conformance of the implementation w.r.t. the specification. The process is applied to the validation of the SAM medical software, that is used to measure the patients' stereoacuity in the diagnosis of amblyopia.

## 6. How to Assure Correctness and Safety of Medical Software: The Hemodialysis Machine Case Study

(co-authors: P. Arcaini, A. Gargantini, E. Riccobene)

in *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016*, LNCS, Springer International Publishing, (2016): 344–359, 978-3-319-33600-8

**Abstract:** Medical devices are nowadays more and more software dependent, and software malfunctioning can lead to injuries or death for patients. Several standards have been proposed for the development and the validation of medical devices, but they establish general guidelines on the use of common software engineering activities without any indication regarding methods and techniques to assure safety and reliability. This paper takes advantage of the Hemodialysis machine case study to present a formal development process supporting most of the engineering activities required by the standards, and provides rigorous approaches for system validation and verification. The process is based on the Abstract State Machine formal method and its model refinement principle.

## 7. Unified Syntax for Abstract State Machines

(co-authors: P. Arcaini, M. Dausend, A. Gargantini, A. Mashkoo, A. Raschke, E. Riccobene, P. Scandurra, M. Stegmaier)

in *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016*, LNCS, Springer International Publishing, (2016): 231–236, 978-3-319-33600-8

**Abstract:** The paper presents our efforts in defining UASM, a unified syntax for Abstract State Machines (ASMs), based on the syntaxes of two of the main ASM frameworks, CoreASM and Asmeta, which have been adapted to accept UASM as input syntax of all their validation and verification tools.

## 8. Visual Notation and Patterns for Abstract State Machines

(co-authors: P. Arcaini, A. Gargantini, E. Riccobene)

in *Software Technologies: Applications and Foundations: STAF 2016 Collocated Workshops: DataMod, GCM, HOFM, MELO, SEMS, VeryComp*, LNCS, Springer International Publishing, (2016): 163–178, 978-3-319-50230-4

**Abstract:** Formal models are a rigorous way to specify informal system requirements. However, they are not widely used in practice, since they are considered difficult to develop and understand. Visualization is often

considered a good means for people to communicate and to get a common understanding. We here make a proposal of a visual notation for Abstract State Machines (ASMs), and we introduce *visual trees* that visualize ASM transition rules. In addition to these graphical components that are based only on the syntactical structure of the model, we also present *visual patterns* that permit to visualize part of the behavior of the machine. A tool is also available to graphically represent ASM models using the proposed notation.



---

---

## List of Acronyms

---

<b>AP</b>	Arterial Pressure
<b>ASMs</b>	Abstract State Machines
<b>BP</b>	Blood Pump
<b>CTL</b>	Computational Tree Logic
<b>DB</b>	Data Base
<b>EBC</b>	Extracorporeal Blood Circuit
<b>EBNF</b>	Extended Backus-Naur form
<b>IEC</b>	International Electrotechnical Commission
<b>ISO</b>	International Organization for Standardization
<b>LTL</b>	Linear Temporal Logic
<b>MBT</b>	Model Based Testing
<b>PEMS</b>	Programmable Electronic Medical Systems
<b>RV</b>	Runtime Verification
<b>SLR</b>	Systematic Literature Review
<b>SMT</b>	Satisfiability Modulo Theories
<b>UASM</b>	Unified Abstract State Machines
<b>VP</b>	Venous Pressure
<b>VRD</b>	Venous Red Detector
<b>V&amp;V</b>	Validation and Verification



---

## List of Figures

---

2.1	Integration of standards with IEC 62304:2006 . . . . .	13
3.1	SLR Process . . . . .	19
3.2	RQ1: Trend of publications . . . . .	24
3.3	RQ2: Types of publications . . . . .	24
3.4	RQ2: Publications in Journal/Proceedings per year . . . . .	25
3.5	RQ3: Number of papers by the same author . . . . .	25
3.6	RQ4: Number of citations/publication . . . . .	26
4.1	ASMs State-Transition . . . . .	37
4.2	ASMs development process . . . . .	39
5.1	From UASM specification to AsmetaL specification . . . . .	69
6.1	Visual notation . . . . .	75
6.2	Structural pattern – Nested guards pattern . . . . .	77
6.3	Semantic pattern – Mutual exclusive guards pattern . . . . .	78
6.4	Semantic pattern – State pattern . . . . .	79
6.5	Semantic pattern – State flow pattern . . . . .	80
6.6	Example 1: Semantic pattern – State flow pattern . . . . .	81
6.7	Example 2: Semantic pattern – State flow pattern . . . . .	81
6.8	AsmetaVis tool . . . . .	83
6.9	Example 1: visual notation using the tool - in red the navigations links . . . . .	83
7.1	M2T Transformation . . . . .	86
7.2	M2M Transformation . . . . .	87
7.3	Flow Diagram: transformation process from UASM specification to Arduino project. . . . .	90
7.4	Parsing process . . . . .	91
7.5	Code generator process . . . . .	91
7.6	Merge process . . . . .	92

## List of Figures

---

7.7	ASM state transition . . . . .	93
7.8	ASM and C++ correspondence . . . . .	93
7.9	Plug-in screenshot . . . . .	99
7.10	Control system architecture . . . . .	100
7.11	Snippets from model and code . . . . .	105
8.1	3D technologies . . . . .	113
8.2	StereoAcuity tests . . . . .	115
8.3	Lancaster test and Aniseikonia measurement . . . . .	115
8.4	Treatments for visual diseases . . . . .	116
8.5	Stuttering refinement from the second refinement model to the third refinement model – Example of refined run . . . . .	119
8.6	Wrap class for testing . . . . .	122
9.1	Hemodialysis machine: schema . . . . .	127
9.2	Ground model basic visualization . . . . .	131
9.3	Ground model semantic visualization . . . . .	131
9.4	Ground model phases . . . . .	132
9.5	Preparation rule: semantic visualization . . . . .	132
9.6	Treatment parameters . . . . .	133
9.7	Hemodialysis case study – Relation between a refined run and an abstract run . . . . .	134
9.8	Check temperature upper limit . . . . .	134
9.9	Simulation trace of first refinement model . . . . .	135
9.10	Second refinement – Semantic visualization . . . . .	138
9.11	Third refinement – Semantic visualization . . . . .	144
9.12	Hemodialysis device program GUI . . . . .	146
9.13	Abstract test sequence . . . . .	147
9.14	JUnit testing results . . . . .	148

---

---

## List of Tables

---

2.1	IEC 62304 software development process . . . . .	12
3.1	Number of queries and entries for each repository . . . . .	22
3.2	Publications with most citations . . . . .	27
3.3	Notations used in the literature . . . . .	28
3.4	Application of formal methods to medical devices . . . . .	32
3.5	Tools used for each methodology . . . . .	33
6.1	$\text{vis}_T$ : Mapping from ASM transition rules to visual trees . . . . .	76
6.2	Experimental results of preliminary evaluation of visual notation . . . . .	84
7.1	ASM and C++ correspondence . . . . .	94
7.2	Parallelism: translation in C++ . . . . .	95
7.3	Nondeterminism: translation in C++ . . . . .	96
7.4	Domain definition: translation from UASM to C++ . . . . .	97
7.5	Rules: translation from UASM to C++ . . . . .	106
7.6	Terms: translation from UASM to C++ . . . . .	107
8.1	3D4AMB applications and technologies . . . . .	113
8.2	Code coverage . . . . .	123
9.1	Hemodialysis device phases . . . . .	130



---

---

## Bibliography

---

- [1] ISO 13485:2003 – Medical devices – Quality management systems – Requirements for regulatory purposes, 2003.
- [2] IEC 60601-1:2005 – Medical electrical equipment – Part 1: General requirements for basic safety and essential performance, 2005.
- [3] IEC 62304:2006 – Medical device software – Software lifecycle processes, 2006.
- [4] ISO 14971:2007 Medical devices – Application of risk management to medical devices, 2007.
- [5] ISO/IEC 12207:2008 – Systems and software engineering – Software life cycle processes, 2008.
- [6] IEC/TR 80002-1:2009 – Medical device software – Part 1: Guidance on the application of ISO 14971 to medical device software, 2009.
- [7] IEC 61010-1:2010 – Safety requirements for electrical equipment for measurement, control, and laboratory use - Part 1: General requirements, 2010.
- [8] IEC 61508-3:2010 – Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 3: Software requirements, 2010.
- [9] ISO/IEC TR 24774:2010 – Systems and software engineering – Life cycle management – Guidelines for process description, 2010.
- [10] IEC/TR 80002-3:2014 – Medical device software – Part 3: Process reference model of medical device software life cycle processes, 2014.
- [11] ISO/IEC 90003:2014 – Software engineering – Guidelines for the application of ISO 9001:2008 to computer software, 2014.
- [12] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

- [13] A.J. Abbate, A.L. Throckmorton, and E.J. Bass. A Formal Task-Analytic Approach to Medical Device Alarm Troubleshooting Instructions. *IEEE Transactions on Human-Machine Systems*, 46(1):53–65, 2016.
- [14] R. Alur, D. Arney, E.L. Gunter, I. Lee, J. Lee, W. Nam, F. Pearce, S. Van Albert, and J. Zhou. Formal specifications and analysis of the computer-assisted resuscitation algorithm (CARA) Infusion Pump Control System. *International Journal on Software Tools for Technology Transfer*, 5(4):308–319, 2004.
- [15] P. Arcaini, S. Bonfanti, A. Gargantini, A. Mashkoor, and E. Riccobene. Formal validation and verification of a medical software critical component. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 80–89. Institute of Electrical and Electronics Engineers Inc., 2015.
- [16] P. Arcaini, S. Bonfanti, A. Gargantini, and E. Riccobene. How to assure correctness and safety of medical software: The hemodialysis machine case study. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - ABZ 2016*, volume 9675, pages 344–359. Springer International Publishing, 2016.
- [17] P. Arcaini, S. Bonfanti, A. Gargantini, and E. Riccobene. Visual notation and patterns for abstract state machines. In Paolo Milazzo, Dániel Varró, and Manuel Wimmer, editors, *Software Technologies: Applications and Foundations: STAF 2016 Collocated Workshops: DataMod, GCM, HOFM, MELO, SEMS, VeryComp, Vienna Austria, July 4-8, 2016*, LNCS, pages 163–178. Springer International Publishing, 2016.
- [18] P. Arcaini, S.a Bonfanti, M. Dausend, A.o Gargantini, A. Mashkoor, A. Raschke, E. Riccobene, P. Scandurra, and M. Stegmaier. Unified syntax for abstract state machines. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016*, LNCS, pages 231–236. Springer International Publishing, 2016.
- [19] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In *Proceedings of the 2nd International Conference on Abstract State Machines, Alloy, B and Z (ABZ 2010)*, volume 5977 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2010.
- [20] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Automatic Review of Abstract State Machines by Meta Property Verification. In *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010)*, pages 4–13. NASA, 2010.
- [21] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. CoMA: Conformance monitoring of Java programs by Abstract State Machines. In *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 2012.



- 
- [22] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Combining model-based testing and runtime monitoring for program testing in the presence of nondeterminism. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Workshops Proceedings, Luxembourg, March 18-22, 2013*, pages 178–187. IEEE, 2013.
- [23] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Rigorous development process of a safety-critical system: from ASM models to Java code. *International Journal on Software Tools for Technology Transfer*, pages 1–23, 2015.
- [24] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. SMT-based automatic proof of ASM model refinement. In *Software Engineering and Formal Methods: 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*, Lecture Notes in Computer Science, pages 253–269. Springer International Publishing, Cham, 2016.
- [25] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41:155–166, 2011.
- [26] Paolo Arcaini, Roxana-Maria Holom, and Elvinia Riccobene. ASM-based formal design of an adaptivity component for a cloud system. *Formal Aspects of Computing*, 28(4):567–595, 2016.
- [27] D. Arney, J.M. Goldman, S.F. Whitehead, and I. Lee. Improving patient safety with X-Ray and anesthesia machine ventilator synchronization: A medical device interoperability case study. *Communications in Computer and Information Science*, 52:96–109, 2010.
- [28] D. Arney, R. Jetley, P. Jones, I. Lee, and O. Sokolsky. Formal methods based development of a PCA infusion pump reference model: Generic infusion pump (GIP) project. In *High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability*, pages 23–33, Cambridge, MA, June 2007.
- [29] D. Arney, I. Lee, J.M. Goldman, and S.F. Whitehead. Synchronizing an x-ray and anesthesia machine ventilator: A medical device interoperability case study. In *BIODEVICES 2009 - Proceedings of the 2nd International Conference on Biomedical Electronics and Devices*, pages 52–60, 2009.
- [30] M.W. Azeem, M. Ahsan, N.M. Minhas, and K. Noreen. Specification of e-health system using z: A motivation to formal methods. In *Convergence of Technology (I2CT), International Conference for Convergence for Technology 2014*. Institute of Electrical and Electronics Engineers Inc., 2014.
- [31] S.M. Babamir and M. Borhani. Formal verification of medical monitoring software using Z language: A representative sample. *Journal of Medical Systems*, 36(4):2633–2648, 2012.

## Bibliography

---

- [32] M. Balsler, O. Coltell, J. Van Croonenborg, C. Duelli, F. Van Harmelen, A. Jovell, P. Lucas, M. Marcos, S. Miksch, W. Reif, K. Rosenbrand, A. Seyfang, and A. Ten Teije. Protocure: Supporting the development of medical protocols through formal methods. In *Studies in Health Technology and Informatics*, volume 101, pages 103–107, 2004.
- [33] R. Banach. Hemodialysis machine in Hybrid Event-B. In M. Butler, K.-D. Schewe, A. Mashkoor, and M. Biro, editors, *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - ABZ 2016*, volume 9675, pages 376–393. Springer Verlag, 2016.
- [34] A. Banerjee, Y. Zhang, P. Jones, and S. Gupta. Using formal methods to improve home-use medical device safety. *Biomedical Instrumentation and Technology*, 47(SPRING):43–48, 2013.
- [35] B. Barbot, M. Kwiatkowska, A. Mereacre, and N. Paoletti. Estimation and verification of hybrid heart models for personalised medical and wearable devices. In Bourdon J. Roux O., editor, *International Conference on Computational Methods in Systems Biology - CMSB 2015*, volume 9308, pages 3–7. Springer Verlag, 2015.
- [36] E. Bartocci, F. Corradini, R. Grosu, E. Merelli, O. Riganelli, and S.A. Smolka. StonyCam: A formal framework for modeling, analyzing and regulating cardiac myocytes. *Concurrency, Graphs and Models*, 5065 LNCS:493–502, 2008.
- [37] A. Blandford, A. Cauchi, P. Curzon, P. Eslambolchilar, D. Furniss, A. Gimblett, H. Huang, P. Lee, Y. Li, P. Masci, P. Oladimeji, A. Rajkomar, R. Rukšėnas, and H. Thimbleby. Comparing actual practice and user manuals: A case study based on programmable infusion pumps. In *Intl. Workshop on Engineering Interactive Computing Systems for Medicine and Health Care.*, volume 727, pages 59–64, 2011.
- [38] S. Bonfanti, M. Carisconi, A. Gargantini, and A. Mashkoor. Asm2C++: a tool for Code Generation from Abstract State Machines to Arduino. In *NASA Formal Methods Symposium*, 2017.
- [39] S. Bonfanti, A. Gargantini, and A. Mashkoor. A preliminary systematic literature review of the use of formal methods in medical software systems. In *23rd EuroAsiaSPI Conference, Graz University of Technology, Graz, Austria*, 2016.
- [40] Silvia Bonfanti, Angelo Gargantini, and Andrea Vitali. A mobile application for the stereoacuity test. In G. Vincent Duffy, editor, *Digital Human Modeling. Applications in Health, Safety, Ergonomics and Risk Management: Ergonomics and Health: 6th International Conference, DHM 2015, Held as Part of HCI International 2015, Los Angeles, CA, USA, August 2-7, 2015, Proceedings, Part II*, LNCS, pages 315–326. Springer International Publishing, 2015.

- 
- [41] Egon Börger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
- [42] J. Bowen and S. Reeves. Modelling user manuals of modal medical devices and learning from the experience. In *EICS'12 - Proceedings of the 2012 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 121–130, 2012.
- [43] J. Bowen and S. Reeves. Modelling safety properties of interactive medical systems. In *ACM SIGCHI symposium on Engineering interactive computing systems*, pages 91–100, 2013.
- [44] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.
- [45] Barrett R Bryant, Jeff Gray, Marjan Mernik, Peter J Clarke, Robert B France, and Gabor Karsai. Challenges and directions in formalizing the semantics of modeling languages. *Computer Science and Information Systems*, 8(2):225–253, 2011.
- [46] Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A Scenario-Based Validation Language for ASMs. In *Proceedings of the 1st International Conference on Abstract State Machines, B and Z (ABZ 2008)*, volume 5238 of *Lecture Notes in Computer Science*, pages 71–84. Springer-Verlag, 2008.
- [47] M. Carissoni. Automatic Code Generation from Formal Specification: UASM to C++ for Arduino. Master’s thesis, University of Bergamo, 2016.
- [48] V. Cehlot and E.B. Sloane. Ensuring patient safety in wireless medical device networks. *Computer*, 39(4):54–60, April 2006.
- [49] L. Cordeiro, B. Fischer, H. Chen, and J. Marques-Silva. Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In *Proceedings - 2009 International Conference on Embedded Software and Systems, ICESS 2009*, pages 396–403, 2009.
- [50] Council of the European Communities. Council Directive 93/42/EEC, 1993.
- [51] P. Curran and K. Norrie. An approach to verifying concurrent systems—a medical information bus (MIB) case study. In *Computer-Based Medical Systems*, pages 74–83, Jun 1992.
- [52] P. Curzon, P. Masci, P. Oladimeji, R. Rukšėnas, H. Thimbleby, and E. D’Urso. Human-Computer Interaction and the Formal Certification and Assurance of Medical Devices: The CHI+ MED Project. *EPSRC Programme Grants*, 2014.
- [53] Z. Daw, R. Cleaveland, and M. Vetter. Formal verification of software-based medical devices considering medical guidelines. *International Journal of Computer Assisted Radiology and Surgery*, 9(1):145–153, 2014.

## Bibliography

---

- [54] N. Decker, F. Kuhn, and D. Thoma. Runtime verification of web services for interconnected medical devices. In *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pages 235–244. IEEE Computer Society, 2014.
- [55] J. Ding and X. He. Formal specification and analysis of an agent-based medical image processing system. *International Journal of Software Engineering and Knowledge Engineering*, 20(3):311–345, 2010.
- [56] Serdar Doğan, Aysu Betin-Can, and Vahid Garousi. Web application testing: A systematic literature review. *J. Syst. Softw.*, 91:174–201, May 2014.
- [57] Nicolas Dulac, Thomas Viguier, Nancy Leveson, and Margaret-Anne Storey. On the use of visualization in formal requirements specification. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 71–80. IEEE, 2002.
- [58] EU. Directive 2007/47/EC of the European Parliament and of the Council. Official Journal of the European Union, September 2007.
- [59] Giancarlo Facchetti, Angelo Gargantini, and Andrea Vitali. An environment for domestic supervised amblyopia treatment. In VincentG. Duffy, editor, *Digital Human Modeling. Applications in Health, Safety, Ergonomics and Risk Management*, volume 8529 of *Lecture Notes in Computer Science*, pages 340–350. Springer International Publishing, 2014.
- [60] T. Fayolle, M.a Frappier, F.b Gervais, and R.b Laleau. Modelling a hemodialysis machine using Algebraic state-Transition Diagrams and B-like methods. In M. Butler, K.-D. Schewe, A. Mashkoor, and M. Biro, editors, *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - ABZ 2016*, volume 9675, pages 394–408. Springer Verlag, 2016.
- [61] Angelo Gargantini, Mariella Bana, and Flavia Fabiani. Using 3D for rebalancing the visual system of amblyopic children. In *Virtual Rehabilitation (ICVR), 2011 International Conference on*, pages 1–7, june 2011.
- [62] Angelo Gargantini, Giancarlo Facchetti, and Andrea Vitali. A random dot stereoacuity test based on 3d technology. In *Proceedings of the 8th International Conference on Pervasive Computing Technologies for Healthcare, PervasiveHealth '14*, pages 358–361, ICST, Brussels, Belgium, Belgium, 2014. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
- [63] Angelo Gargantini and Elvinia Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation. *Journal of Universal Computer Science*, 7:262–265, 2001.
- [64] Angelo Gargantini, Elvinia Riccobene, and Salvatore Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In *Abstract State Machines 2003*, volume 2589 of *Lecture Notes in Computer Science*, pages 263–277. Springer Berlin Heidelberg, 2003.

- 
- [65] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *J. UCS*, 14(12):1949–1983, 2008.
- [66] Angelo Gargantini, Fabio Terzi, Matteo Zambelli, and Silvia Bonfanti. A low-cost virtual reality game for amblyopia rehabilitation. In *Proceedings of the 3rd 2015 Workshop on ICTs for improving Patients Rehabilitation Research Techniques*, pages 81–84. ACM, 2015.
- [67] A.O. Gomes and A. Butterfield. Modelling the haemodialysis machine with Circus. In M. Butler, K.-D. Schewe, A. Mashkoor, and M. Biro, editors, *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - ABZ 2016*, volume 9675, pages 409–424. Springer Verlag, 2016.
- [68] A.O. Gomes and M.V.M. Oliveira. Formal specification of a cardiac pacing system. In *International Symposium on Formal Methods - FM 2009*, volume 5850 LNCS, pages 692–707, 2009.
- [69] A.O. Gomes and M.V.M. Oliveira. Formal development of a cardiac pacemaker: From specification to code. In *Brazilian Symposium on Formal Methods - SBMF 2010: Formal Methods: Foundations and Applications*, volume 6527 LNCS, pages 210–225, 2011.
- [70] P. Groot, A. Hommersom, P. Lucas, M. Balsler, and J. Schmitt. Experiences in quality checking medical guidelines using formal methods. In *Proceedings Verification and Validation of Software Systems (VVSS 2007) March 23, 2007*, pages 164–78, 2007.
- [71] Yuri Gurevich. Specification and validation methods. In Egon Börger, editor, *Specification and validation methods*, chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.
- [72] M.D.a Harrison, J.C.b Campos, R.c Rukšenas, and P.c Curzon. Modelling information resources and their salience in medical device design. In *Conference of 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2016*, pages 194–203. Association for Computing Machinery, Inc, 2016.
- [73] Rob Hierons and John Derrick. Editorial: special issue on specification-based testing. *Software Testing, Verification and Reliability*, 10(4):201–202, 2000.
- [74] Julian PT Higgins and Sally Green. *Cochrane handbook for systematic reviews of interventions*, volume 4. John Wiley & Sons, 2011.
- [75] T.S.a Hoang, C.a Snook, L.b Ladenberger, and M.a Butler. Validating the requirements and design of a hemodialysis machine using iUML-B, BMotion studio, and co-simulation. In M. Butler, K.-D. Schewe, A. Mashkoor, and M. Biro, editors, *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - ABZ 2016*, volume 9675, pages 360–375. Springer Verlag, 2016.

- [76] D.H. Hoglund. Validation and verification of WLAN medical devices. *Biomedical instrumentation & technology / Association for the Advancement of Medical Instrumentation*, Suppl:79–83, 2012.
- [77] J. Jacky. Specifying a safety-critical control system in Z. *IEEE Transactions on Software Engineering*, 21(2):99–106, Feb 1995.
- [78] R. Jetley, S.P. Iyer, and P. Jones. A formal methods approach to medical device review. *Computer*, 39(4):61–67, 2006.
- [79] R. Jetley, S. Purushothaman Iyer, P. Jones, and W. Spees. A formal approach to pre-market review for medical device software. In *Annual International Computer Software and Applications Conference*, volume 1, pages 169–177, Chicago, IL, 2006.
- [80] R.P. Jetley, C. Carlos, and S.P. Iyer. A case study on applying formal methods to medical devices: Computer-aided resuscitation algorithm. *International Journal on Software Tools for Technology Transfer*, 5(4):320–330, 2004.
- [81] Z. Jiang, H. Abbas, K. J. Jang, and R. Mangharam. The Challenges of High-Confidence Medical Device Software. *Computer*, 49(1):34–42, Jan 2016.
- [82] Z. Jiang, M. Pajic, R. Alur, and R. Mangharam. Closed-loop verification of medical devices with model abstraction and refinement. *International Journal on Software Tools for Technology Transfer*, 16(2):191–213, 2014.
- [83] Soon-Kyeong Kim and David Carrington. Visualization of formal specifications. In *Proceedings of APSEC'99*, pages 102–109. IEEE, 1999.
- [84] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering - a systematic literature review. *Inf. Softw. Technol.*, 51(1):7–15, January 2009.
- [85] M. Kwiatkowska, H. Lea-Banks, A. Mereacre, and N. Paoletti. Formal modelling and validation of rate-adaptive pacemakers. In *Healthcare Informatics (ICHI) IEEE International Conference on*, pages 23–32. Institute of Electrical and Electronics Engineers Inc., 2014.
- [86] B. R. Larson. Formal semantics for the PACEMAKER System Specification. In *ACM SIGAda Ada Letters*, pages 47–59. Association for Computing Machinery, Inc, 2014.
- [87] J. Leemans and N. Amálio. Modelling a cardiac pacemaker visually and formally. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, pages 257–258. IEEE, 2012.
- [88] M. Leucker, M. Schmitz, and D. à Tellinghusen. Runtime verification for interconnected medical devices. In Margaria T. Steffen B., editor, *International Symposium on Leveraging Applications of Formal Methods - ISoLA 2016*, volume 9953 LNCS, pages 380–387. Springer Verlag, 2016.

- 
- [89] C. Li, A. Raghunathan, and N.K. Jha. Improving the trustworthiness of medical device software with formal verification methods. *IEEE Embedded Systems Letters*, 5(3):50–53, Sept 2013.
- [90] S. Lämmermann, J. Ruf, A. B. L. Pielawa, J. T. Kropf, W. R. Schlemminger, and A. Hein. Heterogeneous Assertion-Based Verification for Medical Devices Development. In *17th Workshop on Synthesis And System Integration of Mixed Information Technologies (Sasimi 2012)*, 2012.
- [91] P. C. Ölveczky. Towards formal modeling and analysis of networks of embedded medical devices in Real-Time Maude. In *Proc. 9th ACIS Int. Conf. Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2008 and 2nd Int. Workshop on Advanced Internet Technology and Applications*, pages 241–248, Phuket, 2008.
- [92] John J. Majikes, Rahul Pandita, and Tao Xie. Literature Review of Testing Techniques for Medical Device Software. In *Proceedings of the Medical Cyber Physical Systems Workshop*, 2013.
- [93] P. Masci, A. Ayoub, P. Curzon, I. Lee, O. Sokolsky, and H. Thimbleby. Model-based development of the generic PCA infusion pump user interface prototype in PVS. In *International Conference on Computer Safety, Reliability, and Security - SAFECOMP 2013: Computer Safety, Reliability, and Security*, volume 8153 LNCS, pages 228–240, 2013.
- [94] P. Masci, P. Curzon, M.D. Harrison, A. Ayoub, I. Lee, and H. Thimbleby. Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example. In *EICS 2013 - Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 81–90, 2013.
- [95] P. Masci, R. Rukšėnas, P. Oladimeji, A. Cauchi, A. Gimblett, Y. Li, P. Curzon, and H. Thimbleby. The benefits of formalising design guidelines: a case study on the predictability of drug infusion pumps. *Innovations in Systems and Software Engineering*, 11(2):73–93, 2015.
- [96] P. Masci, Y. Zhang, P. Jones, P. Curzon, and H. Thimbleby. Formal verification of medical device user interfaces using PVS. In *Conference of 17th International Conference on Fundamental Approaches to Software Engineering, FASE 2014 - Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*, volume 8411 LNCS, pages 200–214, Grenoble, 2014. Springer Verlag.
- [97] A. Mashkoo. Model-driven development of high-assurance active medical devices. *Software Quality Journal*, 24(3):571–596, 2016.
- [98] A. Mashkoo and M. Biro. Towards the Trustworthy Development of Active Medical Devices: A Hemodialysis Case Study. *IEEE Embedded Systems Letters*, 8(1):14–17, March 2016.

- [99] A. Mashkoo, M. Biro, M. Dolgos, and P. Timar. Refinement-based development of software-controlled safety-critical active medical devices. In *Software Quality. Software and Systems Quality in Distributed and Mobile Environments - 7th International Conference, SWQD 2015, Vienna, Austria, January 20-23, 2015, Proceedings*, volume 200 of *Lecture Notes in Business Information Processing*, pages 120–132, 2015.
- [100] Atif Mashkoo. The hemodialysis machine case study. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings*, pages 329–343, Cham, 2016. Springer International Publishing.
- [101] M.J. May, W. Shin, C.A. Gunter, and I. Lee. Securing the drop-box architecture for assisted living. In *Proceedings of the Fourth ACM Workshop on Formal Methods in Security Engineering, FMSE'06. A workshop held in conjunction with the 13th ACM Conference on Computer and Communications Security, CCS'06*, pages 1–12, 2006.
- [102] Dominique Méry and Neeraj Kumar Singh. Automatic code generation from event-B models. In *Proceedings of the Second Symposium on Information and Communication Technology*, page 179. ACM Press, 2011. 00054.
- [103] Huaikou Miao, Ling Liu, and Li Li. Formalizing UML models with Object-Z. In *Formal Methods and Software Engineering*, volume 2495 of *Lecture Notes in Computer Science*, pages 523–534. Springer Berlin Heidelberg, 2002.
- [104] D. Méry and N.K. Singh. Formal Development and Automatic Code Generation Cardiac Pacemaker. In *International Conference on Computers and Advanced Technology in Education*, Beijing, China, November 2011.
- [105] D. Méry and N.K. Singh. Medical protocol diagnosis using formal methods. In *International Symposium on Foundations of Health Informatics Engineering and Systems - FHIES 2011*, volume 7151 LNCS, pages 1–20, 2012.
- [106] D. Méry and N.K. Singh. Formal specification of medical systems by proof-based refinement. *Transactions on Embedded Computing Systems*, 12(1), 2013.
- [107] A. Murugesan, M.W. Whalen, S. Rayadurgam, and M.P.E. Heimdahl. Compositional verification of a medical device system. In *ACM SIGAda Ada Letters*, pages 51–64, 2013.
- [108] E. Neufeld and N. Kuster. Verification & Validation Benchmarks for Assessing and Demonstrating the Credibility of Computational Medical Device Evaluation. In *2015 9th European Conference on Antennas and Propagation, EuCAP 2015*. Institute of Electrical and Electronics Engineers Inc., 2015.
- [109] C. Poerschke, D.E. Lightfoot, and J.L. Nealon. A formal specification in B of a medical decision support system. In *ZB 2003: Formal Specification and Development in Z and B*, volume 2651, pages 497–512, 2003.



- 
- [110] Matteo Pradella, Matteo Rossi, and Dino Mandrioli. ArchiTRIO: A UML-compatible language for architectural description and its formal semantics. In *Proceedings of FORTE 2005*, pages 381–395. Springer Berlin Heidelberg, 2005.
- [111] Jugnoo S Rahi, Stuart Logan, Christine Timms, Isabelle Russell-Eggitt, and David Taylor. Risk, causes, and outcomes of visual impairment after loss of vision in the non-amblyopic eye: a population-based study. *The Lancet*, 360(9333):597–602, August 2002.
- [112] Elvinia Riccobene and Patrizia Scandurra. A formal framework for service modeling and prototyping. *Formal Asp. Comput.*, 26(6):1077–1113, 2014.
- [113] R. Rukšėnas, P. Curzon, A. Blandford, and J. Back. Combining human error verification and timing analysis: A case study on an infusion pump. *Formal Aspects of Computing*, 26(5):1033–1076, 2014.
- [114] Joachim Schmid. Compiling abstract state machines to c++. *Journal of Universal Computer Science*, 7(11):1068–1087, 2001.
- [115] S. Shuja, S.K. Srinivasan, S. Jabeen, and D. Nawarathna. A formal verification methodology for DDD mode pacemaker control programs. *Journal of Electrical and Computer Engineering*, 2015, 2015.
- [116] L.C. Silva, H.O. Almeida, A. Perkusich, and M. Perkusich. A model-based approach to support validation of medical cyber-physical systems. *Sensors (Switzerland)*, 15(11):27625–27670, 2015.
- [117] A. Simalatsar and G. De Micheli. Medical guidelines reconciling medical software and electronic devices: Imatinib case-study. In *Bioinformatics Bioengineering (BIBE)*, pages 19–24, Nov 2012.
- [118] N.K. Singh, M. Lawford, T.S. Maibaum, and A. Wassylng. Formalizing the Cardiac Pacemaker Resynchronization Therapy. In *HCI (17)*, volume 9185 of *Lecture Notes in Computer Science*, pages 374–386. Springer, 2015.
- [119] N.K. Singh, H. Wang, M. Lawford, T.S. Maibaum, and A. Wassylng. Step-wise formal modelling and reasoning of insulin infusion pump requirements. In Duffy V.G., editor, *Conference of 6th International Conference on Digital Human Modeling, DHM 2015 Held as Part of 17th International Conference on Human-Computer Interaction, HCI International 2015*, volume 9185, pages 387–398. Springer Verlag, 2015.
- [120] E.B. Sloane and R. Schrenker. Conceptual design and resources for a general-purpose safety and performance Verification and Validation Toolkit (V2T) for life-critical Wireless Medical Device Networks (WMDN). In *Annual International Conference of the Engineering in Medicine and Biology Society*, volume 7 VOLS, pages 178–181, 2005.
- [121] Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, January 2006.

## Bibliography

---

- [122] Á. Sobrinho, P. Cunha, L. D. Da Silva, A. Perkusich, T. Cordeiro, and J. Rêgo. A simulation approach to certify electrocardiography devices. In *International Conference on E-health Networking, Application Services (HealthCom)*, pages 86–90, Oct 2015.
- [123] Á. Sobrinho, P. Cunha, L.D. Da Silva, A. Perkusich, T. Cordeiro, and J. Rego. A methodology for modeling and simulation of biomedical signal acquisition devices. In *International Conference on E-health Networking, Application & Services*, pages 227–231, 2015.
- [124] Maria Spichkova. Design of formal languages and interfaces: “formal” does not mean “unreadable”. *Emerging Research and Trends in Interactivity and the Human-Computer Interface*, pages 301–314, 2014.
- [125] M. Sun and J. Meseguer. Distributed Real-Time Emulation of Formally-Defined Patterns for Safe Medical Device Control. In *RTRTS*, volume 36 of *EPTCS*, pages 158–177, 2010.
- [126] H.M. Tahir, M. Nadeem, and N.A. Zafar. Specifying electronic health system with vienna development method specification language. In *2015 National Software Engineering Conference, NSEC 2015*, pages 61–66. Institute of Electrical and Electronics Engineers Inc., 2015.
- [127] A. Ten Teije, M. Marcos, M. Balser, J. Van Croonenborg, C. Duelli, F. Van Harmelen, P. Lucas, S. Miksch, W. Reif, K. Rosenbrand, and A. Seyfang. Improving medical protocols by formal methods. *Artificial Intelligence in Medicine*, 36(3):193–209, 2006.
- [128] U.S. Food and Drug Administration (FDA). General Principles of Software Validation; Final Guidance for Industry and FDA Staff, January 2002.
- [129] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2006.
- [130] Andrea Vitali, Giancarlo Facoetti, and Angelo Gargantini. An environment for contrast-based treatment of amblyopia using 3D technology. In *International Conference on Virtual Rehabilitation 2013 - August 26-29, 2013 in Philadelphia, PA, U.S.A.*, 2013.
- [131] Ann L. Webber and Joanne Wood. Amblyopia: prevalence, natural history, functional effects and treatment. *Clinical and Experimental Optometry*, 88(6):365–375, November 2005.
- [132] Yi Zhang, Raoul Jetley, Paul L Jones, and Arnab Ray. Generic Safety Requirements for Developing Safe Insulin Pump Software. *Journal of Diabetes Science and Technology*, 5(6):1403–1419, November 2011. 00008.