Universidade do Minho
Escola de Engenharia

Daniel Nascimento Cadete

From Natural Language Requirements to Formal Descriptions in Alloy through Boilerplates

Janeiro de 2012

Universidade do Minho
Escola de Engenharia

Daniel Nascimento Cadete

From Natural Language Requirements to Formal Descriptions in Alloy through Boilerplates

Dissertação de Mestrado em Engenharia de Informática

Trabalho efectuado sob a orientação do
Professor Doutor José Nuno Fonseca Oliveira

Janeiro de 2012

University of Minho
School of Engineering

From Natural Language Requirements to Formal Descriptions in Alloy through Boilerplates

Daniel Nascimento Cadete
Informatics Department - University of Minho

January 31, 2012

Acknowledgements

I would like to thank my supervisor, José Nuno Oliveira, for all the help, guidance and inspiration, for all the time spending discussing ideas for this dissertation and for allowing me to see software development in a different perspective. Working with him was an inspiring task and something that I will never forget. I strongly believe that his vision of software and mathematics together, can shape the future of software development.

To my colleagues from Braga, João Martins, André Pedro, Manuel Costa, Nelson Gonçalves, João Bordalo, Raquel Ribeiro, and Francisco Ribeiro, I would like to thank for all the relaxing moments and for all the chats which greatly inspired this dissertation.

To my friends João Vieira, Nuno Santos, Bruno Almeida, Ana Sequeira, Samuel Martins, Cátia Conceição, João Palma, Cristel Domingos, Cátia Esteves I would like to thank for all the support, hours of laughing and the countless times you have made me move forward to complete this dissertation.

To Critical Software and particularly José Miguel Faria I would like to thank for ideas, background and support which significantly improved this dissertation. Thanks are also due to the Educed team and particularly André Passos, for their support and interest on the results of this dissertation. Seeing one's research absorbed by industry makes the work extremely fulfilling. To EVOLVE/QREN project I would like to thank for the financial support which partially funded this research through a research grant.

Many thanks to all my family who put expectations on me and always helped me move forward. Specially to my beautiful niece Matilde, whose smile and those big hugs make me want to be near her all the time.

To my father, mother and sister, I dedicate this dissertation. During all my life, you have been my inspiration and my idols, you have taught me how to overcome challenges and work hard to get the things I want in life.

d

Resumo

Os métodos formais são normalmente aplicados por especialistas nas fases finais do desenvolvimento de *software*. A sua aplicação visa identificar erros de programação, reduzindo assim a probabilidade de uma falha futura. Tipicamente, os erros que se encontram prendem-se com má interpretação de requisitos e não má programação. Cada vez mais os documentos de requisitos tratam de termos complexos e fora do conhecimento do programador, o que leva a mais erros de interpretação e consequentemente a um aumento dos custos de execução de um projeto de *software*. A utilização de métodos formais poderia minimizar estes custos, caso eles fossem utilizados não para verificar código, mas sim para verificar requisitos. No entanto, muitas empresas evitam a utilização de métodos formais, devido ao custo elevado da sua aplicação. Os programadores ou engenheiros de requisitos não conseguem aplicar métodos formais de forma eficiente sem terem formação prévia e específica na área, o que implica a contratação de especialistas em métodos formais.

Nesta dissertação são apresentados métodos que visam aproximar os métodos formais da escrita dos requisitos. Para tal, a modelação formal é utilizada não para verificar código, mas para verificar a escrita de requisitos. Inicialmente é apresentado um *standard* para a criação de modelos, que faz uma correspondência direta entre cada requisito e o seu modelo formal. Este *standard* é suportado por uma ferramenta que, entre outras coisas, gera de forma automática representações gráficas dos requisitos através dos seus modelos. Posteriormente é apresentada uma conexão entre templates de requisitos (*requirements boilerplates*) e modelos Alloy. Esta conexão permite a criação de modelos formais de forma automática, sem necessidade de um especialista. Isto reduz drasticamente o custo de utilização de métodos formais. Apresenta-se igualmente o começo de uma álgebra que permite agregar estes templates. Esta agregação permite que um engenheiro de requisitos escreva o seu documento de requisitos através de templates e no fim tenha de forma automática o modelo formal de todos os requisitos.

Quando se está a modelar um documento de requisitos em Alloy e a certo ponto aparecem requisitos com restrições temporais explícitas, é necessário recriar todo o modelo numa ferramenta que permita essa modelação (ex: Uppaal). Este processo está sujeito a erros, porque esta transformação é manual e altamente dependente da interpretação de quem está a modelar. Nesta dissertação é apresentado um método que permite a geração automática de um modelo Uppaal a partir de um modelo Alloy. Esta transformação permite que a qualquer ponto da modelação em Alloy, se crie o modelo Uppaal correspondente e se especifiquem as propriedades temporais.

f

f

Abstract

Formal Methods are usually applied by specialists in the final phases of software development. They aim to identify programming errors, and through that reduce the probability of a future failure. Usually, errors are more related with misinterpretation of requirements than with bad programming. More than ever, requirements documents deal with complex terms, which programmers aren't familiar with, resulting in an increase of misinterpretation of requirements and increasing the costs of the execution of a software project. The use of formal methods could reduce these costs, if properly used to verify requirements and not source code. However, most companies avoid using formal methods due to high costs associated with formal methods application. Programmers or requirements engineers can't apply formal methods efficiently without previously having specific training, which implies hiring expensive specialists in formal methods.

This dissertation presents methods which aim to bring formal methods closer to requirements descriptions. For such, formal modeling is used to verify and validate the descriptions of requirements, and not source code. Initially it's presented a standard to create formal models, which makes a direct correspondence between each requirement and its model. This standard is supported by a tool which, among other things, automatically generates graphics representations of requirements using its models. Afterwards it's presented a connection between requirements boilerplates and Alloy models. This connection allows to generate formal models in an automatic fashion, without the need of a specialist. This drastically reduces the costs of using formal methods in software projects. It's also presented the beginning of an algebra which allows to aggregate these templates. This aggregation allows one to write its requirements documents throught boilerplates and at the end have the complete model of all requirements, for free.

When one is modeling a requirements document in Alloy and at some point appears requirements with explicit temporal restrictions, it's necessary to recreate the whole model in a tool which allows that kind of specification (eg. Uppaal). This process is highly error prone, because it's a manual transformation and highly dependent on the interpretation of who is modeling. In this dissertation it's presented a method which allows to automatically generate an Uppaal model from an Alloy model. This transformation allows that at any point in the requirements document, the requirements engineer can generate the correspondent Uppaal model and there specify the temporal properties.

Contents

Acronyms

API Application Programming Interface.

CC Common Criteria.

CMS Configuration Management System.

DFA Deterministic Finite Automaton.

FMTR Formal Methods Tool Repository.

GUI Graphical User Interface.

IFIP International Federation for Information Processing.

LCS Life-Critical System.

LHS Left Hand Side.

NATO North Atlantic Treaty Organization.

NL Natural Language.

OOP Object Oriented Programming.

PIFP Partition Information Flow Policy.

RB Requirements Boilerplates.

RE Requirements Engineering.

RHS Right Hand Side.

SAT Boolean satisfiability problem.

SPK Secure Partitioned Kernel.

UML Unified Modeling Language.

VSR Verified Software Repository.

XML Extensible Markup Language.

List of Figures

List of Tables

Chapter 1

Introduction

Software has gained a prominent place in mankind history. It is today an essential part of every service, business process or research activity. Software usage has proliferated and is omnipresent not only in companies and universities but also in the every day life of the anonymous citizen: in cars, mobile phones, bank ATM systems, home computers, TV sets, houses, and so on.

Since the first computer program by Ada Lovelace to the highly complex software systems of today which control nuclear power-stations, maintain airplanes flying or coordinate complex financial operations across the globe, the process of software construction has undergone an impressive evolution. However, under such rapid growth in both complexity and demand for software, programmers have become unable to deliver 100% safe software systems, meaning that quality is hard to achieve both in the development process and in the end product.

Such an embarrassing situation dates back to the 1960s, when the world first witnessed what became known as the *Software Crisis*. This was the first evidence that the archaic development methods of those times were inefficient and resulted in poor quality, highly error prone software often offering more costs than benefits. Under the urgent need to change this situation, the North Atlantic Treaty Organization (NATO) organized a Software Engineering Conference in 1968 at Garmisch, Germany where academics from several universities, staff from software companies and other contributors from the civil sector addressed issues such that the design, production, implementation, distribution and service of software [65]. In this conference the phrase “Software Engineering” was coined to reflect the need for software manufacture being based on solid foundations:

> *In late 1967 the Study Group recommended the holding of a working conference on Software Engineering. The phrase ‘software engineering’ was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.* (Quoted from [65].)

Provocative or not, the need for sound theoretical foundations has clearly been under concern since the very beginning of the discipline.

The *NATO Conference* of 1968 triggered the development of a series of methods and guidelines which significantly improved software quality. Business could now rely on machine support without fear that at any moment the software would start to fail with consequent massive losses. Software development was structured in a sequence of formal phases, usually starting with informal meetings where clients exposed their needs and requirement engineers tried to gather a sketch of what the system should be able to do and in which conditions. The outcome of such meetings would be grouped and refined until all requirements were written in Natural Language (NL) and put together into a so-called requirements document. These requirements documents could then be passed to the development team who would be charged thereupon with developing a system that should meet the requirements written in the document.

In this way, requirements engineering emerged as one of the most important areas in the software industry. After the NATO conference further investigation was carried out by an ever larger number of researchers, industry and universities. In 1995, the International Federation for Information Processing (IFIP) Working Group 2.9 was established, aiming at providing insights on requirements specification, interpretation and documentation. This group encompasses a number of different areas in requirement engineering: formal representation and requirements modeling, requirement elicitation and further analysis, tools and environments to support requirement engineering, requirements for safety-critical, real-time and embedded systems, etc.

The area of IFIP 2.9 which studies requirements on safety-critical and real-time systems is intrinsically related to the scope of this dissertation. Many software systems of today are systems whose failure does not result in life or money loss: one may have to restart the computer or lose a day of hard work, but hopefully one will live another day to work out the problem. However, there are software systems which *cannot fail in any circumstance*. A system of this kind is usually called a Life-Critical System (LCS). LCS failure or malfunction can result in death or injury to people, environmental harm or loss of great amounts of money [7].

Preventing LCSs from failure or malfunction has became extremely important and the history of 20th and 21th century is full of catastrophes caused by LCS malfunction. On 4 June 1996 the Ariane rocket crashed 37 seconds after launch. According to [57] the cause for its crash was an integer overflow arising from bad software design practice. The Denver International Airport main advance would be its top technology in an automated baggage handler, but software problems delayed the airport opening by 16 months [66]. All these events (and many others found in literature [32, 60, 12]) have resulted in millions of dollars spent to resolve failures and sometimes in human life losses.

As the software industry started to face new challenges, universities and software companies started to invest in new approaches to maintain the strict requirements of LCS. Some invested in methods in the area of software engineering, trying to improve existing methods. Others discovered new methods ensuring that systems wouldn't fail or that the probability of failure is residual [64, 52, 11]. Other invested in mathematical theories to prove or increase the confidence that software system won't fail based on sound mathematical proofs. This is what the scientific community recognizes as *Formal methods* [14].

1.1 Requirements Engineering

Ian Sommerville presents a widely accepted definition for software engineering in [78]:

> *Software Engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification to maintaining the system after it has gone into use.*

Software engineering thus appears across all phases of software development. Software engineers try and improve this discipline every day so as to discover new methods of analysis. In order to improve software quality one needs to know first and foremost why systems fail or don't do what they are supposed to do. The Standish Group present in [79] a number of factors for software failure, given in Table 1.1.

Table 1.1: *Why software projects fail*

Factors	%
Incomplete Requirements	13.1
Lack of User Involvement	12.4
Lack of Resources	10.6
Unrealistic Expectations	9.9
Lack of Executive Support	9.3
Changing Requirements & Specifications	8.7
Lack of Planning	8.1
Didn't Need it Any Longer	7.5
Lack of IT Management	6.2
Technology Illiteracy	4.3
Other	9.9

Inspection of this table shows that poor use of technologies is not a main factor for software project failure, but rather incomplete requirements, which is the factor standing at the top. So software engineering had to find "new engineering" inside itself: requirements engineering. Laplant defines requirements engineering in [58] as:

> *Requirements engineering is the process of eliciting, documenting, analyzing, validating, and managing requirements.*

The term requirement usually means a service the system should provide. Clearly, by improving the quality of requirements one not only improves the quality of target software system but also does so at a much lower cost when compared to improving the other phases in development and maintenance, where (bad) design decisions have been committed into the system already.

According to [78], requirements can be divided into:

- *User Requirements* - Statements in NL telling what services the system is expected to provide and under which circumstances.
- *System Requirements* - System's functions, services and operational constraints in detail.

Both system and user requirements shouldn't be directly delivered to the development team without passing through a series of stages which aim to improve the quality and understanding of requirements. First of all, they should be elicited through a number of different techniques such as group elicitation, prototyping, model-driven techniques, etc. This *elicitation phase* [1] aims to find out which main problems need to be solved, how the system will fit with the existing organization and the stakeholders [33] and so on.

After the elicitation phase, requirements should be modeled and analyzed. Typically in this phase, the requirements are represented in some abstract form. One finds in [67] find different types of modeling that one submit requirements to:

[1] The word "elicitation", which comes from the Latin *"elicit" ('draw out by trickery or magic')* means drawing forth something that is latent or potential into existence.

- *Enterprise Modeling* - Organization structure and how the system will interact with it.
- *Data Modeling* - Analyze how the system data will be manipulated and kept.
- *Behavioral Modeling* - Interactions of the stakeholders with the system.
- *Domain Modeling* - How the system will interact with the world around it.
- *Non-Functional Requirements Modeling* - Quality objectives that the system should meet.

It is important to define from the early stages of the project who will use and read requirements. In most projects, requirements are used by several people:

- *User* - Someone who uses and tests the final system.
- *Systems Engineer* - Someone who focuses on the design and management of the project life cycle.
- *System Designer* - Someone who designs the architecture, components, modules, interfaces of the software.
- *Programmer* - Someone who writes the software.
- *Tester* - Someone who writes and performs software tests.

Requirements are usually written in NL and they hardly can be written in other format due to the fact that all people mentioned above, and participates in the software development should be able to easily understand any requirement [1]. Because NL is ambiguous and imprecise, requirements engineers have tried to find ways to improve the quality of requirements without writing them in other (eg. formal) way. A major advance in achieving higher quality requirement textual descriptions is to consistently write each requirement following a Requirements Boilerplates (RB).

Requirements boilerplates are textual templates of the form:

The <Stakeholder> shall be able to <Capability>

The idea is to cast arbitrary requirements into such templates, as a means to ensure that one is not writing possibly ill-formed or difficult to understand free text, but rather a piece of text which as been used by others and proved effective and implementable before. Faced with a requirement such as "The Operating System can schedule processes“, one can write the requirement by instantiating the above boilerplate: *The **Operating System** shall be able to **Schedule Processes***.

Requirements boilerplates play a major role in the approach to requirements engineering put forward in this dissertation. A detailed account of requirements boilerplates will be given in Chapter 5.

Another important concept in requirements engineering is traceability. Traceability is described in [34] as

> *"Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)."*

Requirement traceability is important because requirements change all over the development cycle of a software project. Keeping track of those changes helps bringing up requirement quality and justifies why clients want particular requirements. In design process, traceability helps in justifying design options, with a complete history of the design process and its changes. It is essential for testers, because test cases are designed to test specific sets of requirements. Whenever a requirement changes, testers know exactly which changes need to be made in their test suites in order to cover the changed requirements [74].

Requirements traceability is usually achieved using frameworks. Researchers have put together several frameworks which aim to improve traceability using both manual an automatic tasks. However, there is today an increasing need for automatic frameworks catering for automatic requirement traceability [15]. Chapter 3 will present a method and a tool which helps in requirement traceability using formal models and a standard layout for writing and analyzing requirements.

1.2 Formal Methods

Above it was mentioned that, after the elicitation phase, requirements should somehow be modeled and analyzed. Formal Methods are techniques that use mathematics in the modeling, making it possible not only to specify but also to verify the design, be this hardware or software. A method is formal if it has precise mathematical foundations that enable the definition of properties like consistency, completeness, specification, implementation and correctness [81]. Formal methods usage in software systems can be divided into *specification* and *verification*.

Specification is the process whereby a system and its desired properties are formally described. Usually, the important properties of a system are the behavioral ones. To specify a system one needs to write such properties in a precise manner, allowing one to gain a deeper understanding of the system and to discover design flaws, inconsistencies, ambiguities and incompleteness. The specification also provides a useful communication between the client, designer, programmer and tester [35].

There are several methods such as Z [82], VDM [49] and Larch [35] which cater for the formal specification of sequential systems. In such methods the system (machine) states are mathematical entities like sets, relations and functions; state transitions are specified by pre and post conditions. Other methods, such as CCS [63] and Temporal Logic [61], enable the specification of concurrent systems where states are defined over domains and the behavior is represented through sequences, trees, traces or events.

Verification is the process of ensuring that a system satisfies the properties described in its specification. Formal Verification used maths for this, the verification consisting in *proving* that all properties are met. In a sense, every property rises a *theorem* — the assertion of its preservation in the implementation — which is discharged by proof. As already mentioned, this is common practice in safety-critical systems, which include protocols, cryptographic algorithms, software programs and kernel hardware devices [81]. This class of verification techniques is known as *Theorem Proving*.

Verification can still be approached by *incomplete* techniques such as *Model Checking*. Model checking is the process of building a finite model of the system and checking if a desired property holds in that model. Because such a model is a finite approximation of the final system one can never be sure that a property is valid; but can show that the property is invalid by means of counter-examples. Model checking has been widely used to verify hardware specifications and is now being used to verify software [3].

In theorem proving both the system and its properties are expressed as formulæ in some formal system with axioms and inference rules. In order to verify a system, one has to discharge the proof of a property using the axioms of the formal system. Although theorem proving can be carried out manually, it is usually done using a computer assisted program (*Theorem Prover*) helping discharge the proof. This prover can be automatic [25] or semi-automatic [4], where the user guides the program in the proof process.

Model Checking. Later in this dissertation, model checking will be shown to be at the epicenter of the approach put forward for requirement engineering in this dissertation. It therefore deserves a more detailed account. The technique invented separately by E.M. Clarke, E.A. Emerson and J. Sifakis who earned the 2007 Turing Award. The great disadvantage of model checking is the state space explosion [2] and the impossibility to specify and verify systems with infinite states. There are two approaches to the model checking problem, the *Explicit State Model Checking* and *Symbolic Model Checking* [76].

In *Explicit State Model Checking* the system is modeled as a finite automaton and the properties are expressed in a temporal logic. An efficient algorithm is used to determine if the property is true in the automaton. With this approach, the automaton usually consists of several spaces and the verification of properties in useful time is highly related to the user's ability to model a system with fewer state variables [62]. *Symbolic Model Checking* tries to overcome this weakness by using Boolean formulæ to represent sets and relations that are manipulated using so-called *ordered binary decision diagrams* (OBDD) [62]. With this approach, systems with up to 10^{20} system can be verified, resulting in a wider range of verifiable systems.

1.3 Formal Methods and Requirements Engineering

Formal Methods (F.M) have been used in several phases of the software development with different objectives and results [50]. In the literature one finds research aiming at using F.M in the early phases of software development, namely in areas related with requirements engineering.

The approaches followed by [72, 38, 75] use formal methods to model the requirement specifications and find some ambiguities in them. Although these approaches are valuable and interesting, little is done in order to clear up the textual descriptions of the requirements. Their main focus is on identifying ambiguous requirements and fixing them before implementation. For a requirement engineer it is not enough to identify all requirements which are bad specified, for she/he also needs to have some kind of tips or guidelines on how to re-write them, making them more clear and suitable to be implemented.

Elsewhere researchers have tried to focus on the descriptions of requirements and methods which could improve them. References [9] and [83] put forward two approaches which try and identify bad descriptions of requirements. With methods like these, requirements engineers already have the possibility to identify and re-write requirements using a set of guidelines. The main disadvantage of these approaches is their lack of mathematical formalism which doesn't shorten the gap between requirements descriptions and implementation.

[2] State space explosion usually means that the number of states needed to represent the system doesn't fit into computer memory.

Requirements engineers would much benefit from a mathematical formalism able to provide guidelines, methods or techniques that could help in rewriting the requirement's descriptions that are ambiguous. Through mathematics, the distance from description to implementation of requirements would be shorter, which would be helpful in projects where requirements are constantly changing.

1.4 Aims of the dissertation

The software industry must use tools and techniques to prevent disasters caused by LCS failures. Formal Methods appeared in the late 1980s as a good solution to detect and prevent software bugs [14] in such systems. Although these methods have shown significant improvement on LCS, they are often criticized because of its difficult adoption and the high costs they purport by requiring highly specialized software engineers. This happens mainly because they are applied when the system is already developed or in an advanced state of development. As Daniel Jackson puts it [42]:

> *Almost all grave software problems can be traced to conceptual mistakes made before programming started.*

Further to applying formal methods to code analysis and synthesis, these methods should be applied to requirements. Find a serious flaw at requirement phase in an expensive project in eg. the aerospace industry, well before any development has started, much can be saved in code refactoring and bug correction. It is widely accepted that one of the main reasons for such mistakes is the use of poorly written NL requirements descriptions [1, 23]. Following this idea, researchers started to develop techniques and tools that help software engineers to detect and correct flaws in the early phases of software development [73, 48].

This dissertation aims to improve the use of Formal Methods in requirements documents with the objective to identify mistakes in such an early phase — when actually *writing them* (earlier than this is not possible!). F.M will be applied in order to create a mathematical foundation for requirements documents which otherwise would not exist. This mathematical foundation may be hidden from readers of requirements documents but, if used, will provide valuable contribution to the quality of the final system through the use of mathematical meaningful boilerplates.

Through F.M it is possible to achieve not only a mathematical model which is meaningful in further phases of development but also an insight on the quality of textual descriptions of requirements. Requirements description are bound to be written in NL because this is the language easily understood by everyone (hopefully). But it suffers from major disadvantages: it is imprecise, ambiguous and could lead to different interpretations in the same design. A combined use of Requirement Boilerplates (RB) with the Alloy model checker will be eventually proposed in order to add to structure and clearance of such textual descriptions.

1.5 Document Structure

The following Chapter will present the tools used throughout the dissertation and explain why they where chosen. Chapter 3 presents a method which helps requirements engineers to write better requirements documents while creating formal models for them. This method is supported by a tool developed on purpose, which is also explained in the chapter. The method is illustrated using a small example which evolves into a larger *case study* in Chapter 4, concerning a .Secure Partitioned Kernel (SPK)

The ideas presented in these chapters are refined into Chapter 5, where a connection between requirement patterns and formal models is presented. This chapter presents an interesting connection between Alloy and commonly used patterns to write requirements. In Chapter 6, Alloy is integrated into Upppal, creating an automated method which connects the two worlds (relational calculus and first order logic to networks of timed automata). Finally, Chapter 7 concludes and presents suggestions for future work, showing new possibilities while stressing on the industrial adoption of the ideas put forward in earlier chapters.

Chapter 2

Formal Methods Tools

The formal methods community offers today several tools which support different notation styles and purposes. In the *Formal Methods Tool Repository (FMTR)* [1] one can see a categorized account of formal methods tools by their different features. Which such features should a tool offer in case one wishes to adopt it to support an efficient and fast method to analyze requirements descriptions and quickly provided insights on inconsistencies and ambiguities? Such a tool should be able to identify flaws (eg. by providing counterexamples), should be versatile (making it easy to re-factor specifications) and offer a good visual representation of models and use cases. The last point is important because it will allow the user to present results to non-technical staff, that is, engineers with no deep knowledge in F.M.

Looking at the FMTR table one finds such criteria in columns *"Model Checking"*,*"Animation / Execution"*, *"Graphical User Interface (GUI)*˜ and *"Refinement"*. In the following section we will present Alloy, a tool which meets most of such criteria. This tool will play an important role in the method presented in Chapter 3, allowing for a quick way to analyze requirement descriptions. In Section 2.2 we will introduce Uppaal, another tool offering excellent support to model real-time constraint systems. Chapter 6 will show how to transform Alloy models into Uppaal and then apply real-time constraints in an efficient fashion.

2.1 Alloy

Alloy was created by a team in the Software Design Group at MIT lead by Daniel Jackson. The main aim was bringing the benefits of model checking and the power of abstraction to the software development world. |Alloy is a language (widely inspired by Z and object modeling notions) together with an analyzer which performs automatic verification of model properties. Alloy's mathematical foundation is set theory (sets, relations, etc) allowing one to easily model structures like file systems, naming schemes, architectures, etc. With sets and relations, primitive data types like records or arrays can easily be expressed and analyzed efficiently [31].

Alloy's lemma is *In Alloy everything is a relation*. The tool which comes with the language is called Alloy Analyzer and it is atypical as a model checker. What it does is the following: given a specification of the some system or problem, it transforms it into a Boolean formula and then sends this to an off-the-shelf Boolean satisfiability problem (SAT) solver [2] which tries to find some

[1] See `http://fmtoolsrepository.di.uminho.pt` .

[2] SAT is the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to true.

model that makes the formula true.[44]

Let us illustrate the Alloy approach to software modeling through the following example: suppose one wishes to model a *mini file system* with some basic file management operations. The fictional requirements call for an operation which allows the user to navigate through the file system and two operations for creating and removing directories. The following is also required: the root directory doesn't have a parent directory, files and directories don't have names and one can remove any empty directory as long as the directory isn't the root.

Below we will go through all the steps in using the Alloy language and analyzer. The reader will see how easy it is to model a system and how the Alloy approach is a lightweight one, proving effective in a wide range of problems in the software development world.

Signatures. A signature in Alloy represents a set of atoms. A signature can be interpreted as a class (in Object Oriented Programming (OOP) terms) and (like a class) it can declare relations associated to its atoms and the creation of implicit types. In our example three signatures are readily identified:

```
sig File{}
sig Dir{}
sig FileSystem{}
```

These offer files and directories in the file system and the atoms of the signature *FileSystem* will represent the file system at different points in time. A real file system has a *parent* relation among files and directories. Knowing that both directories and files have parent directories suggests that both *File* and *Dir* be subsets of an abstract signature called (file-system) *Object*:

```
abstract sig Object {}
sig File, Dir extends Object {}
```

Note the two keywords in creating signatures, *extends* and *abstract*. Where declaring a signature as *extends* of a top-level signature, this means that it will be a subset of the top-level signature (in the above: *File* and *Dir* are subsets of *Object*). The sets created by signatures *File* and *Dir* are completely disjoint. By declaring a signature as *abstract*, it will have no further atoms except those belonging to its extensions.

The *FileSystem* signature as given above doesn't tell much about what a file system is. If we interpret each atom of *FileSystem* as a "picture“ of the file system in each moment in time, such atoms should have a root, a parent relation, a print working directory (pwd) and a set of files and directories:

```
sig FileSystem {
  root : one Dir,
  pwd : one Dir,
  objects : set Object,
  parent : Object → lone Dir
}
```

This already shows some interesting aspects of Alloy language. The new *FileSystem* signature now has three new relations between atoms from *FileSystem* and atoms from *Dir* and *File*. The relation *root* declares that each atom of *FileSystem* has only one root and it is a member of *Dir*. The same happens for relation *pwd*. The relation *objects* declares that an atom of *FileSystem* has a set of files and directories. The parent relation is a little bit richer than the previous one, as it is a ternary relation between a *FileSystem*, a *Object* and a *Dir* and introduces the new word *lone*. This reserved is a multiplicity operator which represents zero or exactly one atom. So one can read the *parent* relation as follows: a particular file system has objects which may have directories as parents. Recall that the root directory has no parent (as required), entailing that the signature doesn't force all objects to have a parent directory. Later on we will show how correctly express this assumption in the model through the declaration of an *invariant*.

Multiplicities, Operators and Constants. Before moving forward in the *mini file system* example attention should be paid to the operators, multiplicities and constants available in Alloy. Alloy provides a rich set of operators allowing for easy manipulation of both sets and relations to express properties in several ways. In general, when wishing to declare a relation r from a set A to a set B one writes r as:

```
r : A m -> n B
```

This will force relation r to map each member of set A into n members of set B, and map m members of set A into each member of set B. Instances of "*m*" and "*n*" are called *set multiplicities*. In Alloy multiplicities can be as in the following table:

Multiplicity Word	**Meaning**
`some`	one or more
`one`	exactly one
`lone`	zero or one
`set`	any number

Multiplicities *some*, *one* and *lone* are also used when wishing to quantify some variable (examples of this will be given later concerning the *mini file system* example).

Concerning Alloy's operators, these can be separated into two categories: the set operators and the relation operators. Set operators are, as expected: **+** (union), **-** (difference), **in** (inclusion) and **=** (equality). These operators can also be applied to two relations as long they have the same *arity* [3].

What really makes the difference is the rich (yet small) set of relational operators which allow one to manipulate, combine and construct all kind of relations. The most relevant operators are: **->** (product), **.** (join, composition),**~** (transpose) and **^** (transitive closure or closure). For a detailed explanation of each operator available in Alloy it is highly recommended to see Daniel Jackson's book [43]. [4] Alloy offers not only a rich set of operators and multiplicities but also three important and time saving constants:

[3] The *arity* of a relation, is the number of sets that it relates. A relation with arity *two* relates objects from two domains, and is often called a *binary* relation. A relation with arity *three* relates objects from three domains and is usually called a *ternary* relation. Relations can have an arbitrary arity.

[4] The operator of closure it is a little more complicate than others. A relation r it is said transitive if: when it contains the tuples (a, b) and (b, c) then also contains (a, c). The transitive closure ^r of a binary relation r it is the smallest relation which contains r and is transitive.

Constant	**Meaning**
`univ`	universal set
`none`	empty set
`iden`	identity

The best way to visualize what these constants mean in Alloy is through a small example. Suppose we have a model with the following sets:

```
File  =  {(F1),(F2),(F3)}
Dir   =  {(D1),(D2)}
```

For these Alloy computes the following sets:

```
univ  =  {(F1),(F2),(F3),(D1),(D2)}
none  =  {}
iden  =  {(F1,F1),(F2,F2),(F3,F3),(D1,D1),(D2,D2)}
```

These constants can be used together with the operators presented above to construct elegant relational properties. Suppose we have a relation $Dir \xrightarrow{R} File$:

```
R  =  {(D1,F1),(D2,F3)}
```

By resorting to *composition* and *transposition*, one can specify that ”R is injective“ as `R.~ R in iden`. We have:

```
~R    =  {(F1,D1),(F3,D2)}
R.~R  =  {(D1,F1),(D2,F3)}.{(F1,D1),(F3,D2)}
R.~R  =  {(D1,D1),(D2,D2)}
```

Clearly, `R.~ R in iden` is true for relation `R`. Constructs of this kind are useful in writing properties which are easily understandable and memorable, which is important when we need to understand models with several lines and rich properties.

Constraints. The syntax for signatures allows the definition of structural restrictions on their inhabitants. However, this kind of restriction is not enough to create realistic models. Most of the time we need to impose restrictions in the model's overall behavior, which isn't easy to do through signatures. Therefore, besides signatures Alloy offers the definition of predicates, functions, facts and assertions which altogether create new ways of defining restrictions.

Restrictions which one wants to hold universally at any time can be written as *facts*. A fact is written using the keyword `fact` and the expressions defined inside it are always true for every instance of the model. We can have an arbitrary number of facts in the same model and Alloy interprets them all in the same way. When a model has facts, Alloy only shows instances of the model which entirely verify all facts defined.

In the *mini file system* example we could define as fact the assumption that the root never has a father:

```
fact RootOrphan{
  all f:FileSystem | no (f·root)·f·parent
}
```

This fact begins by quantifying a variable `f` with keyword ”all“ meaning that the expression on the right hand side (after the vertical bar “|”) will hold for every atom in set `FileSystem`. Expression `no (f.root).f.parent` calls for additional explanation. Where writing `f.root` one refers to the root atom of the `f` file system. Then one states that there can be no directory related with such root by relation parent. Altogether, fact `RootOrphan` says that no file system exists whose root has a parent. Later we will see why sometimes it is better to express properties of this kind using predicates rather than facts.

When wishing to analyze the behavior of a model with different restrictions for different situations in the model, one has to use Alloy predicates. A predicate (defined using the keyword `pred`) is a restriction with a name and zero or more arguments. The predicate must be always true to the arguments passed in. Predicates are very useful to analyze and model the behavior of the system's operations.

In the *mini file system* example there should be operations for changing directory (cd), creating a new directory (mkdir) and removing an empty directory (rmdir). To define a predicate modeling the *cd* operation, it should have as a parameter the new directory and two instances of *FileSystem*, one staying for the state before the operation and the other for the one after the operation — respectively `f` and `f'` in:

```
pred cd[f,f' : FileSystem, d : Dir] {
  d in f·objects
  f'·pwd = d
  f'·root = f·root
  f'·objects = f·objects
  f'·parent = f·parent
}
```

Clearly, the directory which one will change to must exist as a file system object before the operation takes place (restrictions of this kind are known as *pre-conditions*). The *post-condition* will maintain all properties of the file system except *pwd*, which will change to the new directory.

The predicate which models the removal of a directory has more elaborate pre- and post-conditions:

```
pred rmdir[f,f' : FileSystem, d : Dir] {
  d in f·objects
  d ≠ f·root
  d ≠ f·pwd
  no (f·parent)·d
  f'·objects = f·objects - d
  f'·parent = f·parent - (d → Dir)
  f'·pwd = f·pwd
  f'·root = f·root
}
```

Again interpreting `f,f'` as the before and after states of the operation, respectively, we see that now we have more clauses in the *pre-condition* which need to be verified for `f` in order to ensure the success of the operation. First, the directory to be removed must exist in the system before the operation (`d in f.objects`). By convention, the root cannot be removed (so `d != f.root`). Another clause (`d != f.pwd`) prevents from removing the working directory. To express the property ”the directory must be empty“ one writes `no (f.parent).d` and this property concludes the set of clauses of the *pre-condition*.

The operation must ensure two things in the new file system: the argument directory must disappear from the objects relation (`f.objects - d`) and the parent relation cannot relate that directory with any other object. The definition of parent relation was given as `parent : Object -> lone Dir`. As seen above, the operator $- >$ represents the product, ie the parent relation relates a file system with pairs `(Object,Dir)`. Where writing `f'.parent = f.parent - (d -> Dir)` one is forcing that the new parent relation will have all pairs in the old one except the ones in which `d` appears as the first element of the pair.

To complete the illustration, we give below the dual operation which creates an empty directory:

```
pred mkdir[f,f':FileSystem, d:Dir]{
  d not in f·objects
  f'·objects = f·objects + d
  f'·parent = f·parent + (d → f·pwd)
  f'·pwd = f·pwd
  f'·root = f·root
}
```

The details of this operation are similar to the ones present in *rmkdir*. Instead of removing a directory, we add it to the *objects* and *parent* relation, maintaining all that is static (*pwd* and *root*).

Assertions. *Assertions* are Boolean expressions which express desirable properties of the model which one wishes to check using the Alloy analyzer. Invalid assertions mean faults in the design, unless there is some misspecification in the assertion. Assertions are automatically verified by the Alloy analyzer, for the given scopes. In the *mini file system* example one could write an assertion saying that all objects in the file system are accessible from the root directory:

```
assert RootReachable{
  all f:FileSystem | all o:f·objects | f·root in o·^(f·parent)
}
```

To correctly write this property one has to use the transitive closure operator. The expression `^o.parent` gives us all directories above the object `o` in the parent relation (”above“ in the sense of a file system hierarchy). One only has to ensure that the root directory is present in that set of directories for every object.

Automatic verification of assertions by the Alloy analyzer offers an interesting possibility. Every model has a set of properties which must always verify in the model (such properties are called *invariant*). As shown above, one can write such properties using *facts* and Alloy will guarantee those properties verify for every instance of the model. Actually, Alloy doesn't prove such properties for every instance of the model. What it actually does is to show only instances of the model which absolutely verifies all the facts.

When modeling a system, it is important to somehow prove that an operation never breaks the set of invariant properties declared. Writing invariant properties as *facts*, proving that an operation doesn't violate the invariant for a given scope is impossible. Why? Because Alloy only creates atoms which verify all facts, and therefore the operation will always be applied on instances of the model which already verify the invariant. A complete proof that some operation maintains an invariant needs to check it for every possible instance of the model and show that if the invariant verifies before the operation, it will also verify after the operation. To show this method, let us start by defining our set of invariant properties in the *mini file system* example using a *predicate*:

```
pred Inv[f : FileSystem] {
  f·root in f·objects
  f·pwd in f·objects
  f·parent in f·objects → f·objects
  no o : f·objects | o in o·^(f·parent)
  no f·root·(f·parent)
  all o : f·objects - f·root | some o·(f·parent)
}
```

The predicate forces a set of restrictions into a certain element of `FileSystem`. Note that some properties captured by the predicate haven't been addressed before. For instance `f.parent in f.objects -> f.objects` ensuring that the parent relation only relates objects existing in the file system. Another property, `all o : f.objects - f.root | some o.(f.parent)` states that every object (except the root directory) must have a father. Finally, a clause stating that no object can be father of itself: `no o : f.objects | o in o.^(f.parent)`.

Assertion

```
assert rmkdirOk{
  all f,f':FileSystem | all d:Dir | Inv[f] && rmdir[f,f',d] ⇒ Inv[f']
}
```

will check whether predicate *rmdir* maintains the invariant defined above. This assertion can be read as follows: if the invariant property verifies in the system before the operation *rmdir* is performed, then the invariant property must verify after the same operation takes place. The Alloy analyzer can be used to check this assertion. Using assertions in this form we can check that all operations in a certain model don't violate the invariant properties, which wouldn't make sense if invariants were specified as facts.

Commands and Scope. Having shown how to declare entities in models (*signatures*), how to specify the operation behavior (*predicates* and *facts*) and how to check some property in the model (*assertions*), it is time to show how Alloy animates models by showing models instances.

There are two types of commands in Alloy: *run* (animation) and *check* (verification). Command *run* uses the analyzer to retrieve model instances which verify some predicate. Command *check* searches for counter-examples to a certain assertion:

```
run rmdir
check rmkdirOk
```

Using commands in this way, the analyzer executes them to a maximum of three atoms in each signature (by default). One can explicit tell how many such atoms are wanted (**scope**) per signature:

```
run rmdir for 4 but 2 Dir
check rmdirOk for 5 but exactly 4 FileSystem
```

Increasing the size of assertions' scopes slows down verification but it is required as a means to achieve a reasonable degree of trust in the model. Alloyś underlying logic is *first order logic*, which is undecidable [5]. This means that there is no algorithm able to determine whether arbitrary infinite formulæ are logically valid. To overcome this problem Alloy presents this notion of *scope*, which bounds a number of objects to each type. This procedure transforms the infinite propositional formulæ of first order logic into finite ones that can be validated with a SAT solver [43].

By executing a *run* command Alloy shows instances of its argument predicate. Suppose one wants to observe the *mkdir* predicate running on some file system. One starts by defining another predicate,

```
pred mkdirTst[f,f':FileSystem, d:Dir]{
  Inv[f]
  mkdir[f,f',d]
}
```

which guarantees that *mkdir* will run on states which are valid file systems, as prescribed by the invariant property. By running

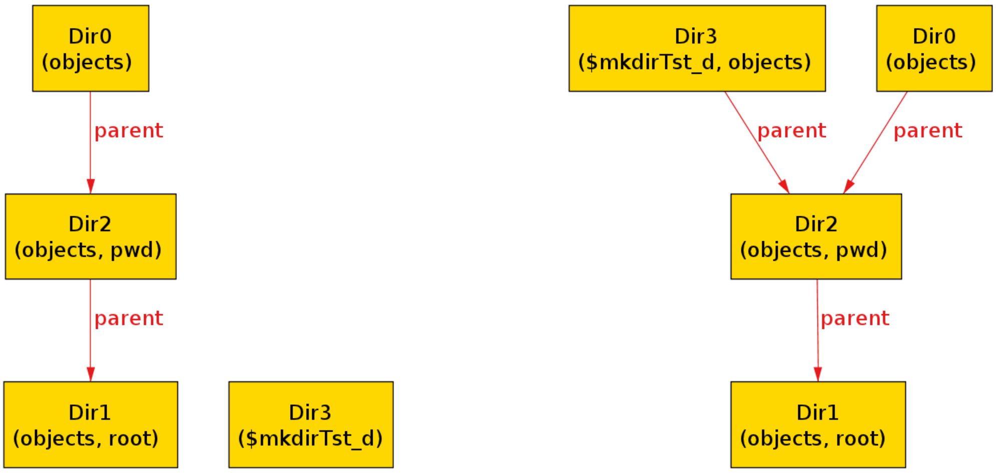
```
run mkdirTst for 4 but exactly 2 FileSystem
```

Alloy will show the creation of empty directories in all possible file system instances with at most four objects. One such instance is given in Figure 2.1.

Figure 2.1 shows how a directory *Dir3* is created inside the *pwd* directory *Dir2*. Such a visual representation of models is a winning factor of Alloy. Representing atoms as boxes and relations between them as arrows recalls the Unified Modeling Language (UML) notation style. Because Alloy can show models in this way, people with different background can understand and evaluate models (this is particularly important if one is using Alloy to do Requirements Engineering (RE) and needs to show use-cases to clients). This visual representation of models is also used to show counter-examples of invalid assertions. Thanks to such counter-examples, one can easily find which invariant property the predicates are violating.

Alloy is a versatile tool which doesn't force one to model in a fixed manner. Sometimes, problems are easily modeled with different approaches than the one presented thus far, particularly if one has dynamic relations which evolve over time. Below we will see how to efficiently specify and model-check problems of this kind.

[5] A decision problem is a yes-or-no question. The problem is undecidable if is impossible to construct a algorithm that always stops and outputs the correct yes-or-no answer.



(a) *File System before the operation*

(b) *File System after the operation*

Figure 2.1: *Make directory operation*

2.1.1 Modeling Idioms

Today programmers don't need to solve from scratch every problem they are faced with. There are some common problems which appear so often that a solution is available which has already been widely studied and applied. These common solutions are usually called *Design Pattens* [27]. A pattern is a reusable solution to a commonly occurring problem which programmers have to solve. When the programmer is faced with a problem which has already been studied, he just need to search for the pattern solution and adapt it to the specific situation. When design patterns are to be applied in a OOP context, they often refer relational properties between classes and objects. This leads one to wonder if there are such patterns in Alloy modeling. In fact they exist and are called *Modeling Idioms*.

In Alloy there are two ways of specifying a state of a model: *globally* or *locally* [43]. These two ways of representing models states are called *idioms* and result in different models for the same problem. When modeling a problem using the global state idiom, the state of the model is represented by a signature which contains all relations of the model. On the other hand, if one chooses to model the problem using the local state idiom, the state of the model will be the set of all signatures which contain all relations present in the problem. As example, consider for example a colored light bulb:

```
abstract sig State{}
sig On, Off extends State{}
one sig Color{}

sig LightBulb{
  color : one Color,
  state : one State
}
```

It has a color and it is always in one of two states (either `On` or `Off`).

As the state of the light bulb changes over time, at some point in time it can be on or off. The following piece of Alloy models the dynamic behaviour of the light bulb in the global state idiom:

```
sig Time{
  s : LightBulb
}
```

The `Time` signature allows one to see how the color of a light bulb evolves over time. Using this global state idiom we can see clearly which relations are static and which don't.

This idiom opens an interesting possibility: it can be used to generate traces of execution of an Alloy model as presented in [47]. A trace of a state machine is a finite sequence of transitions starting from a initial state and evolving by a series of constraints. To simulate a trace using Alloy we just have to define a global state like the `Time` signature and then restrict its evolution by adequate predicates:

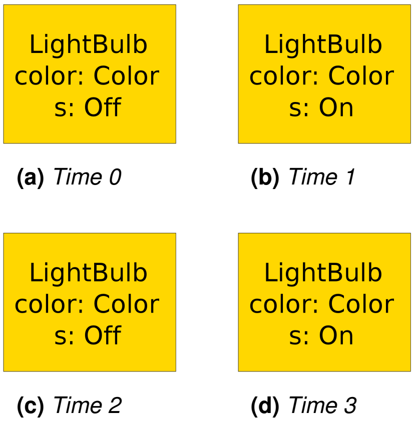
```
open util/ordering[Time]

fact{
  (first·s)·state=Off
  all t:Time, t':t·next | LightOn[t,t'] || LightOff[t,t']
}

pred LightOn[t,t':Time]{
  t·s·state = On
  t·s·state = Off
}

pred LightOff[t,t':Time]{
  t·s·state = Off
  t·s·state = On
}
```

If one asks Alloy to give instances of the model a `LightBulb` will turn up whose state is changing over time, but its color remains constant, as shown in in Figure 2.2.

The other options is to model the light bulb using the local state idiom. In this option, each dynamic relation is added with a new column to represent its evolution over time:

```
sig Time{}

sig LightBulb{
  color : one Color,
  state : one State → Time
}
```

The local state idiom is the natural approach to specify dynamic relations because when a relation is dynamic this is represented in the signature it belongs to and not in some other



(a) *Time 0* **(b)** *Time 1* **(c)** *Time 2* **(d)** *Time 3*

Figure 2.2: *Running the* `LightBulb` *model in global state idiom.*

signature, mixed with completely different relations. The local state idiom allows for an easy conversion between Alloy models and the UML [28].

These two idioms give some flexibility in choosing one idiom or another according to the user intuition. In addition, converting models in one idiom to another is very simple. In [30] we can find some formal reasoning about both idioms and a simple refactoring technique for changing the idiom of a model.

2.1.2 Alloy and Relational Calculus

The Alloy lemma *"In Alloy everything is a relation"* has already been quoted. It is this characteristic of Alloy which allows us to relate its constructions to well known operators and entities of the relational calculus [6]. In Chapter 5 we will see how such a mapping between relational calculus and the Alloy language allow us to connect two different approaches for analyzing requirements.

Studying relational calculus is important to every programmer or system modeler. A program is nothing else than an agglomerate of several types of relations combined together to achieve a clear purpose. Sometimes programmers mistakenly call functions to relations and vice versa. Because functions are only a particular case of relations, their properties are more specific. Thus a careful study of the relational calculus will definitely represent a quality improvement in a programmer's work allowing for a deeper knowledge of code properties and behavior. This combination of relational calculus and system modeling usually results in an clearly understanding of system's requirements and their restrictions [5].

Binary relations. Relations are everywhere, eg. whenever we refer to people addresses or phone numbers, colors of objects, etc. As an example take the following phrase "John **lives at** 767 Fifth Avenue". In this phrase we have a relation *lives at* which is binary and relates people with their addresses. In this particular example, relation *lives at* relates *John* with *767 Fifth Avenue*. A binary relation always relates objects from two domains. If two objects a and b are related by a relation R this is formally asserted by writing aRb.

As seen above, the specification of binary relation *lives at* in Alloy is easy to achieve:

```
sig Person{
  livesAt : set Address
}

sig Address{}
```

In relation algebra all relations are binary and can therefore be represented as arrows between domains. Relation `livesAt` above is one such relation, relating elements from set *Person* with elements from set *Address*. In arrow notation: $Person \xrightarrow{livesAt} Address$. Alloy allows for different ways of representing relations, either using quantifiers and bound variables or the point-free style there are no variables or quantifications in formulæ [2]. The latter style is more terse but far more economic and elegant. The change between the two is referred to as the *pointfree transform* in [70]. Quoting this reference:

> *"The pointfree transform offers to the predicate calculus what the Laplace transform offers to the differential/integral calculus: the possibility of changing the underlying mathematical space so as to enable agile algebraic calculation."*

Pointfree notation is grounded on the composition operator (.) of relations. Given two relations $A \xrightarrow{R} B$ and $B \xrightarrow{S} C$, one will say that a is related with c by the composition of R and S iff $\exists b \in B :: aRb \wedge bSc$. This is written as $a(R.S)b$. This corresponds to Alloy's "dot join" operator, which coincides with relation composition for binary relations. Together with converse (see below), this provides much of what is required for writing models in Alloy using the pointfree style.

In order to see how pointfree notation is useful to express relational properties suppose one wishes to say that a relation $A \xrightarrow{R} B$ is simple (or functional). Using the usual pointwise style, one would write this property as

$$\forall x \in A, \forall y, z \in B :: xRy \wedge xRz \Rightarrow y = z \tag{2.1}$$

and encode this in pointwise Alloy, leading to predicate:

```
pred SimpleWise{
  all x:A, y,z:B | x in r.y and x in r.z ⇒ y==z
}
```

To write the same in the pointfree style we need to introduce some important, generic notions. The first is the notion of *converse* of a relation. Every relation $A \xrightarrow{R} B$ has a converse R° defined as:

$$\langle \forall a \in A, \forall b \in B : aRb : b(R^\circ)a \rangle \tag{2.2}$$

Thanks to converse, one can define the so-called *kernel* of a relation as: $ker\ R = R.R^\circ$ and its image as: $img\ R = R^\circ.R$. Kernel and image are converse-dual definitions:

$$ker(R^\circ) = img(R) \tag{2.3}$$

$$img(R^\circ) = ker(R) \tag{2.4}$$

Finally defining the *identity* relation as $a\ id\ b$ iff $a = b$, one can now easily write the pointfree counterpart of (2.1) as follows:

$$ker\ R \subseteq id \tag{2.5}$$

The economy of this definition contrasts with definition 2.1. Interestingly, Alloy is able to encode this pointfree definition as well as the pointwise one, in this case:

```
pred SimpleFree{
  ~r·r in iden:>B
}
```

Where id is represented as `iden:>B` and the composition is written backwards in Alloy. The predicate `SimpleFree` is simpler and more elegant than `SimpleWise`, allowing for easier calculations.

Alloy supports the full range of relations, including functions. In relational calculus a function f is regarded as a relation which is both simple and entire,

$$f.f^{\circ} \subseteq id \wedge id \subseteq f^{\circ}.f$$

(In the pointfree notation, a relation f is said to be *entire* iff $id \subseteq f^{\circ}.f$). As learned in elementary school, “a function assigns exactly one output to each input”. Declaring functions in Alloy is straightforward using multiplicity `one`:

```
sig A{
  f : one B
}
```

What about combining functions? Functional composition is derived from the relational one: given functions f, g, their composition is $(f.g)x \triangleq f(g(x))$. So, in Alloy we compose functions in exactly the same way as relations.

Pairing and disjoint union. In [6] we find two important relational combinators for gluing functions which do not compose: the **split** and **either** combinators (also called *products* and *coproducts*, respectively).

Suppose we have two functions which share the same source: $A \xrightarrow{f} B$ and $A \xrightarrow{g} C$. There must be a function $\langle f, g\rangle$ which pairs both outputs, that is, such that

$$\begin{aligned} \langle f, g\rangle :\ & A \longrightarrow B \times C \\ \langle f, g\rangle a \triangleq\ & (f(a), g(a)) \end{aligned} \tag{2.6}$$

holds. The relational calculus provides a precise mathematical foundation to study relations and visual representations of them. Relation arrows allow us to draw diagrams showing relational types and restrictions [24]. For example, the *split* transformation can be displayed in the following diagram,

$$\begin{tikzcd}
B & B \times C \arrow[l, "\pi_2"'] \arrow[r, "\pi_2"] & C \\
& A \arrow[ul, "f"] \arrow[u, "\langle f,g\rangle"'] \arrow[ur, "g"'] &
\end{tikzcd}$$

where π_1 and π_2 are projections such that $\pi_1(a, b) = a$ and $\pi_2(a, b) = b$.

Functional *splits* allow us to join two functions with same domain into a single one. To see how we can encode this combinator in Alloy let us first define projections π_1 and π_2:

```
fun p1 [R : univ → one univ] : univ {
  R·univ
}

fun p2 [R : univ → one univ] : univ {
  univ·R
}
```

We can implement the projection through the composition operator. With this projections, the split of two function f, g can be simulated as:

```
sig K{
  f : A → one B,
  g : A → one C
}

fun split[R: A] : B→ C{
  R·(K·f)→  R·(K·g)
}
```

where $B \rightarrow C$ is the product operator making $\langle f, g\rangle$ typed as: $\langle f, g\rangle : A \longrightarrow B \times C$. This operator will be used in Chapter 5 to construct the kernel of algebra for combining different signatures from different Alloy models.

The split combinator glues functions with the same domain. The dual operator in relational algebra is referred to as *either*. To defined it we need the two injections

$$\begin{tikzcd} B \arrow[r, "i_1"] & B + C & C \arrow[l, "i_2"'] \end{tikzcd}$$

which tag elements as they belong to A or B and inject them into the disjoint union $A + B$:

$$i_1 b = (t1, b) \text{ and } i_2 c = (t2, c)$$

In literature the function i_1 and i_2 are usually called ”injections“ [69]. This injections can be easily mapped to Alloy using the *extends* and *abstract* keywords:

```
abstract sig D{}
sig B extends D{}
sig C extends D{}
```

This declares $D = B \cup C$ and $C \cap B = \emptyset$. Because injection functions are just tagging functions, i.e, they decide if a value is in B or C we can implement them as:

```
pred i1[d:D]{
  d in B
}

pred i2[d:D]{
  d in C
}
```

Finally, given functions $B \xrightarrow{f} A$ and $C \xrightarrow{g} A$, the either function $[f, g]$ (either f or g) is defined by:

$$\begin{aligned} &[f, g] : \ B + C \longrightarrow A \\ &[f, g]x \triangleq \begin{cases} x = i_1 b => f(b) \\ x = i_1 c => g(c) \end{cases} \end{aligned} \tag{2.7}$$

If we use the previous Alloy mapping for functions $i1, i2$ and the disjoint union $B + C$, this function can be easily mapped to Alloy as:

```
sig K{
  f : B → A,
  g: C → A
}

pred either[d: D, k:K]{
  i1[d] ⇒ one d·(k·f)
  i2[d] ⇒ one d·(k·g)
}
```

As happens with splits, the either construction will be used in Chapter 5 to develop interesting operators which combine Alloy models.

2.2 Uppaal

Uppaal is a toolbox developed in the universities of Uppsala and Aalborg. It is useful to model real-time and concurrent systems whose models can be created using a collection of non deterministic processes with real-time restrictions. Uppaal has been widely adopted to model such systems since its creation [41, 37, 36, 8].

In Uppaal these processes are represented as a collection of timed automata [6] which communicate using shared variables. One can model a system through a timed automaton together with a background language allowing the definition of data structures and functions (which can be called when the automaton transits). Uppaal not only provides time automata support and such a language but also has a verifier allowing for the verification of properties in a logic which is a subset of Computation Tree Logic (CTL) [59]. Below we will present a more detailed explanation of the modeling and verification language of Uppaal.

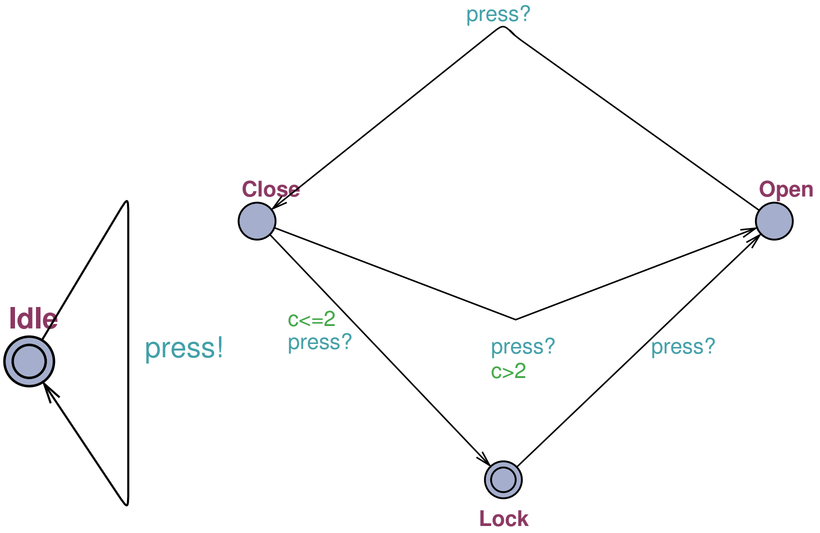
2.2.1 Specification

Models in Uppaal are timed automata together with data structures and functions which operates on bounded variables. The clock variables in Uppaal are represented through real-time values and all clock variables progress synchronously. The system state is no longer the current location of all automata but also the value of all clock variables and the value of all other variables.

Let us see an example to clearly understand the ingredients of Uppaal. The behavior of a door is to be modeled where the user has at his disposal a button which opens, closes and locks

[6] A timed automaton is a finite state machine with clock variables.

the door. When the door is open, a clique in the button will close the door. When a door is closed, a click in a button will open the door and two quick clicks will lock the door. To model this door we create two timed automata:



(a) *User* **(b)** *Door*

As expected, the doors automaton has three states: Open, Close and Lock. The transitions between these states are made according to the shared variable *press*. Notations *!* and *?* are used to represent an output and an input, following the CSP style. Each time the user presses the button, the doors automaton reacts and transits according to the specification. This automaton differs from a usual automaton with the appearance of the clock variable *c*. This variable is used as a guard to the transitions from state *Close*. If we interpret this variable as a clock we can see that the guard *c<=2* represents exactly the requirement ”when the door is closed, two quick clicks will close it“, ie. if the user presses the button in less then 2 time units the door will close, otherwise the door will open again.

2.2.2 Verification

Uppaal also provides a model-checker that checks properties expressed in a subset of Computation Tree Logic (CTL). This model checker is designed for interactive and automated analysis of system behavior and provides counterexamples when system models fail to verify their properties. The subset of CTL used in Uppaal allows the specification of path formulæ (reachability, safety and liveness) and state formulas.

State Formulas. State formulæ are expressions whose evaluation depends only on the current state and not on the model’s behavior (eg: *i>0, x==15, x==15 and i>0*, etc). Moreover, if P is a process and l a location, $P.l$ is a state formula which verifies if process P is in location l. Another useful state formula in Uppaal is the *deadlock*. The *deadlock* state formula verifies for some state if there is no possible transition from that state. This state formula can be combined into a safety property to verify that a given model is deadlock free.

Reachability Properties. Given a state formula φ, reachability properties ask whether there exists a path starting from the initial state where φ is eventually verified along that path. To specify such a formula in Uppaal one writes $E <> \varphi$. Properties of this kind help to verify that something desirable can eventually happen. Suppose we have a model of a mutual exclusion zone. We could write a reachability property asking if there is a path such that the mutual exclusion zone has some process inside.

Reachability properties alone don't prove model correctness but they can give insight on their behavior.

Safety Properties. These properties express that something bad will never happen. Given a state formula φ we have two ways of express safety properties in Uppaal. Wishing to express that φ must verify in every state, one writes $A[]\varphi$; wishing to express that there is a maximal path such that φ always verifies in every state, one writes $E[]\varphi$ instead.

Safety properties represent behavior invariants in models and guarantee that something catastrophic will never happen. In the mutual exclusion zone model, a safety property might be that there is no state whether two processes are inside the mutual exclusion zone.

Liveness Properties. Liveness properties are of the form: *"Something good will eventually happen"*. Again thinking in a model of a mutual exclusion zone, a liveness property could be *"Every process eventually enters in the mutual exclusion zone"*. Given a formula φ, a liveness property in Uppaal is written as $A <> \varphi$. Liveness properties are often expressed in the following way: given the state formulæ φ and ψ, when φ verifies then eventually ψ will verify. Liveness properties of this kind are written in Uppaal as $\varphi \to \psi$.

Uppaal includes several other ingredients (state invariants, definition of new data types, parameterized templates, etc) which make this tool widely used to verify concurrent and real-time systems. For a detailed explanation of all the features of the Uppaal toolbox please see [29].

2.3 Summary

This chapter presented two tools — Alloy and Uppaal — which allow for different ways to do formal modeling. Alloy is a versatile tool with a simple, yet powerful language which welcomes abstraction skills. The language is usually well received by programmers because of its similarity with OOP concepts and its simple mathematical underpinning. However, Alloy isn't suitable for everything. When one needs to model a system restricted by temporal constraints and show which behaviors such a system have (in terms of its states and transition), Uppaal is a better choice. Because the mathematics underlying Uppaal are more difficult to understand than Alloy, programmers tend to avoid tools like this.

Stressing on complementary aspects of these tools, the following chapters will show how they can be used together to improve requirement analysis by allowing a requirement engineer to find inconsistencies and improve requirements quality. Tools of this kind offer features which are particularly interesting in requirement analysis. They allow for defining versatile models, without restriction on problem domains and both produce visual models which help in applying them in industry.

Chapter 3

Requirements Engineering Assisted by Formal Methods

3.1 Introduction

Requirements for software systems appear when clients present their needs related to expectations for the solution of a particular problem. The job of a requirement engineer is gather client needs and put them together into what is called a ”requirements document“. Usually requirements are elicited through a set of meetings with clients and the requirements engineer uses these meetings to write what is required in NL. Although NL presents several disadvantages, it is after all the “natural” way to express one’s thoughts in order to allow both clients and technical personnel read them. It is absolutely mandatory that requirements documents be understood by clients, because they need to validate them before the system starts to be developed.

Once requirements are compiled in some document, software companies invest in building *models* from them, often using visual languages. These models present several advantages: they provide a deeper insight on the problem and sometimes help to discover inconsistencies in the document. They may also help in generating the source code.

When one wants to construct a model from a set of requirements, the usual approach is to incrementally add to the model while reading requirement items one after the other. If a new set of requirements is found necessary of missing, the requirement analyst looks at the model and re-factors it in order to take them into account.

As starting example consider the set of requirements for a missile launcher acting in some battlefield given in Table 3.1. In a set of requirements like the example there can be several inconsistencies which pass undetected using NL. A good choice for a modeling tool which finds such kind of inconsistencies is the Alloy model checker, as seen in the previous chapter. Let

Requirement Number	Description
01	The missile launcher shall be able to fire missiles.
02	The missile launcher fires missiles if and only if a warning signal is received.
03	Warning signals are launched when an enemy is spotted by an ally.

Table 3.1: *Missile Launcher Requirements*

us simulate the reaction of a requirement analyst to these requirements. She/he will start by identifying the structural components of the model: entities and relations between them. Entities `Enemies`, `Warning Signals` and `Missile` don't exhibit any apparent relation between them:

```
sig Enemy{}
sig WarningSignal{}
sig Missile{}
```

Further inspection of the requirements will identify two other entities: `Ally` and `Missile Launcher`. Because an `Ally` can spot `Enemies`, it must have a relation which represents this:

```
sig Ally{
  spoted : set Enemy
}
```

Finally, a `Missile Launcher` can launch missiles and issue warning signals:

```
sig MissileLauncher{
  launched: set Missile,
  warning: lone WarningSignal
}
```

`Warning Signal` reception is represented by the relation `warning` which relates a `Missile Launcher` with zero or one `Warning Signal`.

The next step in the modeling is to consider the actions which can take place. First, there must be an action which emits a warning signal:

```
pred EmitWarningSignal[a:Ally, ml,ml':MissileLauncher]{
  some a·spoted
  no ml·warning
  no ml·launched
  one ml'·warning
}
```

This predicate has a *pre-condition* which states that there must be some `Enemy` spotted. If the *pre-condition* verifies, a warning signal is launched. The launching of warning signals is another operation present in the model:

```
pred LaunchMissile[ml,ml':MissileLauncher, m:Missile]{
  one ml·warning
  ml'·warning = ml·warning
  ml'·launched = ml·launched + m
}
```

This operation has a *pre-condition* which ensures that a warning signal has been launched. It relates a missile launcher with a fresh new missile. Finally, one can declare the overall invariant property as a predicate:

```
pred Inv[ml:MissileLauncher]{
  some ml·launched <=> one ml·warning
}
```

This invariant is derived from the requirement that a missile can be launched if and only if a warning signal has been emitted.

With predicates representing actions and the model invariant represented in the predicate *Inv*, we can check that none of the operations breaks the invariant by asserting:

```
assert EmitWarningSignalOK{
  all a:Ally, ml,ml':MissileLauncher |
    Inv[ml] && EmitWarningSignal[a,ml,ml'] ⇒ Inv[ml']
}

assert LaunchMissileOK{
  all ml,ml':MissileLauncher, m:Missile |
    Inv[ml] && LaunchMissile[ml,ml',m] ⇒ Inv[ml']
}
```

Although mathematical models such as the above provide help for developers and all people involved in a software project, creating only one model of the whole set of requirements doesn't benefit from all advantages they can give. A recent declaration of David Parnas in [71] expresses the approach that requirements engineers should follow when create mathematical models of requirement documents:

> *”One of the most important roles that mathematics could play in software development would be to provide precise, provably complete, easy-to-use, testable documents.“*

Requirements documents should be handled in one-to-one correspondence with models, i.e., one should be able to easily identify which model parts correspond to which specific requirements. Recall from Section 1.1 that this is known as *traceability*. Such easy correspondence allows requirement engineers who later have to change some requirement to immediately spot where is the impact of the changes in the model of the whole system. A good requirement analysis method should not only provide this correspondence but also provide techniques, tools and guidelines helping the requirement analyst to create models in a fast but paced way, thus answering to critics who say that formal methods are too slow and difficult to use.

From an implementation point of view, it makes sense to have both formal specifications and NL in the requirements documentation. A requirement is usually a description of what the machine or software should do in order to satisfy some useful purpose. Documenting and processing requirements is an informal task, but the machine or software that requirements call for are completely formal [45, 46]. Methods which bring requirement descriptions closer to their formal specification are welcome by developers and usually result in significant improvements on end-product quality [68].

Below we will see a method to analyze requirements which provides such a one-to-one mapping between requirements and models, techniques to re-write requirement descriptions based on conclusions gathered from the model and automatic generation of the whole system model from each requirement step contribution. The method will be supported by a tool-set.

3.2 Methodology

This section will present in detail a methodology to analyze requirements using formal methods tools such as Alloy and Uppaal. The tool-set which supports the methodology is described in Section 3.3.

Binding models to requirements. The overall methodology consists in enriching the original requirements document which the corresponding formal model. The first principle to ensure correspondence between a requirement and its formal model is to write the model right after the particular requirement text and bind both together in the document. As an example, suppose we want to model requirement 01 from the set of requirements presented in Table 3.1. We could write this requirement and its model as:

Structured Requirement 1 Requirement and Model

Requirement :
The missile launcher shall be able to fire missiles.

Model :

```
sig Missile{}

sig MissileLauncher{
  launched: set Missile,
}

pred LaunchMissile[ml,ml':MissileLauncher, m:Missile]{
  ml'·launched = ml·launched + m
}
```

Binding models to requirements provides in-place evidence of the mathematical meaning of each requirement. Clearly, models of requirements are dependent on each other. In general, requirements talk about things mentioned in other requirements. As requirements do, models rely on entities modeled in previous steps.

We need to define a method which not only binds models to requirements but also creates a formal relation between models, allowing the user to obtain at each modeling step (one per requirement) what *the system model thus far* is. An obvious solution to this problem would consist of forcing requirement analysts to write the whole model at each requirement step. This would also give a mathematical basis to each requirement and show the system model until that point in the document to the reader. Although this solution would work for small requirements documents, it is impracticable for medium and large size documents consisting of dozens of requirements (the later in the text the more unreadable models, due to their sizes).

Our approach is based on a particular standardization of model writing. Suppose the requirement engineer wants to model requirement 02 from the example presented in Table 3.1. Having already written the model of requirement 01, the user is asked to write only what is new in requirement 02 (new predicates or entities) and likely changes to what has been written in the previous steps. This is because requirement engineers quite often write later what they should have written in the first place.

Bearing this principle in mind, we can write the structural part of the requirement 02 model as:

```
sig WarningSignal{}

sig MissileLauncher{
  launched: set Missile,
  warning: lone WarningSignal
}
```

The requirement text mentions a new entity model as `WarningSignal` and a new constraint for launching a missile. This new constraint imposes changes in signature `MissileLauncher` and `LaunchMissile`. Thus the enrichment of `MissileLauncher` coming from the previous step with a new `warning` relation.

By applying the same standard we can derive the set of predicates which complete the model of requirement 02:

```
pred EmitWarningSignal[ml,ml':MissileLauncher]{
  no ml·warning
  no ml·launched
  one ml'·warning
}

pred LaunchMissile[ml,ml':MissileLauncher, m:Missile]{
  one ml·warning
  ml'·warning = ml·warning
  ml'·launched = ml·launched + m
}

pred Inv[ml:MissileLauncher]{
  some ml·launched <=> one ml·warning
}
```

Predicate *EmitWarningSignal* models the emission of a warning signal and the predicate *Inv* verifies the condition that missile launcher should only fire missiles when there is a warning signal.

Note how the predicates considered at this point are the ones induced by requirement 02 and another one which had to be quoted from requirement 01 and changed (`LaunchMissile`). At this point one can resort to Alloy to prove the consistency of the operations with respect to the invariant property stating that a missile shall be launched if and only if a warning signal appears. In order to verify that predicates *LaunchMissile* and `EmitWarningSignal` don't break the invariant one adds assertions `EmitWarningSignalOK` and `LaunchMissileOK`:

```
assert EmitWarningSignalOK{
  all ml,ml':MissileLauncher |
    Inv[ml] && EmitWarningSignal[ml,ml'] ⇒ Inv[ml']
}

assert LaunchMissileOK{
  all ml,ml':MissileLauncher, m:Missile |
    Inv[ml] && LaunchMissile[ml,ml',m] ⇒ Inv[ml']
}
```

Writing the model in this stepwise manner is far readable than pasting the whole model of the system at each step. In this way the user can easily see what changes and new additions requirements bring to the system model thus far, adding to traceability.

Tool support relies on requirement documents available in LaTeX format in which new environments have been added to support the embedding of Alloy models in a *literate programming* style [51]. The tool processes text files and extracts the models from them, chaining the changes in order to derive the current model, up to a given requirement. When the tool finds a predicate, signature or fact, it searches for it in the previous models: if it is already there, it replaces it with the new definition, otherwise it is new and is added directly to the model. Summing up, the approach allows one to incrementally create a model of the requirements through a series of small "*dif-models*" written at each requirement analysis step.

Requirement refactoring through formal modeling. The same standard can be helpful also in actually suggesting changes into the requirements descriptions themselves as consequence of finding inconsistencies in the text unveiled by the formal modeling. In fact, by looking at the model we often finds the opportunity for re-writing the original requirements text in order to make the text less ambiguous and closer to what the model actually prescribes while meeting the client expectations at the same time.

Let us see an example. Requirement 02 in the *Missile Launcher* running example reads: *"The missile launcher fires missiles if and only if a warning signal is received."*. This text suggests the invariant of the system as:

```
pred Inv[ml:MissileLauncher]{
  some ml·launched <=> one ml·warning
}
```

Alloy finds no counter-example for this invariant giving confidence that at least the requirement is consistent with the specification. Although there isn't any counter-example, predicate *Inv* shows us something interesting: the condition says that a missile is launched if and only if a warning signal appears. This is indeed what the requirement says; however, requirement 03 from Table 3.1 tell us that this equivalence should be an implication, i.e, *The missile launcher fires missiles if a warning signal is received.*, resulting in a new version of the *invariant*:

```
pred Inv[ml:MissileLauncher]{
  some ml·launched ⇒ one ml·warning
}
```

This *invariant* makes the model closer to what the text says. If we now check our assertions with this invariant, Alloy continues to find no counter-examples for our assertions, as expected (because the equivalence is stronger than the implication), but we clearly come with a model which is closer to reality and found a misleading requirement description.

This example shows that requirement models should not only be integrated in requirements documents but also their analysis could have immediate impact on requirements. A good requirement analysis method should end with a revised, clean set of requirements and a model supporting all of them.

Requirement layout standard. Below we give the formal, textual structure which is recommended for laying out each requirement item so as to cater for all that has been advocated before:

- **Requirement :** Quotes verbatim the requirement text block as written in the original requirements document.
- **Model :** The formal model educed by the analyst from the requirement text block. This model should be as "small" (unbiased) as possible, that is, it should not contain details which are not referred to by the requirement text.
- **Meta-model :** A graphic representation of the model.
- **Questions and Suggestions :** Questions and suggestions which were raised while modeling the requirement text; these should be discussed among the requirement engineer team in order to validate or disprove them.
- **New Requirement :** The improved requirement text resulting from the above discussion; both versions to be kept for traceability reasons.
- **New Model:** The model of the improved requirement. This model is written following the standard presented above: only differences and new predicates, facts or signatures are written.
- **New Meta-model :** A graphic representation of the improved model.

By writing requirement documents in this manner requirement traceability is ensured. Items *Questions and Suggestions*, *New Requirement*, *New Model* and *New Meta-model* keep track of the changes occurred throughout the requirement analysis process. This process can even be extended if a requirement need more changes (creating new models and meta-models for the changed requirement).

Usually, the *Meta-model* component of the layout is automatically generated by the formal tool associated to the notation used for modeling. It thus requires no intervention from the user. In the case of Alloy, this graphic representation is obtained using Alloy's Application Programming Interface (API), which delivers a diagram with all signatures and relations between them. Readers with little knowledge of Alloy can use these diagrams to better understand requirements.

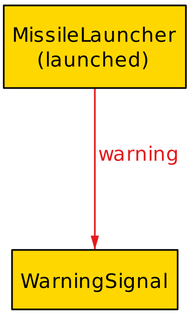
The following page shows the layout of requirement 02 once written in the proposed format.

Structured Requirement 2 Standard to write models

Requirement :
The missile launcher fires missiles if and only if a warning signal is received.

Model :
Model remains equal to model derived for requirement 02 of table 3.1.

Meta-model :



Questions and Suggestions :
Reading the requirement we model it as : a missile is launched if and only if a warning signal appears. Although this is exactly what we see in the requirement description, when we read requirement 03 we see that warning signals are launched when enemies are spotted. This tell us this equivalence should be an implication, i.e, *The missile launcher only fires missiles after the receipt of warning signal.*.

New Requirement :
The missile launcher only fires missiles after the receipt of warning signal.

New Model:

```
pred EmitWarningSignal[ml,ml':MissileLauncher]{
  no ml·warning
  no ml·launched
  one ml'·warning
  some ml'·launched
}

pred Inv[ml:MissileLauncher]{
  some ml·launched <=> one ml·warning
}
```

New Meta-model :
No Changes.

The tool reads this requirement and aggregates the *New Model* part with the *Model*, i.e, replaces predicate *Inv* written in *Model* part with the predicate *Inv* of *New Model* part. In the following section we will see in detail how the tool is implemented in order to support this requirement analysis method.

3.3 Tool Support

The method described above helps the user to develop a formal model of a system from its requirements in a stepwise manner. Without proper tool support, to follow the methodology the user would have to retrieve each model of the requirements at each step and aggregate everything in the document in a *cut & paste* manner — obviously a painful task.

The tool developed to support the methodology creates a running model at each requirement analysis step. This is helpful wherever the client drops some requirement for the system or if the requirement analyst needs to see the model of the whole system but a particular requirement. The tool also generates the meta-models assembled up to each requirement step, automatically creating the diagrams with the meta-data. To relief the user from using the Alloy GUI every time she/he wants to check a property or run a command the tool uses the Alloy API to run the commands without any user intervention.

Global Architecture. The tool is built in Java and uses the Alloy API to perform all tasks related with the Alloy modeling. Requirements are written using the document preparation system LaTeX [55]. Having the requirements in LaTeX allows us to process documents without using an external API, because in LaTeX the documents are written into plain text files. The tool scans the documents and searches for a set of macros defined which contains specific data (models, commands, etc).

For each requirement a folder is created and inside it the requirement is written into a *.tex* containing the same structure as presented in Section 3.2. Figure 3.1 shows how all requirements are included by a main one in order to create a complete requirements document. This is how the tool currently supports requirements documents, but no restriction is made on the format of how the requirements are written. The tool is developed in a modular manner, allowing the user to easily construct a different module to support document preparation systems other than LaTeX.

Requirement document processing is carried out through a series of stages shown in Figure 3.2. Below we will see each step of the pipeline in detail. Although a complete processing of the requirements document consists of all phases in the pipeline, the tool allows the user to use part of the pipeline if she/he wishes so.

LaTeX Preprocessing. The tool starts by extracting models and commands from the LaTeX files. This phase is responsible for obtaining every model and Alloy commands written in the *.tex* files. The environments (tags) listed in Figure 3.3 are available for embedding models in LaTeX sources.

The comment %**check_mark** is used to extract models from *.tex* files. Commands should be written using the macro *runners*, allowing the requirement analyst to hide commands from the reader of the document (readers usually only have interest in the models) and at the same time leaving commands hidden in the document and ready to be called at any time. Macro *moduleutil* can be used to call pre-defined modules of Alloy, e.g., open util/ordering[Missile]. Alloy imposes that the opening of modules must be written before any signature, predicate or facts. Using macro

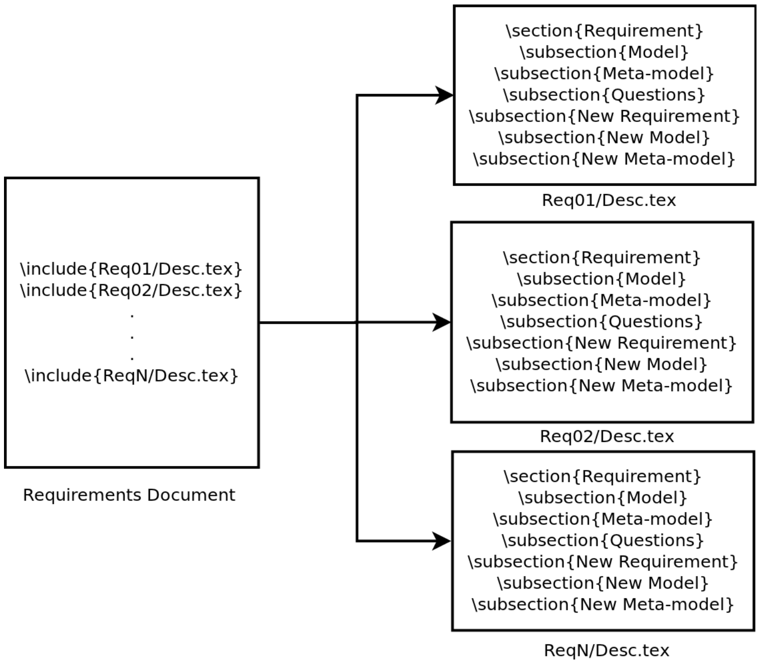


Figure 3.1: *A Requirements Document*

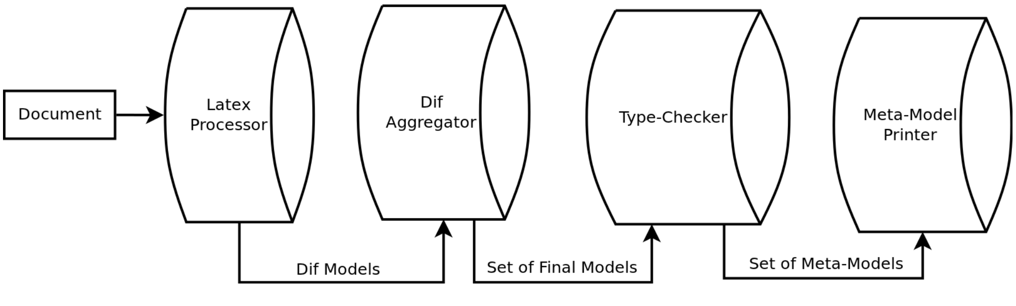


Figure 3.2: *Tool Pipeline*

Models :

```
%**check_mark**
\begin{lstlisting}[frame=single]
Alloy model
\end{lstlisting}
```

Commands :

```
\begin{runners}
Alloy commands
\end{runners}
```

Utils (ordering, sequence, etc):

```
\begin{moduleutil}
Open modules
\end{moduleutil}
```

Figure 3.3: *Macros for writing models*

moduleutil, the tool can easily control this and the requirement analyst has a possibility to write a complete Alloy inside *.tex* files.

Recall from Section 3.2 that a requirement may have two Alloy models, the first reflecting the original requirement text block and possibly a second, revised alternative trying to improve the original. For each *%**check_mark***, the *LaTeX Preprocessor* creates a folder inside the folder of requirement it appears and inside each new folder the tex processor creates a *dif-model*. A *dif model* consists of open commands of Alloy modules (macro *moduleutil*), followed by the Alloy model of the requirement and finally all commands the user defined with the macro *runners*.

This component of the tool pipeline can be changed to support other formats than LaTeX. What this component must provide to the *Dif Aggregation* block is a collection of dif-models, one per each Alloy model of the requirement. Any component which successfully extracts models from requirements can be used in the pipeline without any other changes in other components.

Dif Aggregation. The *Dif Aggregation* component is responsible for creating a complete, running model for each *%**check_mark***. It starts by reading the first and second dif-model and applying the following algorithm to combine them:

Algorithm 1 DiffModels(D1,D2)

Require: Two partial Alloy models D1 and D2
Ensure: An Alloy model which is the evolution of D1 through D2

```
$Preds \Leftarrow GetPreds(D1) + GetPreds(D2)$
$Sigs \Leftarrow GetSigs(D1) + GetSigs(D2)$
$Facts \Leftarrow GetFacts(D1) + GetFacts(D2)$
$Asserts \Leftarrow GetAsserts(D1) + GetAsserts(D2)$
for all $Pred\ P1 : GetPreds(D1)$ do
    for all $Pred\ P2 : GetPreds(D2)$ do
        if $P2.name = P1.name$ then
            $Preds \Leftarrow Preds - P1$
for all $Sig\ S1 : GetSigs(D1)$ do
    for all $Sig\ S2 : GetSigs(D2)$ do
        if $S2.name = S1.name$ then
            $Sigs \Leftarrow Sigs - S1$
for all $Fact\ F1 : GetFacts(D1)$ do
    for all $Fact\ F2 : GetFacts(D2)$ do
        if $F2.name = F1.name$ then
            $Facts \Leftarrow Facts - F1$
for all $Assert\ A1 : GetAsserts(D1)$ do
    for all $Assert\ A2 : GetAsserts(D2)$ do
        if $A2.name = A1.name$ then
            $Asserts \Leftarrow Asserts - A1$
```

The outcome of applying this algorithm to dif-models `D1`, `D2` respectively, is a complete Alloy model `FinalModel2.als`. This model will be saved in the `D2` directory. To obtain the system model for `D3`, we apply the previous algorithm on `FinalModel2.als` and in the *dif-model* `D3`, resulting in a `FinalModel3.als` which will be saved in directory `D3`. Repeating the same procedure for all dif-models will result in all complete models for every *%**check_mark***.

At this phase the requirement analyst can get the complete evolution of the system, without having to manually combine all Alloy models written throughout the document.

Type-Checker. The Alloy GUI is an excellent combination of a friendly user interface and a powerful API. This API is open source and can be used to completely manipulate an Alloy model without using the GUIȮur tool-set makes ample use of the Alloy APIto relieve the requirement analyst from going through the Alloy GUI every time she/he wants to check some property or see if the working model is correct. In the *Type-Checking* phase the tool reads all *FinalModel.als* produced by the *Dif Aggregation* and performs two tasks: type-checking and command running.

Type-checking models through the tool provides immediately feedback to the requirement analyst. If the tool finds any error or warning provided by Alloy, this is reported to the requirement analyst by showing precisely where the error is. This error can result from a real error in the model (caused by some misinterpretation of requirement analyst) or from a real inconsistency in model. Either way, requirement analysts have the opportunity to draw conclusions from the error without going to the Alloy GUI for every requirement.

The command running feature of this phase is also very useful to the requirement analyst. When we are modeling requirements with this methodology we can leave commands in our requirements in order to prove that our predicates are consistent. These commands run every time we use the tool, proving confidence that previous models are still valuable and correct.

Meta-Model Printer. Meta-models are visual representations of Alloy models. They provide quick insight on the system entities and relations between them. The requirement analyst can use these diagrams to show the model to anyone without any knowledge of Alloy. A tool that does all previous steps without users intervention but requires them to print meta-models one by one would behave against the objectives presented above, for two reasons: first the user would have to use the Alloy GUI and manually print the meta-models; and second, as we don't want to show the whole system model in every requirement, we also don't want to show the whole system meta-model in every requirement.

To solve this, the tool uses the set of functions which in the Alloy API manipulate meta-models. In this phase we know that our models are correctly typed and wish to extract correct meta-models from them. The tool goes to every *FinalModel* generated by *Dif Aggregation* and previously type checked in the *Type-Checker* phase, and uses the Alloy API to extract the meta-model. This meta-model isn't enough for our purposes because it contains all signatures and relations of the whole system. The *Meta-Model Printer* step in the pipeline projects this meta-model in every signature which appeared in the dif-model of the requirement, resulting in visual diagrams showing only the entities related to requirements.

3.4 Summary

This chapter presented an approach which invites requirements engineers to write a model in a stepwise manner, as long as they read requirements text chunks. One doesn't need the write the whole model but only what each requirement exactly mentions because a tool will gather all these models and create a complete, running model up to each requirement block. Using a text standard presented in this chapter, requirements engineers can enjoy requirements traceability and model traceability at the same time. This is useful because at any moment one can go back to a requirement and see how its model has evolved. This direct correspondence between models and requirements allows one to check which changes a particular requirement introduced in the model and (together with Alloy) have a graphical representation of each requirement. Thanks to the modeling, requirements engineers can gain insights on the quality of the original text and, if necessary, introduce improvements removing ambiguities or inconsistencies.

In the following chapter we will go through a case study where requirements are brought closer to formal models. The case study was developed using this method and shows that writing models immediately after requirements makes requirements and models easier to understand and analyze.

Chapter 4

Case Study : Partitioning Microkernel

Formal Methods are mostly applied in life critical systems (LCS) in order to ensure safe behaviour. Thus the the techniques described in Chapter 3 must prove effective in coping with requirements documents in the LCS domain. One such system is taken in this chapter as case study.

The starting requirements document, proposed by ***Critical Software*** [1], describes a *Partitioning Kernel* enforcing the *Partition Information Flow Policy* (PIFP) of a *Secure Partitioning Kernel* (SPK) described in [17]. Partitioning by criticality class has been applied since long ago in LCS, whereby functions with different criticality levels are deployed into different computer boards. This kind of system partitioning is hardware implemented and presents some disadvantages: *physical* (power, weight and space increase with the number of boards), *testability* (systems running in a single board are far easier to test) and *reuse* (because functions are distributed across several boards, some simpler functionalities must be replicated to each board).

In order to overcome this disadvantages other approaches have emerged such as eg. the ARINC 653 [40] together with DO-297 (Integrated Modular Avionics, IMA). ARINC 653 introduced the concept of partitioning microkernel which allows to separate functions deployed into different real-time operating systems (RTOS) as shown in Figure 4.1.

A consistent and reliable partitioning kernel must be certified under the Common Criteria (CC). CC is an international standard which specifies security and assurance requirements which guarantee that systems are developed and verified certain ways ensuring confidence and reliability [80].

4.1 Document Structure

The requirements document specifies not only functions that the system must perform, but also a numbers of assurance requirements which allow the system to be certified under the CC. The requirements are thus divided into:

- **Functional Requirements**: Requirements concerning the kernel functionalities.
- **Assurance Requirements**: Requirements which the system must present in order to be verifiable under the CC.

[1] See: `http://www.criticalsoftware.com`.

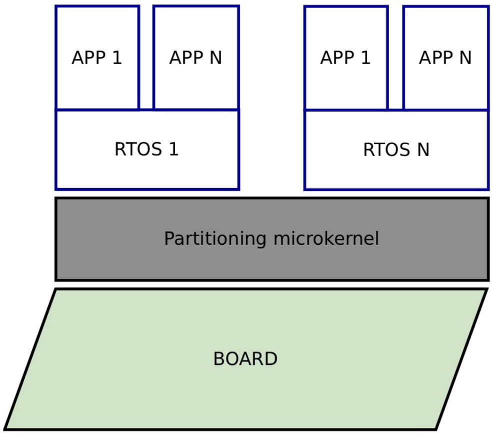


Figure 4.1: *Partitioning Microkernel*

Altogether, these two types of requirements sum up to 158 requirements, 84 from assurance and 74 from functional requirements. In order to better see the outcome of the methodologies presented in Chapter 3, emphasis is put on functionality rather than assurance requirements.

A partitioning kernel is a system composed of several ingredients and requirements indicating how processes and partitions communicate, which real-time constrains there are, how to recover from errors, etc. Functional requirements are thus divided into:

- **Configuration Management System (CMS)**: A set of parameters that must be defined before systems start.
- **Health Monitor (HM)**: Monitoring and reporting of hardware and software failures.
- **System Errors and Faults (SEF)**: Definition of all error types and who is in charge of handling them.
- **Recovery Actions (REC)**: How the system should act in the presence of errors.
- **System Tables (SYT)**: Requirements concerning some information tables of the kernel.
- **Partition (PRT)**: Partition attributes, rules on how to schedule partitions, partitions modes, etc.
- **Process (PRC)**: Processes behaviors, attributes and scheduling rules.
- **Partition Information Flow (PIFP)**: Requirements on how processes from different partitions (can) communicate with each other.

Requirements Selection. Looking only at functional requirements only, there are 74 requirements to account for. After a careful analysis of these requirements became clear that most of requirement are structural: they say which attributes some kind of entity should have. If all functional requirements were considered, would lead to a large model model with very few operations and interesting aspects.

Having no operations this model would be uninteresting because we can see none or little animation and proving invariant properties would be impossible. In order to overcome this drawback, let us try to capture the important operations of a partitioning kernel. Looking at Figure 4.1 we see that a partitioning kernel performs scheduling at two levels: partitions and processes. Scheduling is a highly important operation in any operating system whose, and it is properties can be modeled, animated and checked. To select the requirements which talk about Scheduling, we proceed as follow:

1. Select partition and process requirements which talk about the scheduling characteristic: Requirements PRT007 and PRC013.
2. Select all requirements needed to understand requirements PRT007 and PRC013: Requirements PRT001, PRT004 and PRC004.

This criteria leave us five requirements which allow for modeling of the system scheduler and all constraints associated, resulting in an interesting model where important conclusions can be draw.

4.2 Requirements Modeling

Having selected the relevant requirements, the techniques described in Chapter 3 could now be applied to them. Requirements and models where gather into a document with the same structure presented in Chapter 3. When a temporal requirement need to model the requirements, and the model associated to. The reader can find the complete document in Chapter A of the Appendices.

Each requirement model will be followed by an explanation of the modeling options.

PRT001 - Partition Attributes.

Requirement Number	PRT#001
Title	Partition Attributes
Description	Each Partition shall have the following attributes: • Duration: the amount of processor time (minimum and maximum quotas) given to the partition every period of the partition. • Period: defines the activation period of the partition, and is used to determine the partition's runtime placement within the core module's overall time frame.
Rationale	N/A.

This requirement defines the structure of a partition. In Alloy, this is achieved by representing a partition as a signature:

```
sig Partition {
  minimum : one Int,
  maximum : one Int,
  period : one Int
}
```

Using the *Int* construction of Alloy, we can easily model the *duration* and *period* requirements for a given partition. This defines the structure of the partition, but alone, allow for negative periods or maximum duration quotas. Let us add a *fact* which bounds the previous attributes to the correct values:

```
fact Constraints{
  all p:Partition | gt[p·minimum,0]
  all p:Partition | gt[p·maximum,0]
  all p:Partition | gt[p·period,0]
  all p:Partition | gt[p·maximum,p·minimum]
  all p:Partition | gte[p·period,p·maximum]
}
```

Now, the partition in the model correspond to the partition described in the requirement. It has all the attributes specified and their values are in the correct range. As explained in Chapter 3, this method has a tool support which automatically generates a visual representation of the requirement. This requirement induces the following diagram:

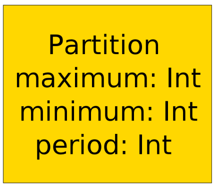


Figure 4.2: *Partition Model*

These diagrams are a visual representation of requirements and can be shown to non-technical people (eg: Clients). We can use Alloy to instantiate the model and get some examples of valid partitions in a partitioning kernel:

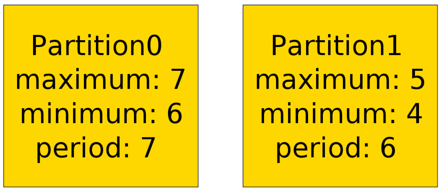


Figure 4.3: *A model instance*

In this model instance we have a system composed of two partitions, each one with its own attribute values, as described by the requirement.

PRT004 - Partition Modes.

Requirement Number	PRT#004
Title	Partition Modes
Description	The Partition shall have the following modes: • IDLE: In this mode, the partition is not executing any processes within its allocated partition windows. The partition is not initialized (e.g., none of the ports associated to the partition are initialized), no processes are executing, but the time windows allocated to the partition are unchanged. • NORMAL: In this mode, the process scheduler is active. All processes have been created and those that are in the ready state are able to run. The system is in an operational mode.
Rationale	N/A.

This requirement tell us that partitions have pre-defines modes, and they change over time. This adds a new ingredient to partitions: a new attribute which changes over time. Because this attribute will change over time for a given partition, a good modeling idiom to model it is the *Local State* idiom presented in Section 2.1.1:

```
open util/ordering[Time]

sig Time {}
sig Mode {}
sig Normal, Idle extends Mode {}

sig Partition {
  minimum : one Int,
  maximum : one Int,
  period : one Int,
  mode : Mode one → Time,
}
```

We've represented partition modes *Idle* and *Normal* as extends of a *Mode* signature together with a new attribute to *partition* which maps a mode into a specific point of time. This allow us to model and represent the evolution of partitions modes as the scheduling proceeds. Alloy now gives us the following visual representation of the model:

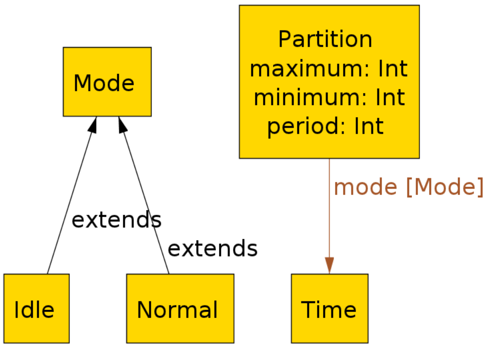
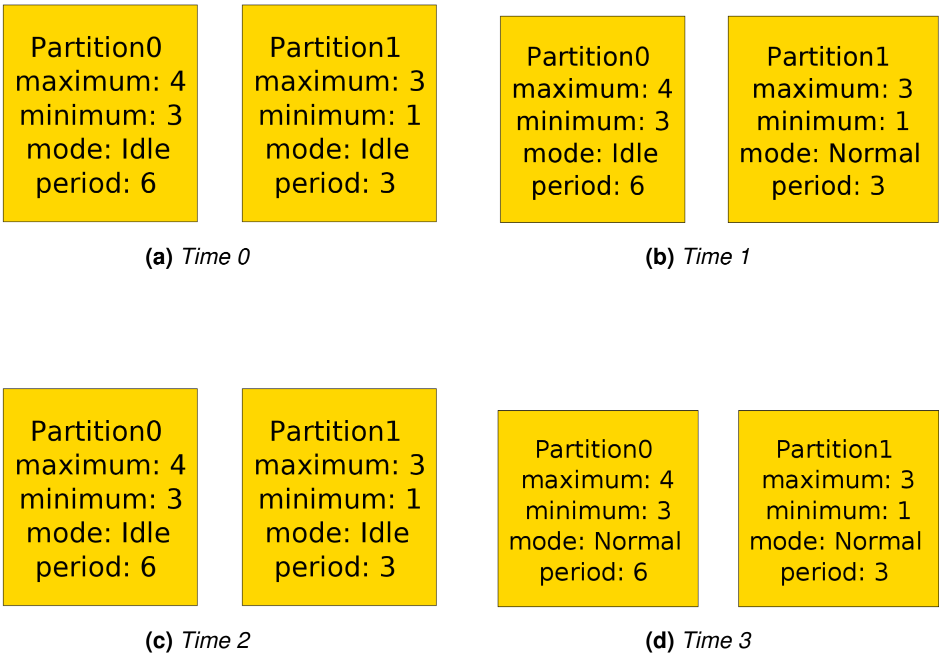


Figure 4.4: *Partition Model*

This model has two different characteristics from the one in Figure 4.2. First, there is a new signature *Mode* with two sub-signatures *Idle* and *Normal*, then we clearly see that now each partition mode is mapped into a atom of the ordered signature *Time*. At each model instance, will be possible to see how the partitions modes are changing over time. Figure 4.5 shows us two partitions changing over time (this kind of visualization can be achieved by projecting signature *Time* in Alloy):



(a) *Time 0*

(b) *Time 1*

(c) *Time 2*

(d) *Time 3*

Figure 4.5: *Partitions Evolution over Time*

In Figure 4.5 shows two partitions where all attributes are fixed in every instant of time (except partitions modes). The next requirement will introduce the rules which control the change of a particular partition mode.

PRT007 - Partition Scheduling Characteristics.

Requirement Number	PRT#007
Title	Partition Scheduling Characteristics
Description	The main characteristics of the partition scheduling model shall be: • The scheduling unit is a partition. • Partitions have no priority. • The scheduling algorithm is predetermined, repetitive with a fixed periodicity, and is configurable by the system configuration only. At least one partition window is allocated to each partition during each cycle. • The core module level O/S exclusively controls the allocation of the resources to the partition.
Rationale	N/A.

This is where the scheduling of partition is specified. This requirement tell us that at this level, partitions have no priority, both periodicity and duration quotas need to be respected and there must always exists a partition scheduled (i.e in the *NORMAL* mode).

What this scheduling requirement tell to the model, is how to partitions can evolve over time. We'll represent this using a fact, which restricts the evolution of the partitions modes. Before defining that fact, lets us define some functions and predicates which will be useful:

```
fun prox : Time → Time {
  ordering/next +last → first
}

fun diff [t,t' : Time] : Int {
  #(t' in (t·nexts +t) ⇒(t·nexts & (t' +t'·prevs)) else (t·nexts +t'·prevs +t))
}

pred i2n [p : Partition, t : Time] {
  p·mode·t = Normal
  p·mode·(t·~prox) = Idle
}

pred n2i [p : Partition, t : Time] {
  p·mode·t = Idle
  p·mode·(t·~prox) = Normal
}
```

The function *prox* will allow us to see time in a circular fashion. This is important because the problem of scheduling must be modeled using infinite traces (the system is supposed to schedule for ever), and allow us to abstract from the special cases of initialization and end of the execution traces.

The function *diff* count how many time instants are between two points in time. This function will be used to respect the requirements of period and duration of partitions. Remember that Alloy doesn't have an explicit representation of time, in order to achieve same behavior of time, we represent it as clicks (much like Leslie Lamport reasons about time [54, 56]).

Predicates *i2n* and *n2i* model the partition mode at a given time instant *t* and guarantee that it changes in the following instant.

With this predicates and functions, the fact which will restrict the scheduling of partitions can now be defined:

```
fact Scheduler{
  all t : Time | lone (mode·t)·Normal
  all p : Partition | some t : Time | p·mode·t = Normal
  all p : Partition, t : Time | i2n[p,t] ⇒
  {
    (some t' : Time | eq[diff[t,t'],p·period] and i2n[p,t']) &&
    (all t' : Time | lte[diff[t,t'],p·minimum] ⇒p·mode·t' = Normal) &&
    (all t' : Time | gt[diff[t,t'],p·maximum] and lt[diff[t,t'], p·period] ⇒p·mode·t' = Idle) &&
    (all t' : Time | gt[diff[t,t'],0] and lt[diff[t,t'],p·period] ⇒not i2n[p,t'])
  }
}
```

The fact start by stating that can't be more than one partition executing at the same time (this is a fair assumption, because requirement doesn't talk about the number of processors, so it's assumed that only one processor is available) and all partition must execute at some point in time.

Then it continues to restrict what happens when a partition is *IDLE* and will execute in the following instance. To guarantee a correct scheduling of these partitions a number of conditions must verify:

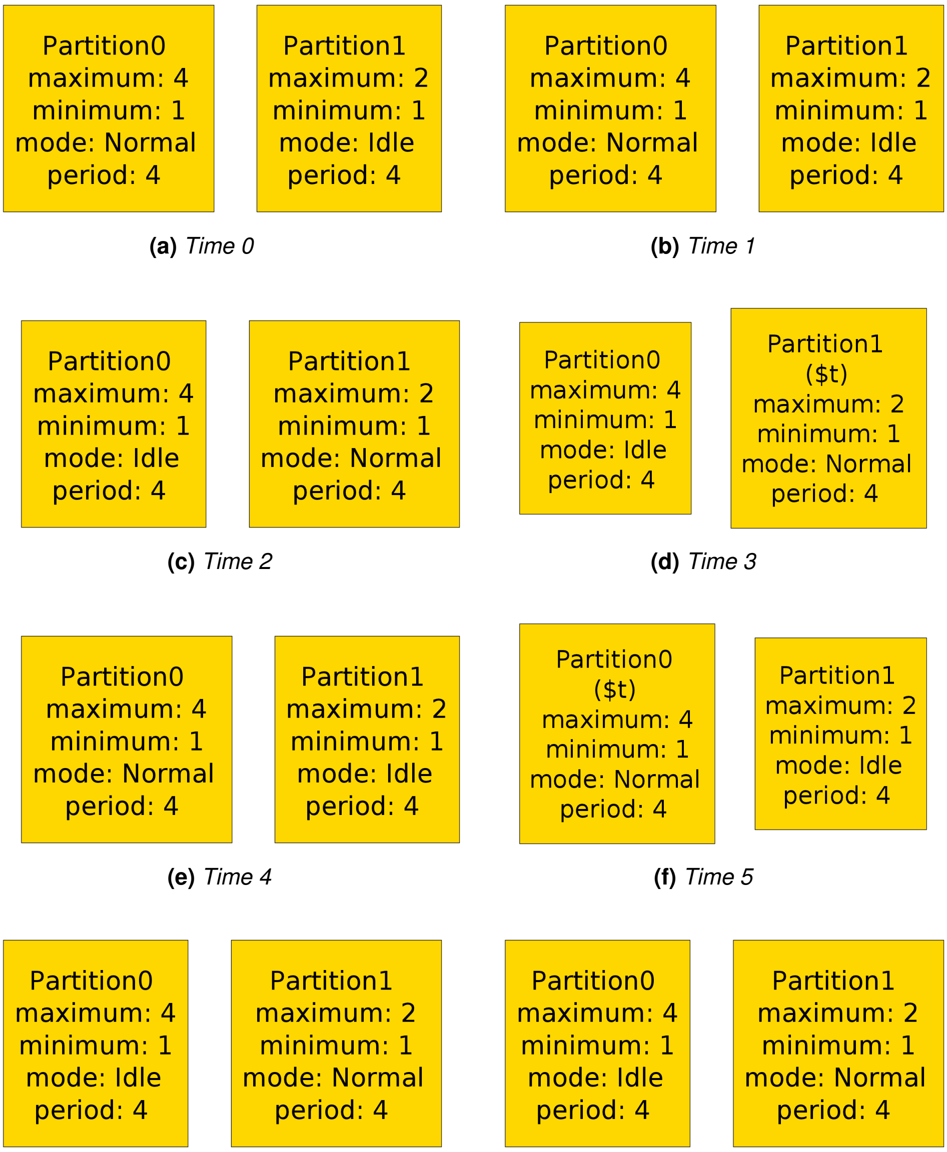
1. The *period* condition of the partition must verify: The difference between the current time and the last time the partition executed must be equal to the partitions period: **some t' : Time | eq[diff[t,t'],p.period] and i2n[p,t']**

2. The *partition* execution time must always be higher or equal to the minimum duration : **all t' : Time | lte[diff[t,t'],p.minimum] => p.mode.t' = Normal**

3. If the partition is running and its maximum duration time has bean achieved, it must be in the *IDLE* mode: **all t' : Time | gt[diff[t,t'],p.maximum] and lt[diff[t,t'], p.period] => p.mode.t' = Idle**.

4. At all points in time, a partition cannot go to a *NORMAL* state if its period hasn't been respected: **all t' : Time | gt[diff[t,t'],0] and lt[diff[t,t'],p.period] => not i2n[p,t']**

The structure of the model (metamodel) remains equal to the one in Figure 4.4 because this requirement doesn't introduced any structural changes in signatures. Figure 4.6 shows an instance where two partitions are correctly scheduled according to this requirement:



(a) *Time 0*

(b) *Time 1*

(c) *Time 2*

(d) *Time 3*

(e) *Time 4*

(f) *Time 5*

(g) *Time 6*

(h) *Time 7*

Figure 4.6: *Partitions Evolution over Time*

In this instance, two partitions are being scheduled accordingly to their attributes. Figure 4.6a shows us that at the time instant 0, *Partition 0* is in the *Normal* mode and *Partition 1* is in *Idle* mode. *Partition 0* cannot be more than 4 instances in the *Normal* mode (the *minimum* attribute has value 4). Figure 4.6c shows that at instant 2 *Partition 0* goes *Idle* and *Partition 1* goes *Normal*. Because *Partition 1* has the *maximum* attribute with value 2, it cannot stay in *Normal* mode more than two instants. In Figure 4.6f shows us that *Partition 1* changes its mode to *Idle* at instant 5, respecting its *maximum* attribute. In further instants, partitions continue to be scheduled, always respecting its real-time values.

PRC004 - Processes Attributes.

Requirement Number	PRC#004
Title	Processes Attributes
Description	A set of unique attributes shall be defined for each process within the system. These attributes differentiate between unique characteristics of each process as well as define resource allocation requirements. The following attributes a process shall have: • Base Priority - Denotes the capability of the process to manipulate other processes. • Period - Identifies the period of activation for a periodic process. A distinct and unique value should be specified to designate the process as aperiodic. • Time Capacity - Defines the elapsed time within which the process should complete its execution. • Current Priority - Defines the priority with which the process may access and receive resources. It is set to base priority at initialization time and is dynamic at runtime. • Process State - Identifies the current scheduling state of the process. The state of the process could be either dormant, ready, running or waiting.
Rationale	N/A.

This requirement introduces the second most important concept in a partitioning kernel: the process. It states which attributes the process must have and give us some information about them. Process attributes are modeled in the following signature:

```
sig State {}
sig Ready, Running extends State {}

sig Process{
  prt : one Partition,
  p_period : one Int,
  time_capacity : one Int,
  base_priority : one Int,
  curr_priority : Int one → Time,
  state : State one → Time,
}
```

The state of a process was modeled as the state of partitions: every process as a *Ready* or *Running* state at some point in time. This is again the application of the local trace idiom presented in Section 2.1.1. A new attribute was added to connect processes and partitions. This attribute is justified by our common knowledge of the partitioning kernel architecture which is reflected in Figure 4.1, and tell us that each process is associated with a partition.

The attributes of processes must be scoped and restricted in order to reflect what the requirement is telling us:

```
fact ProcessConstraints{
  gt[Process·p_period,0]
  gt[Process·time_capacity,0]
  all p:Process | gt[p·base_priority, 0]
  all p:Process, t:Time | gt[p·curr_priority·t,0]
  all p:Process | eq[p·base_priority, p·curr_priority·first]
  all p:Process | gte[p·p_period,p·time_capacity]
  all p:Process | gt[p·p_period, p·prt·period]
  all p:Partition | some p':Process | p'·prt = p
}
```

Fact *ProcessConstraints* starts by specifying which attributes must have positive values. Then it obligates that for all processes the base priority is equal to the current priority in the initial time instant, following the property that processes periods must be greater than its time capacities (otherwise the process period requirement could not be respected). The last property of the *fact* states that all partitions must have at least one process associated with it.

Using Alloy, we can see the model of the process. Remember that as explained in Section 3.3, the tool which support this requirement analysis method automatically generates the visual model of the requirement, allowing the user to see only what changes the requirement induced. Figure 4.7 shows a visual representation of a *Process* and states *Ready* and *Running*:

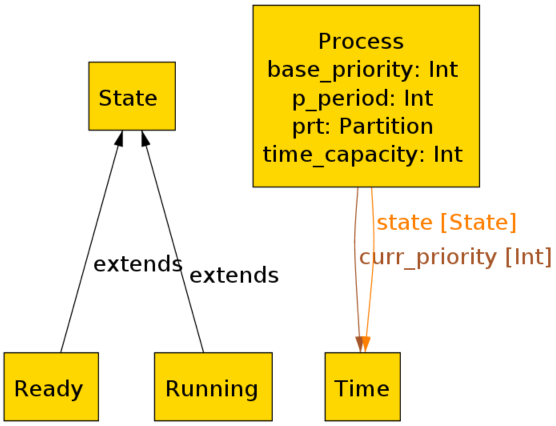


Figure 4.7: *Process Model*

In the following requirement the reader will be able to see model instances of the whole partitioning kernel, containing partitions and processes.

PRC013 - Processes Scheduling Model.

Requirement Number	PRC#013
Title	Processes Scheduling Model
Description	The main characteristics of the scheduling model used at the partition level shall be: • One of the main activities of the O/S is to arbitrate the competition that results in a partition when several processes of the partition each want exclusive control over the processor. • Each process has a priority. • The scheduling algorithm is priority preemptive. If several processes have the same current priority, the O/S selects the oldest one. • Periodic and aperiodic scheduling of processes are both supported. • All the processes within a partition share the resources allocated to the partition.
Rationale	

The scheduler of a partitioning works in two levels, this requirement introduces the restrictions to scheduling processes in a partitioning kernel (the second level). This will inevitably lead to changes where the previous rules for the scheduler are defined: the fact ”*Scheduler*”.

Like we did to partitions, let us start by defining some auxiliary predicates:

```
pred re2ru[p : Process, t:Time]{
  p·state·t = Running
  p·state·(t·~prox) = Ready
}

pred ru2re[p : Process, t:Time]{
  p·state·t = Ready
  p·state·(t·~prox) = Running
}

pred HasMaxPriority[p:Process, t:Time]{
  all p':Process | p'·prt=p·prt ⇒lte[p'·curr_priority·t,p·curr_priority·t]
}
```

The predicates *re2ru* and *ru2re* are similar to predicates *i2n* and *n2i* used to check partitions states. One significant difference between scheduling of partitions and processes, is the fact that processes scheduling must respect the processes priority. To check if a process has top priority within a partition, the *HasMaxPriority* predicate check if a process has the maximum current priority of all processes within a partition at a time instant *t*.

With this auxiliary predicates the fact *Scheduler* can now be re-factored in order to reflect this new requirement:

```
fact Scheduler{
  //Partition
  all t : Time | lone (mode·t)·Normal
  all p : Partition | some t : Time | p·mode·t = Normal
  all p : Partition, t : Time | i2n[p,t] ⇒
  {
    (some t' : Time | eq[diff[t,t'],p·period] and i2n[p,t']) &&
    (all t' : Time | lte[diff[t,t'],p·minimum] ⇒p·mode·t' = Normal) &&
    (all t' : Time | gt[diff[t,t'],p·maximum] and lt[diff[t,t'], p·period] ⇒p·mode·t' = Idle) &&
    (all t' : Time | gt[diff[t,t'],0] and lt[diff[t,t'],p·period] ⇒not i2n[p,t'])
  }

  //Process
  all t: Time | lone p:Process | (p·state·t)=Running
  all p : Process | some t:Time | p·state·t = Running
  all p : Process, t:Time | p·state·t = Running ⇒{
    p·prt·mode·t=Normal && HasMaxPriority[p,t]
  }
  all p : Process, t:Time | re2ru[p,t] ⇒{
    (some t':Time | eq[diff[t,t'],p·p_period] && re2ru[p,t']) &&
    (all t':Time| gt[diff[t,t'],p·time_capacity] && lt[diff[t,t'],p·p_period] ⇒p·state·t' = Ready) &&
    (all t':Time | gt[diff[t,t'],0] && lt[diff[t,t'],p·p_period]⇒ not re2ru[p,t'])
  }
}
```

This fact now has two distinct sections: partitions scheduling (already explained) and the processes scheduling. The explanations here are referring to the process scheduling section of the fact.

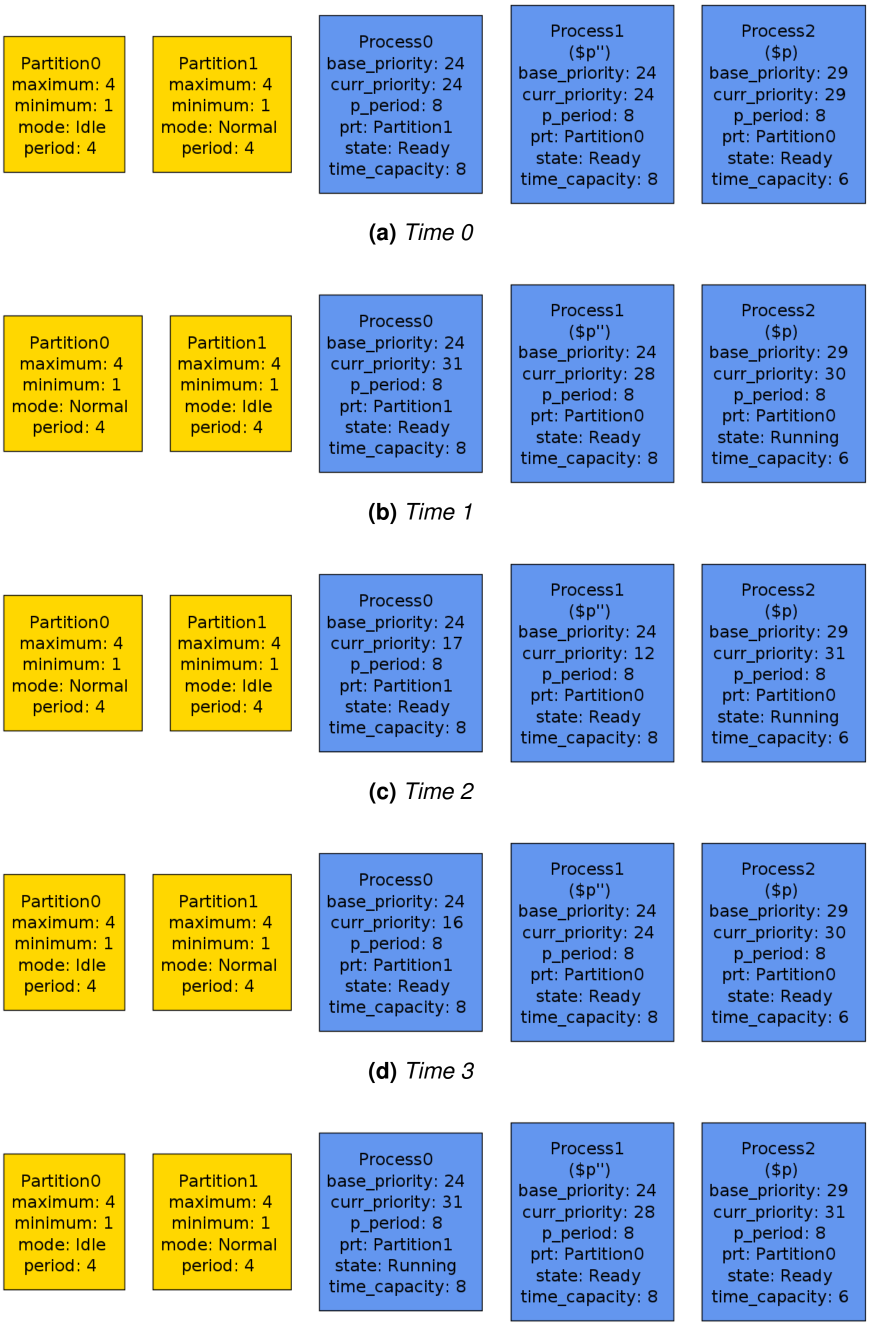
The first two lines states that at some point in time there can be no more than one process executing and all processes will eventually execute. Again, it is assumed that there is only one CPU available and therefore only one process can be in the state *Running*. The following line states that if a process is running then its partition must be in the *Normal* state and the process must have top priority in the whole system. This restriction comes directly from the requirement.

Let us see in detail the rules which must verify in order to a process pass from a *Ready* to *Running* state:

1. The period of the process must be respected, i.e. the difference of the present time and the last time the process was executing must be equal to the process period: **some t':Time | eq[diff[t,t'],p.p_period] && re2ru[p,t']**.

2. The process must be in a ready state if its period time has not pass and if the time capacity of the process has exceeded: **all t':Time| gt[diff[t,t'],p.time_capacity] && lt[diff[t,t'],p.p_period] => p.state.t' = Ready**.

3. If the period of the process is not respected at time instant *t'*, the process cannot be in a running state in the next time instant: **all t':Time | gt[diff[t,t'],0] && lt[diff[t,t'],p.p_period]=> not re2ru[p,t']**.



(a) *Time 0*

(b) *Time 1*

(c) *Time 2*

(d) *Time 3*

(e) *Time 4*

Figure 4.8: *Partitions Evolution over Time*

At this point we are able to see how the model respects all constraints in the model and tries to schedule the process with the higher current priority inside the scheduled partition. Figure 4.8 shows us two partitions and three processes. Process 0 belongs to partition 1 and processes 1 and 2 belong to partition 0. The scheduler schedules partition 0 at instant 1 and the process with the higher priority inside it: process 2. Process 2 continues to be scheduled at instant 2 and at instant 3 the scheduler decides to put the process 2 into *Ready* state to schedule Process 0 at the instant 4. This is a demonstration of how Alloy can be used to correctly model a partitioning kernel scheduler with some real-time constraints.

The reader can find the whole execution of these three processes and two partitions at Section A.1 of the Appendix.

4.3 Summary

This Chapter showed how the method presented in Chapter 3 can be effectively used into requirements documents of LCS. We've seen how writting a model imediatly after the appearance of the requirement, can improve readability and comprehension of its formal model. This readability improvement is gained mainly due the fact that we don't have to write the complete model but only a portion of it, justifying with the requirement the appearance of each model.

This Case Study showed the beginning of a process of requirements traceability, obligating the requirement engineer to write every model for all requirements considered. This approach is completely different from creating one model for the whole requirement, which is against the principles of requirement traceability.

The following Chapter will present an evolution from what we've seen so far. Chapter 3 still calls for an expert to create formal models for each requirement. Below we will see that there is a correspondence between common ways of describe requirements and formal models. This correspondence will allow requirements engineers to have *free* formal models for theirs requirements, without having to write a single specification into some formal language.

Chapter 5

From Boilerplated Requirements to Abstract Models

5.1 Introduction

Today requirements for software are usually written using NL and that won't change in the near future, do to the many advantages which NL brings. It is accepted that the gap between requirements and the final code:

$$Requirements \xrightarrow{\hspace{6em}} Code$$

is too wide, despite the many attempts to shorten it and automate the whole process. Research has come to the conclusion that such gap must be splinted in shorter paths. One way to split the gap between *requirements* and *cfode*, is to add an *Abstract Model* between them:

$$Requirements \longrightarrow Abstract\ model \longrightarrow Code$$

Such *Abstract Model* is used to improve the knowledge of the software requirements and to easily the life of programmers when they need to write the code (sometimes the code can be partially generated from such abstract model). Those abstract models can be constructed using *Formal methods* like the ones we've seen in Chapters 1 and 2. In order to construct these models, one must be have abstraction skills and good mathematical background. The problem with the construction of abstract model, is that abstraction is the accepted as the hardest part of programming and the skill which programmers lack most [53].

Programming is usually learn as a repetitive task: one apply common solutions to common problems. That is reflected to requirements descriptions, similar texts patterns are used to describe tasks or functions which the system must have. Such patterns in requirements documents, can be used to standardize and generalize the writing of requirements using *boilerplates* [22]. The use of boilerplates, is another split in the path between *Requirements* and *Code* and it appears immediately after requirements are written and before the abstract model is constructed:

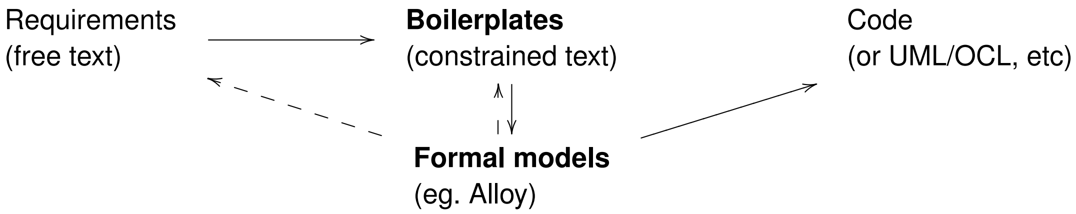
$$Requirements \longrightarrow \textbf{Boilerplates} \longrightarrow Abstract\ model \longrightarrow Code \tag{5.1}$$

But what is a boilerplate? A boilerplate is nothing else than parametric text, with place holders to be filled in. An example of a boilerplate is:

The <**STAKEHOLDER**> *shall be able to* <**CAPABILITY**> (5.2)

The above boilerplate can be used to write requirements which describe that some entity shall have a certain capability. Using boilerplates as a standard to write requirements descriptions, boilerplates are classified according to their characteristics and then use that classification to group requirements which have the same structure [22].

Boilerplates can be used to shorten the gap between requirements and their abstract models, reliving programmers from having sophisticated abstract skills. However there are a number of challenges to be addressed in order to use boilerplates effectively in this manner. The alternative which is followed in the following sections, is to find requirement boilerplates not in text, but in formal models:



Requirements
(free text)

Boilerplates
(constrained text)

Code
(or UML/OCL, etc)

Formal models
(eg. Alloy)

We'll see how to gather boilerplates from common patterns in modeling languages. This method of finding the boilerplates, requires expertise in formal methods and requirement engineer, but once a boilerplate is found we can added it a boilerplate repository and we have three guarantees:

- A new boilerplate which can be used to write requirements in a standard manner.
- We can assume that the boilerplate has a subsequent abstract model, which has been proved useful.
- A free abstract model, retrieved by construction.

Alloy was adopted as a formal language to construct the abstract models. Using Alloy, allow us to incorporate this use of boilerplates in the methods described in Chapter 3. Also, Alloy language is highly invites the user to model using the composition operator, which is closer to the way sentences appear in NL.

5.2 Boilerplates meets Alloy

Let us consider the following requirement concerning a Partition Information Flow Policy (PIFP) of a SPK addressed in Chapter 4:

Req. PIFP#009: *Partitions shall have access to channels via defined access points called ports.*

If we look at the boilerplate repository:

http://freespace.virgin.net/gbjedi/books/re/boilerplates/repository.htm

it's clear that doesn't exist any boilerplate which is applicable for this requirement. Let us start by defining our Alloy model of such requirement:

```
sig Channel{}
sig Partition{ port: one Channel }
```

Looking at the model of the requirement, we can infer the following boilerplate:

A <**Partition**> *is linked to one and only one* <**Channel**> *through a* <**Port**> (5.3)

This boilerplate has a subsequent formal model associated within and can be used in any other problem domain. Boilerplate 5.3 can be used in requirements concerning keyboards and computers:

A <**Keyboard**> *is linked to one and only one* <**Computer**> *through a* <**USB-Cable**> (5.4)

Boilerplate 5.3 is nothing else than a manner of stating that one entity is related by only one other entity, through some device. Boilerplate 5.3 can be generalized as:

A <**Entity1**> *is linked to one and only one* <**Entity2**> *through a* <**Device**> (5.5)

Technically, what we did was to instance boilerplate 5.5 placeholders with different values. In the first under substitutions:

<**Entity1**> := <**Partition**>,
<**Entity2**> := <**Channel**>,
<**Device**> := <**Port**>

and the latter under substitutions

<**Entity1**> := <**Keyboard**>,
<**Entity2**> := <**Computer**>,
<**Device**> := <**USB-Cable**>.

The generic boilerplate 5.5 results in the Alloy model:

```
sig Entity2{}
sig Entity1{ device: one Entity2 }
```

It is possible that two completely different boilerplates, generate the same *abstract model*. Consider a requirement concerning a Configuration Management System (CMS) of the SPK addressed in Chapter 4:

Req. CMS#001: *The system shall have a Configuration Management System (CMS)*

An appropriate boilerplate for this requirement could be:

A <**System**> *shall have a SubSystem* (5.6)

The corresponding Alloy model for this boilerplate is similar to the one of boilerplate 5.5:

```
sig SubSystem{}
sig System{ subsystem: one SubSystem }
```

In Verified Software Repository (VSR) [1], we can find a set of commonly occurring patterns in formal specification. Such repositories like VSR can be used to construct a repository of boilerplates and then be applied into requirement from different problem domains. In the following section, will be shown a set of rules which specify the translation from boilerplates to Alloy.

5.3 Boilerplates Repository

Boilerplates can be either structural or behavioral. Structural boilerplates are the ones which declares entities and relations between them. The boilerplates we've seen so far, are structural boilerplates. Behavioral boilerplates define how entities evolve over time, and usually introduces actions and restrictions on the model.

Structural boilerplates follow the grammar:

$$
\begin{aligned}
\mathit{structural} &::= \texttt{every}\ \mathit{entity}\ \texttt{shall have}\ [\mathit{mult}]\ [\texttt{fixed}]\ [\mathit{attribute}]\ \mathit{entity}\\
&\quad\mid \mathit{quantifier}\ \mathit{entity}\ \texttt{shall contain}\ [\mathit{mult}]\ [\texttt{fixed}]\ [\mathit{attribute}]\ \mathit{entity}\\
&\quad\mid \mathit{quantifier}\ \mathit{entity}\ \texttt{is a}\ \mathit{entity}\\
&\quad\mid \mathit{quantifier}\ \mathit{entity}\ \texttt{shall be able to}\ \mathit{action}\ \mathit{entity}\\
\mathit{mult} &::= \texttt{one} \mid \texttt{at most one} \mid \texttt{some}\\
\mathit{entity} &::= \mathit{noun}, \mathit{attribute} ::= \mathit{adjective} \mid \mathit{noun}, \mathit{action} ::= \mathit{verb}
\end{aligned}
$$

The structural boilerplates which one may derive from the grammar above, has the following relationships: *association* (`shall have`); *composition* (`shall contain`); *generalization* (`is a`). Associations and compositions can optionally be declared as immutable (`fixed`), and be given an explicit attribute name and multiplicity. In addition to this relationships, the grammar allows one to define boilerplates with the notion of *capability* (`shall be able to`), meaning that an entity can act somehow on another entity.

This set of boilerplates have a direct correspondence with Alloy model. In order to translate structural boilerplates to Alloy, one should use the following rules:

$$
\begin{aligned}
[\![\texttt{every}\ e_1\ \texttt{shall have}\ m\ \texttt{fixed}\ a\ e_2]\!] &\equiv \texttt{sig}\ e_1\ \{\ a : [\![m]\!]\ e_2\ \}\\
&\ \texttt{sig}\ e_2\ \{\ \}\\
[\![\texttt{every}\ e_1\ \texttt{shall have}\ m\ a\ e_2]\!] &\equiv \texttt{sig}\ e_1\ \{\ a : e_2\ [\![m]\!] \rightarrow \texttt{Time}\}\\
&\ \texttt{sig}\ e_2\ \{\ \}\\
[\![\texttt{every}\ e_1\ \texttt{shall contain}\ m\ \texttt{fixed}\ a\ e_2]\!] &\equiv \texttt{sig}\ e_1\ \{\ a : [\![m]\!]\ e_2\ \}\\
&\ \texttt{sig}\ e_2\ \{\ e_1 : \texttt{lone}\ e_1\ \}\\
&\ \texttt{fact}\ \{\ e_1 = \texttt{\~{}}e_2\ \}
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{every}\ e_1\ \texttt{shall contain}\ m\ a\ e_2]\!] &\equiv \texttt{sig}\ e_1\ \{\ a : e_2\ [\![m]\!] \rightarrow \texttt{Time}\}\\
&\ \texttt{sig}\ e_2\ \{\ e_1 : e_1\ \texttt{lone} \rightarrow \texttt{Time}\}\\
&\ \texttt{fact}\ \{\ \texttt{all t : Time} \mid e_1.\texttt{t} = \texttt{\~{}}(a.\texttt{t})\ \}\\
[\![q\ e_1\ \texttt{is a}\ e_2]\!] &\equiv \texttt{sig}\ e_1\ \texttt{extends}\ e_2\ \{\ \}\\
&\ \texttt{sig}\ e_2\ \{\ \}\\
[\![\texttt{every}\ e_1\ \texttt{shall be able to}\ a\ e_2]\!] &\equiv \texttt{sig}\ e_1\ \{\ a : e_2\ \texttt{lone} \rightarrow \texttt{Time}\}\\
&\ \texttt{sig}\ e_2\ \{\ \}
\end{aligned}
$$

[1] See `http://vsr.sourceforge.net`.

To translate the multiplicities, one just have to replace *at most one* by lone, the other two are equal (one an some). This translation rules only consider boilerplates which impose a name on the relation (like boilerplate 5.3 where *Device* is the name of the relationship). When a boilerplate doesn't define the nome for the relationship, the name of the entity can be used. In boilerplate 5.5 relation *subsystem* was renamed with the name of the entity in lowercases.

Some of these rules are defined using the local state modeling idiom described in Section 2.1.1 which allow us to model in terms of traces of execution (Please See [47]). When a relation is dynamic, an extra column *Time* is added, allowing us to visualize evolution over time.

As an example consider the composition boilerplate:

$$
\begin{aligned}
[\![\texttt{every } e_1 \texttt{ shall contain } m\ a\ e_2]\!] \equiv\ & \texttt{sig } e_1 \ \{\ a : e_2\ [\![m]\!] \rightarrow \texttt{Time}\} \\
& \texttt{sig } e_2 \ \{\ e_1 : e_1 \texttt{ lone} \rightarrow \texttt{Time}\} \\
& \texttt{fact } \{\ \texttt{all t : Time} \mid e_1.\texttt{t} = \sim(a.\texttt{t})\ \}
\end{aligned}
$$

Besides the relation between a component and its part, its added a contained relation from the part to its component with the name of the former. This relation must be simple, i.e. have multiplicity `lone`, since a part cannot be shared between components, and should be symmetric in relation to the contains relation. In this particular case, the composition is mutable and both these relations are extended with `Time` and the aforementioned constraints become invariants over execution traces. As an example of composition consider the following requirement on channels:

$$
\begin{aligned}
& [\![\mathit{Every\ Channel} \texttt{ shall contain } \mathit{Message}\texttt{s}]\!] \equiv \\
& \texttt{sig } \mathit{Channel} \ \{\ \mathit{message} : \mathit{Message} \texttt{ set} \rightarrow \texttt{Time}\ \} \\
& \texttt{sig } \mathit{Message} \ \{\ \mathit{channel} : \mathit{Message} \texttt{ lone} \rightarrow \texttt{Time}\ \} \\
& \texttt{fact } \{\ \texttt{all t : Time} \mid \mathit{channel}.\texttt{t} = \sim(\mathit{message}.\texttt{t})\ \}
\end{aligned}
$$

Because a channel can contain different messages over time, relations *message* and *channel* are considered mutable and the *Time* column is added to them.

These rules can be applied to translate boilerplate individually. In requirements documents, entities are mentioned more than once and if we use boilerplates to write requirements, the same entity will appear in different boilerplates. This raises the challenge of combining different Alloy models from different boilerplates, in order to gather in the same model all that has been said about an entity or set of entities. The case of adding new information to an entity and reflect that in a final Alloy model, can be solved using a *coalesced sum* operator :

$$
\begin{pmatrix}
\textit{sig}\ \texttt{B \{\}} \\
\textit{sig}\ \texttt{A \{ rs1: B \}}
\end{pmatrix}
\oplus
\begin{pmatrix}
\textit{sig}\ \texttt{C \{\}} \\
\textit{sig}\ \texttt{A \{ rs2: C \}}
\end{pmatrix}
=
\begin{pmatrix}
\textit{sig}\ \texttt{B \{\}} \\
\textit{sig}\ \texttt{C \{\}} \\
\textit{sig}\ \texttt{A \{ rs1: B,} \\
\qquad \texttt{rs2: C} \\
\quad \texttt{\}}
\end{pmatrix}
$$

This operation is well know in relation algebra known as *"split"* [6] or *"fork"* [26] and its presented in Section 2.1.2. Another case arises when we instantiate the same boilerplate and change only some entities. For example, suppose boilerplate 5.3 is instantiated as:

$$\textit{A} <\textbf{A}> \textit{is linked to one and only one} <\textbf{B}> \textit{through a} <\textbf{D}> \tag{5.7}$$

*A <***A***> is linked to one and only one <***C***> through a <***D***>* (5.8)

If the above boilerplates were translated to Alloy, they would originate similar models. The former would be:

```
sig B{}
sig A{ D: one B }
```

and later:

```
sig C{}
sig A{ D: one C }
```

Those models need to be combined and reflect both information in only one signature *A*. This leads to the following rule:

$$\left(\begin{array}{l}\texttt{sig B \{\}}\\ \texttt{sig A \{}\\ \quad\texttt{D: B}\\ \texttt{\}}\end{array}\right) \oplus \left(\begin{array}{l}\texttt{sig C \{\}}\\ \texttt{sig A \{}\\ \quad\texttt{D: C}\\ \texttt{\}}\end{array}\right) = \left(\begin{array}{l}\texttt{sig B \{\}}\\ \texttt{sig C \{\}}\\ \texttt{sig A \{}\\ \quad\texttt{D: B} + \texttt{C}\\ \texttt{\}}\end{array}\right)$$

This operation is nothing else then the *Either* operator described in Section 2.1.2.

Behavioral boilerplates are inspired by the navigational style of Alloy, where constraints enforce cardinality or inclusion of sets resulting from applying the relational composition operator defined in Section 2.1.2. This navigational style is captured in English by the use of possessive form, which lead to the behavioral boilerplates described in the following grammar:

$$\begin{aligned}
\mathit{behavioral} &::= \mathit{lset}\ \texttt{shall}\ [\texttt{not}]\ \texttt{be}\ ([\texttt{in}]\ \texttt{the}\ \mathit{rset} \mid \texttt{empty})\\
\mathit{lset} &::= \texttt{every}\ (\mathit{entity} \mid \mathit{attribute})\ \{\mathit{attribute}\}\\
&\mid \texttt{the}\ \mathit{attribute}\ \{\mathit{attribute}\}\ \texttt{of the}\ \mathit{lset}\\
\mathit{rset} &::= \texttt{the}\ (\mathit{entity} \mid \mathit{attribute})\ \{\mathit{attribute}\}\\
&\mid \texttt{the}\ \mathit{attribute}\ \{\mathit{attribute}\}\ \texttt{of the}\ \mathit{rset}\\
\mathit{entity} &::= \mathit{noun},\ \mathit{attribute} ::= \mathit{adjective} \mid \mathit{noun} \mid \mathit{verb}
\end{aligned}$$

Possessive forms can be constructed both with the possessive apostrophe or preposition `of`. To improve readability, we can write an attribute which is then followed by its target entity. This behavioral boilerplates can be translated to Alloy using the following rules:

$$
\begin{aligned}
\llbracket l\ \texttt{shall be}\ r \rrbracket &\equiv \ \texttt{fact}\ \{\ \texttt{all t : Time},\ \llbracket l \rrbracket^{\epsilon} = \llbracket r \rrbracket \} \\
\llbracket l\ \texttt{shall be in}\ r \rrbracket &\equiv \ \texttt{fact}\ \{\ \texttt{all t : Time},\ \llbracket l \rrbracket^{\epsilon}\ \texttt{in}\ \llbracket r \rrbracket \} \\
\llbracket l\ \texttt{shall not be}\ r \rrbracket &\equiv \ \texttt{fact}\ \{\ \texttt{all t : Time},\ \llbracket l \rrbracket^{\epsilon}\ ! = \llbracket r \rrbracket \} \\
\llbracket l\ \texttt{shall not be in}\ r \rrbracket &\equiv \ \texttt{fact}\ \{\ \texttt{all t : Time},\ \llbracket l \rrbracket^{\epsilon}\ \texttt{not in}\ \llbracket r \rrbracket \} \\
\llbracket l\ \texttt{shall be empty} \rrbracket &\equiv \ \texttt{fact}\ \{\ \texttt{all t : Time},\ \llbracket l \rrbracket^{\texttt{no}} \} \\
\llbracket l\ \texttt{shall not be empty} \rrbracket &\equiv \ \texttt{fact}\ \{\ \texttt{all t : Time},\ \llbracket l \rrbracket^{\texttt{some}} \} \\
\llbracket \texttt{the}\ a_1\ \ldots\ a_l\ \texttt{of every}\ l \rrbracket^{m} &\equiv \ \llbracket \texttt{every}\ l\ a_1\ \ldots\ a_l \rrbracket^{m} \\
\llbracket \texttt{the}\ b_1\ \ldots\ b_n\ \texttt{of the}\ r \rrbracket &\equiv \ \llbracket \texttt{the}\ r\ b_1\ \ldots\ b_n \rrbracket \\
\llbracket \texttt{every}\ e\ a_1\ \ldots\ a_l \rrbracket^{m} &\equiv \ x : e\ ,\ y : x.\llbracket a_1 \rrbracket.\ldots.\llbracket a_{l-1} \rrbracket \mid m\ y.\llbracket a_l \rrbracket \\
\llbracket \texttt{every}\ a\ a_1\ \ldots\ a_l \rrbracket^{m} &\equiv \ y : \texttt{univ}.\llbracket a \rrbracket.\llbracket a_1 \rrbracket.\ldots.\llbracket a_{l-1} \rrbracket \mid m\ y.\llbracket a_l \rrbracket \\
\llbracket \texttt{the}\ e\ b_1\ \ldots\ b_r \rrbracket &\equiv \ x.\llbracket b_1 \rrbracket.\ldots.\llbracket b_r \rrbracket \\
\llbracket \texttt{the}\ b\ b_1\ \ldots\ b_r \rrbracket &\equiv \ \texttt{univ}.\llbracket b \rrbracket.\llbracket b_1 \rrbracket.\ldots.\llbracket b_r \rrbracket \\
\llbracket a \rrbracket &\equiv \begin{cases} a & \text{if } a \text{ immutable} \\ a.\texttt{t} & \text{otherwise} \end{cases}
\end{aligned}
$$

In this translation only the definition of *invariant* properties is considered, which justifies why all facts start with quantification over *Time*. Mutable relations are mapped into *Time* events. When behavioral boilerplates are constructed with the word *of*, they are reduced to the case of attributes separated by the possessive apostrophe. If the Left Hand Side (LHS) starts with an *Entity*, a new universally quantified variable x is introduced to be reused in the translation of the Right Hand Side (RHS) if the entity is again referred to. Such an example is the requirement:

$$
\begin{aligned}
&\llbracket \texttt{The}\ \mathit{destination}\ \texttt{of every}\ \mathit{Channel}'\texttt{s}\ \mathit{message}\texttt{s shall be} \\
&\qquad \texttt{the}\ \mathit{partition}\ \texttt{of the}\ \mathit{Channel}'\texttt{s}\ \mathit{destination} \rrbracket \equiv \\
&\texttt{all t : Time} \mid \texttt{all x :}\ \mathit{Channel},\ \texttt{y : x.}(\mathit{message}.\texttt{t}) \mid \\
&\qquad \texttt{y.}\mathit{destination} = \texttt{x.}\mathit{destination.partition}
\end{aligned}
$$

If the LHS does not begin with an entity, the relation is composed with the universal set $univ$ and saved in a universally quantified variable x. This variable is used forward to apply the multiplicity test. An example of such requirement is:

$$
\begin{aligned}
&\llbracket \texttt{The}\ \mathit{channel}\ \texttt{of every}\ \mathit{sent\ Message}\ \texttt{shall be empty} \rrbracket \equiv \\
&\texttt{all t : Time} \mid \texttt{all x : univ.}(\mathit{send}.\texttt{t}) \mid \texttt{no x.}(\mathit{channel}.\texttt{t})
\end{aligned}
$$

Boilerplates repositories like this one, can be used to construct tools which automatically translates requirement boilerplates to abstract model in Alloy. In [10] a sketch of the tool *PROVA* is presented. *PROVA* is a tool that automatically generates Alloy models from the boilerplates present in presented repository, enabling early detection of ambiguities and inconsistencies through model checking. *PROVA* receives requirements in boilerplates generated by the above grammars and then using the translation rules, reaches to the correct Alloy model of each boilerplate instantiation.

5.4 Summary

This Chapter shown that there is a shorter gap between requirements and theirs specification. Boilerplates shorten this gap and present an opportunity for requirements engineers to model their requirements without knowing a formal language. This is an evolution from what we saw in

Chapter 3, where requirements engineers still need to know formal languages and have expertise on the field.

Such as in requirements descriptions, formal models also present commonly used patterns. Identifying this patterns and mapping them to boilerplates, allow one to construct a repository of boilerplates and formal models. These repositories are independent and can be reused in other requirements documents.

This Chapter presented an initial construction for an algebra of boilerplates. This algebra will allow one to combine different boilerplates and models and throught that create complete models for requirements documents.

Chapter 6

From Alloy to free Uppaal models

6.1 Introduction

Alloy it is a useful tool to express behavior and complex constraints for systems without explicit real-time requirements. Sometimes, we can even model with time constraints using idioms like the one explained in Section 2.1.1, allowing the user to express some properties related with time events using the powerful abstraction characteristic of Alloy. However, in some systems, real-time properties can't be expressed with Alloy and this lack of real-time supports requires us to change our models to tools which present better support for real-time and concurrent systems.

We could try to introduce time in Alloy because technical Alloy is a SAT based model-finder, and several authors proposed solutions to encode time in this kind of tools [13, 77]. Another approach would be to encode Lamport's temporal logic [54] in Alloy and through that specify properties with time constraints associated. Both of these approaches lack of the possibility of representing real-time constrains and check systems with those properties (they are focused on specifying casual constraints of time and not on the real-time issues). Another disadvantage is that we would have to re-create another , supporting those methods.

What if there is a connection between Alloy and a tool which efficiently verifies real-time systems? The Uppaal model checker presented in Section 2.2 is an example of such a great tool. If one wants to use these tools together, would probably follow the following process: start to model a system in Alloy and at some point, where the requirements don't allow us to go any further with the Alloy specification (because they impose some real-time constraints, hard to specify in Alloy), we would have to re-create the whole model in the Uppaal idiom. Alloy and Uppaal are tools with different logics for specify models, Alloy comes from the culture Z, Object-Z and Object Oriented Programming and Uppaal uses networks of timed-automata to specify concurrent and real-time systems. This re-creation of the model would be a highly error prone and an arduous task. Ideally we would like to somehow use the Alloy specification to generate our Uppaal model, with the less user intervention possible.

Suppose one wants to model a set of requirements, rich in behavior constraints but also with some real-time impositions. If one had to choose between Alloy and Uppaal, a non Alloy user would probably choose Uppaal right away because it perfectly support real-time and the requirements have real-time constraints. However, if the user has already crated a model in Alloy he would certainly choose it again to model the requirements, using the far more powerful abstraction of Alloy, saving many hours of work. The problem with the second type of users, is that at some point they just can't go any further or will develop a model which doesn't perfectly reflect the requirements. If they can't go any further with Alloy, they must re-crate the model in Uppaal based on what they learn with the Alloy model.

Because Alloy language is based on Object-Z notation, we could try to use the work presented in [19, 20, 18, 21]. The authors presented a technique called *OZTA* which is described as "An integrated an integrated formal modeling technique for specifying real-time complex systems, which combines Object and Timed Automata.". The authors identified a set of commonly used timed automata patterns to specify real-time systems and added them to the Object-Z notation. This work shows us that a connection between Alloy and timed automata can be built, but obligates the user to model systems using only those identified patterns.

In the following Sections will be shown a method and tool which allows the user to generate a Uppaal model from an Alloy model. This method was implemented directly in the GUI and creates a Uppaal automata from a Alloy model. The method allows the user to specify real-time problems in an interactive fashion and releases him of having to be constrained to a set of timed automata patterns.

6.2 Methodology

Apparently Alloy and Uppaal models are in two completely different modeling idioms, but we will see a re-factor which can be applied to every Alloy model, which makes a connection between both idioms of specification.

Suppose one needed to transform a Alloy model into a Uppaal one. The problem which arises is: given a Alloy model of a system, how can be generated a correspondent automata which have the same system states and respects the allowed transitions expressed in Alloy?

Declaring a signature is essentially declaring a set of items which are populating when we run a certain command. Alloy has several modeling idioms, one highly used is the Trace idiom [47]. This idiom allow us to see the evolution of atoms from a particular signature. The idea is to create a ordering on the signature and then allow the atoms to evolute by the application of the predicates defined in the model.

As an example let us use model of the known river crossing puzzle: A farmer needs to bring a wolf, a goat and a cabbage from the right side of the river to the left side. The boat can only carry the farmer and other passenger at a time. If the farmer leaves the wolf with the goat, wolf eats the goat. If the farmer leaves the goat with the cabbage, the goat will eat the cabbage. We need to find a series of steps which the farmer must takes in order to transport all three without they eating each other. The following model is a solution for this puzzle (the model was inspired by the one which comes with Alloy):

Alloy Code Display 1 River Crossing Model. Model adapted from river crossing model example which comes with Alloy.

```
open util/ordering[State]

abstract sig Object { eats: set Object }
one sig Farmer, Cabbage, Goat, Wolf extends Object {}

fact eating { eats = Wolf→ Goat + Goat→ Cabbage }

sig State {
   right: set Object,
   left: set Object
}

fact initialState { first·right = Object && no first·left }

pred crossRiver [from, from', to, to': set Object] {
  // either the Farmer takes no items
  ( from' = from - Farmer && to' = to - to·eats + Farmer ) ||
  // or the Farmer takes one item
  (some item: from - Farmer {
    from' = from - Farmer - item
    to' = to - to·eats + Farmer + item
  })
}

fact stateTransition {
  all s: State, s': next[s] { Farmer in s·right ⇒
      crossRiver[s·right, s'·right, s·left, s'·left] else
      crossRiver[s·left, s'·left, s·right, s'·right]
  }
}

fact{ last·left = Object }
```

In this model the state of the river is encoded into as a order. The evolution of the element in this order is constrained by the fact *stateTransition* which states that can cross the river if the predicate *crossRiver* can be evaluated as true. The predicate *crossRiver* changes things (cabbage, wolf and goat) only if there is no possibility of two things that eat each other can stay in the same side (relation eats is defined in fact *eating*).

This puzzle is modeled using the Trace idiom. Each instance of this model is called an *Execution Trace*. The execution traces of this model will show us possible solutions to the problem, i.e, traces where the state starts by having all things in the right side and end by having all things on the left side of the river.

Each instance of the model is a valid evolution of the signature in the ordering (in this case of signature *State*). When we run the command *run for 8 State*, Alloygives us instances of the model with at most 8 *States*. We can get several of this instances by pressing the button *Next* in the Alloy GUIIf we want instances with more than 8 states, just need to change it in the command.

The Trace idiom doesn't gives us the automata of the problem, but gives us something closer to it: a series of words from a language of that automata (each instance of the model is a valid word of that language). Even if the language is unknown, Alloycan obtain several words from that language.

As an example suppose we have the following instances retrieved through Alloy:

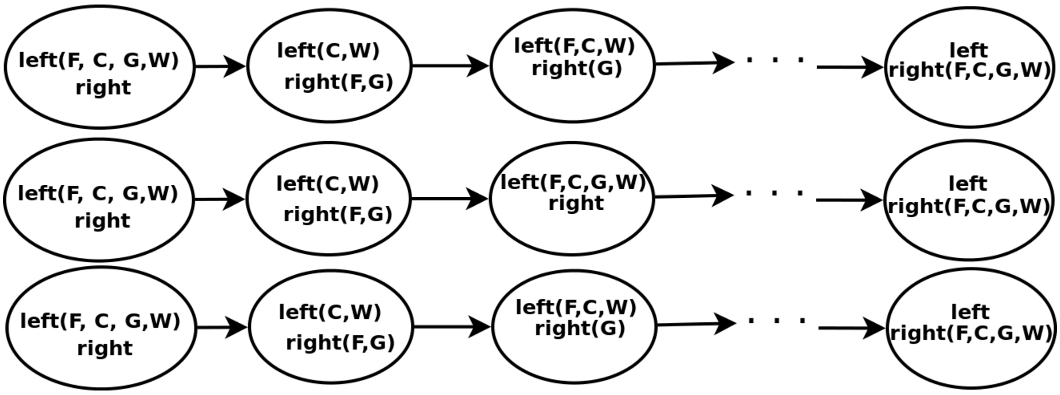


Figure 6.1: *Traces of the River Crossing Puzzle*

Each circle represents the state configuration: right and left margins of the river and where is each thing (F=Farmer, C=Cabbage, G=Goat, W=Wolf). Figure 6.1 shows how the margins evolves in different situations. If we had a Deterministic Finite Automaton (DFA) with the possible states of the river, this automaton would accept each of this traces (assuming the Alloy model is correct). Would be useful if one could automatically generate a DFA that at least accept a set of this traces. With Alloy one could continually ask for more instances of the system (more within a fixed set of river states but also more with a larger number of river states), and thus enrich the language the DFA accepts.

Let us carefully look at the definition of a DFA to see if we can transform instances like the ones presented in figure 6.1 into some DFA:

Definition 1 *A Deterministic finite automaton A consists of:*

- *A finite set Q of states.*

- *A finite set Σ of input symbols.*

- *A transition function $\delta : Q \times (\Sigma \cup \varepsilon) \to Q$, i.e. the transition function δ takes a state $p \in Q$, a symbol $a \ \in (\Sigma \cup \varepsilon)$ and returns a state $\delta(p, a) = q$ such that $q \in Q$. The empty string ε cannot be a member of the alphabet Σ.*

- *One start state $q_0 \in Q$.*

- *A set of final states $F \subseteq Q$.*

Usually the DFA A is written as $A = (Q, \Sigma, \delta, q_0, F)$.

To construct a DFA$A = (Q, \Sigma, \delta, q_0, F)$ from a set of Alloy instances of model written in the Trace Idiom one should take the following steps:

1. The set Σ is input symbols is equal to the sum of all states configuration given by Alloy.

2. Create a new state q_0 and connect that state with the empty transition ε to all initial states in instances given by Alloy. The set Q of states are now q_0 plus all states give by the Alloy instances.

3. The initial state of A is state q_0.

4. The transition function δ is such that: if $p, q \in Q, a \in \Sigma$ and $\delta(p, a) = q$ then state q in the Alloy instance have the configuration a and p was a previous state of q in that instance.

As an example consider the following set of Alloy instances from some arbitrary model (the name of states follows the same notation used in Figure 6.1: each name represents the relations configuration in that state):

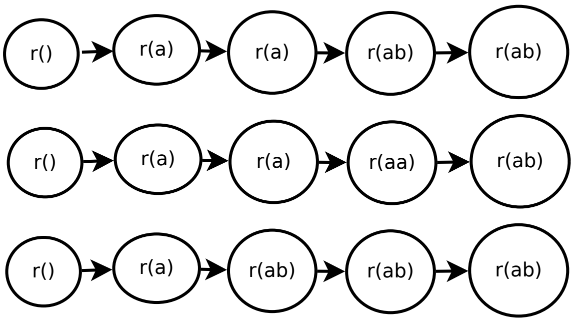


Figure 6.2: *A Set of Alloy instances.*

With these instances we can apply the previous algorithm to obtain a DFA. This DFA accepts the language created by the states configuration provided through Alloy:

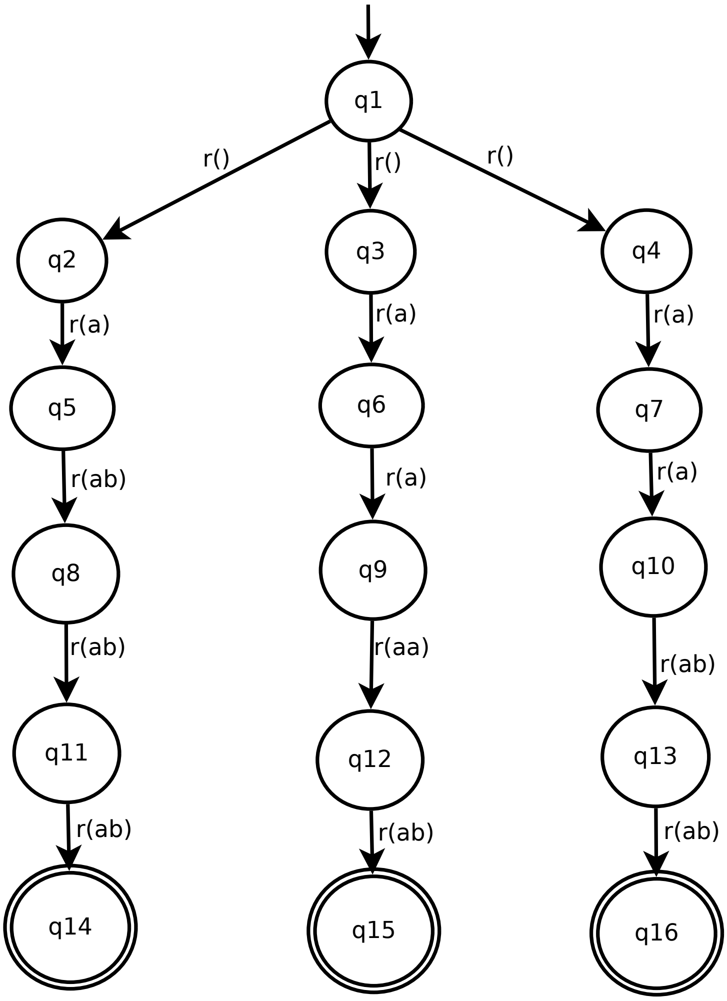


Figure 6.3: *A DFA from a set of instances.*

Although we obtain a DFA which accepts the language induced through Alloy models, this DFA have a lot of states which makes it useless to transfer to a tool like Uppaal. Ideally we want the smallest DFA (smallest in the number of states) which accepts the language induced by Alloy. This can be achieved through the application of a DFA minimization algorithm. The tool described in Section 6.3 implements the DFA minimization algorithm presented in [39].

In order to minimize a DFA we need to find equivalent states and then group them somehow in a new DFA that recognizes the same language. The definition of equivalent states requires the notion of the transition function extended to a word from the alphabet and not only a symbol. This extended function determines where the automaton ends when starting at some state and the input is a sequence of symbols from the alphabet Σ. The definition is of the extended transition function $\hat{\delta}$ is made by induction:

Basis If we are in a state q and read the empty string ε, remain in state q: $\hat{\delta}(q,\varepsilon) = q$.

Induction Let $w \in \Sigma$ be a string of the form xa, where a is the last symbol of w and x is the string consisted of all but the last symbol. The transition function applies recursively: $\hat{\delta}(q,w) = \delta(\hat{\delta}(q,x),a)$.

The idea of the extended transition function is to incrementally compute each symbol with the transition function δ as usual.

We are now ready to test the equivalence of two states in a DFA A.

Definition 2 *Let p, q be two states of a DFAA. States p and q are said to be equivalent if: for all input strings w consisted of symbols from set Σ, $\hat{\delta}(p,w)$ is an accepting state if and only if $\hat{\delta}(q,w)$ is an accepting state.*

From this definition follows that the equivalence of states is recursive:

Theorem 1 *If in some DFA$A = (Q,\Sigma,\delta,q_0,F)$ two states $p,q \in Q$ are equivalent and state $r \in Q$ are equivalent to q, then states p,r are also equivalent.*

The proof of this theorem is straightforward. A detailed proof can be found in [39].

If two states are not equivalent, they said to be distinguishable. In [39] authors present an algorithm, based on this definition of state equivalence, that allows one to find all equivalent states, this algorithm is called **Table-Filling Algorithm** and is implemented in the tool described in Section 6.3.

The *Table-Filling Algorithm* is a recursive discovery of all distinguishable pairs of states in a DFA$A = (Q,\Sigma,\delta,q_0,F)$. After the applying the algorithm, all pairs which are not marked as distinguishable, are equivalent:

Basis For all two states $p,q \in Q$, if p is an accepting state and q is a non-accepting one, the pair p,q is distinguishable.

Induction Let $p,q \in Q$ be states. If for some symbol $a \in \Sigma$ the states $\delta(p,a)$ and $\delta(q,a)$ are distinguishable, then also p,q are distinguishable.

This algorithm not only finds all pairs of distinguishable states, but all pairs which this algorithm doesn't mark as distinguishable, are equivalent. For a proof of this please consult [39].

If one applies the *Table-Filling* algorithm to the automaton of Figure 6.2, would get the following table where a square with a *x* indicates that states are distinguishable, a blank square indicates that states are equivalent:

	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14	q15
q2	x														
q3	x	x													
q4	x	x	x												
q5	x	x	x	x											
q6	x	x	x	x	x										
q7	x	x	x	x	x	x									
q8	x	x	x	x	x	x	x								
q9	x	x	x	x	x	x	x	x							
q10	x	x	x	x	x	x	x		x						
q11	x	x	x	x	x	x	x	x	x						
q12	x	x	x	x	x	x	x	x	x	x					
q13	x	x	x	x	x	x	x	x	x	x					
q14	x	x	x	x	x	x	x	x	x	x	x	x	x		
q15	x	x	x	x	x	x	x	x	x	x	x	x	x		
q16	x	x	x	x	x	x	x	x	x	x	x	x	x		

Table 6.1: *Table of distinguishable pairs of states.*

First we mark all pairs of states which contains a accepting and a non-accepting state (in our examples the accepting states are $q16, q15, q14$. Then we proceed recursively starting with state $q13$. The state $q13$ transits for $q16$ on input "r(ab)" and because $q16$ is final, we can mark immediately all states which transits by "r(ab)" to non-final states. This leave-us with the states $q11$ and $q12$ which both go to final states by the same symbol "r(a,b)". This is applied recursively to all states until we reached state $q2$ where all pairs are distinguishable or equivalent unless the pair with state $q1$ that is also distinguishable.

The *Table-Filling algorithm* can't construct a minimum DFA, but already partitions the states into equivalent classes. This allow us to put together all states which present the same behavior, in the sense of our state equivalence definition. A simple algorithm to construct blocks of equivalent states is for every state p, add in the same set all equivalent states to it. Although blocks of equivalent states can be built, we still don't know if all the states of the DFA will be present in some block of this partitioning. In [39] the author presented a theorem which proves that all states will be present in some set of equivalent states:

Theorem 2 *If we crate for each state q a block of equivalent states, all blocks together form a partition of the set of states. Each state is in exactly one block, all states within the same block are equivalent and states from different blocks are distinguishable.*

Knowing that blocks of equivalent states form a partition, we can pass to the algorithm which transforms a DFA $A = (Q, \Sigma, \delta, q_0, F)$ into a minimum DFA $B = (Q_b, \Sigma_b, \delta_b, q_b, F_b)$:

1. Apply the *Table-Filling algorithm* to find the pairs of equivalent states.

2. Create blocks of equivalent states.

3. The set Q_b of states are the blocks created in the previous step.

4. The transition function δ_b computes as follows: Let S be a block of equivalent states of DFA A and $a \in \Sigma$ a input symbol. Because all states in S are equivalent, there must be a block T of equivalent states in DFA A such that for all $q \in S$, $\sigma(q, a) \in T$.

5. The start state q_b of DFA B is the block containing the start state of A.

6. The set of accepting states F_b is the set of blocks containing at least one accepting state of DFA A.

7. The input symbols remain the same: $\Sigma_b = \Sigma$.

Now one can use this algorithm to construct a minimum DFA from a set of Alloy instances. This DFA can be used by the system modeler to express real-time properties in some real-time model checker as Uppaal or use the DFA to gain a deeper insight on the system states and transitions between them.

Using the previous algorithm, we can get the minimum DFA from the DFA of Figure 6.3:

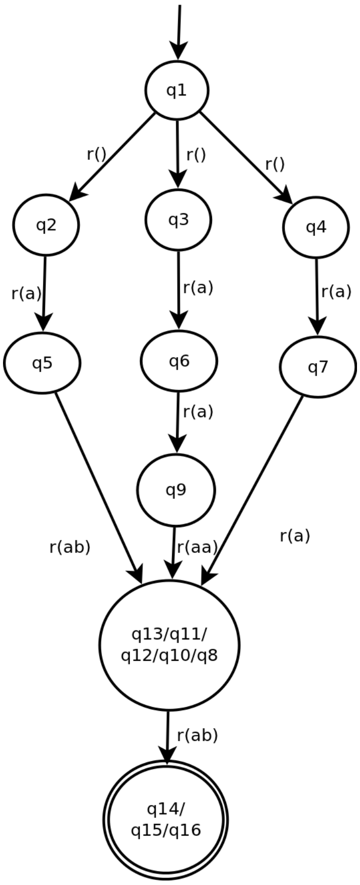


Figure 6.4: *The minimum DFA from a set of instances.*

The DFA A constructed with the previous algorithm are guaranteed to be minimal, i.e. there isn't other DFA which accepts the same language and has less states. If we apply this method to our model of the river crossing problem, we will get all possible states of the problem:

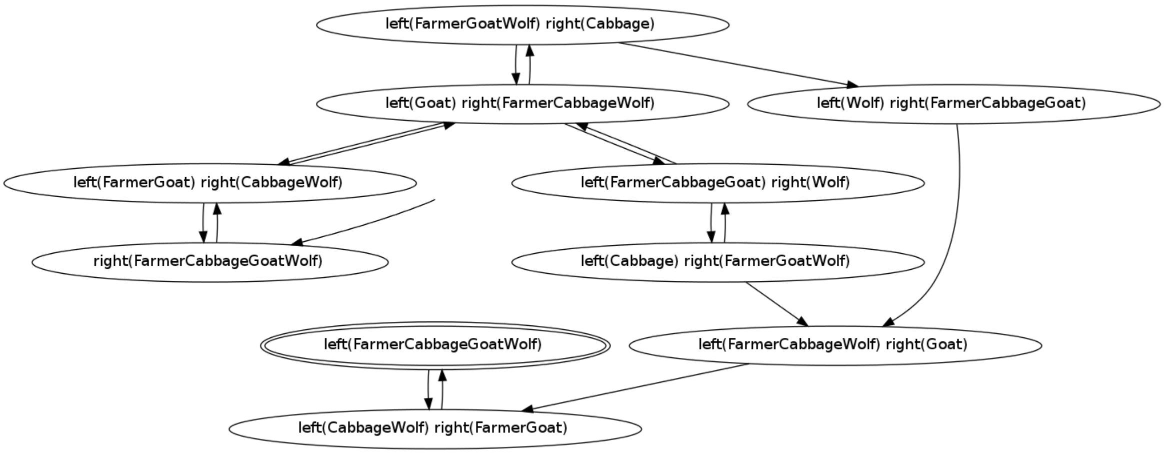

left(FarmerGoatWolf) right(Cabbage)
left(Goat) right(FarmerCabbageWolf)
left(Wolf) right(FarmerCabbageGoat)
left(FarmerGoat) right(CabbageWolf)
left(FarmerCabbageGoat) right(Wolf)
right(FarmerCabbageGoatWolf)
left(Cabbage) right(FarmerGoatWolf)
left(FarmerCabbageGoatWolf)
left(FarmerCabbageWolf) right(Goat)
left(CabbageWolf) right(FarmerGoat)

Figure 6.5: *The minimum DFA of the river-crossing problem.*

Looking at the description of the river-crossing problem, this is exactly what we expected as its DFA. The farmer can always cross alone from one side to the other, this is what makes that every state in the DFA has an arrow to the previous state. The DFA shows all the solutions to the problem (which consists in every path starting in the initial state and ending in the final state).

We've presented a method that allows the transformation of Alloy models into a DFA. The following sections will show how to use this method as a bridge between Alloy and Uppaal models. This is possible due to the fact that Uppaal models are networks of timed-automata, which are mathematical structures with great similarity with DFA's.

6.3 Tool Support

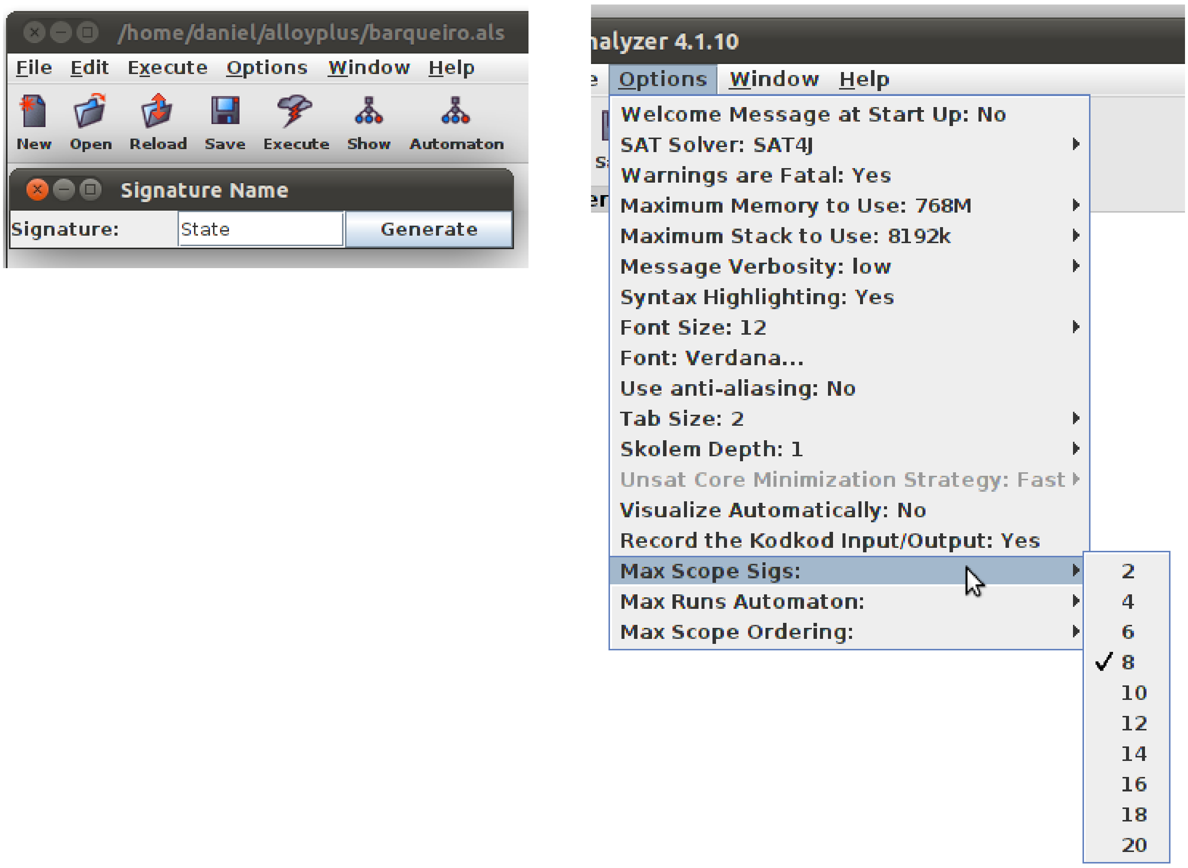
The method described in Section 6.2 was implemented in a tool which fully supports the transformation of a Alloy model to a DFA. The DFA is showed in a graphic representation in both Graphviz and Alloy and then converted automatically to a Uppaal model. In this section we will see in detail how the tool was integrated in the Alloy GUI how to extract model instances from Alloy to a external format which is easier to process and how all this is combined to retrieve a DFA of the model.

Integration with Alloy. This tool was integrated in the Alloy GUI to improve usability and relief the user from changing to other tool when he wants to see the a DFA of the system. The tool assumes the user has applied the trace idiom where he created a ordering in some signature and a fact which constraints the evolution of the model by a set of predicates.

The DFA can only be generated after we've retrieved a set of instances, as explained in Section 6.2. Remember that each instance in the local state is interpreted as a computation of a hidden automaton with length n (n being the number of atoms of the ordering). To extract a set of instances we need three parameters:

- S : The signature name that is suppose to be expanded. Because a model may contain several orderings, we need to ask the user which one is suppose to evolute.
- W : The maximum number of instances for signatures not in the ordering.
- N : The maximum length of the computation we want.
- K : The maximum number of model instances we want to have for each trace size.

The user can set the parameters W, N and K using the following interface:


/home/daniel/alloyplus/barqueiro.als
File Edit Execute Options Window Help
New Open Reload Save Execute Show Automaton
Signature Name
Generate
alyzer 4.1.10
Options Window Help

Signature: State

Welcome Message at Start Up: No
SAT Solver: SAT4J
Warnings are Fatal: Yes
Maximum Memory to Use: 768M
Maximum Stack to Use: 8192k
Message Verbosity: low
Syntax Highlighting: Yes
Font Size: 12
Font: Verdana...
Use anti-aliasing: No
Tab Size: 2
Skolem Depth: 1
Unsat Core Minimization Strategy: Fast
Visualize Automatically: No
Record the Kodkod Input/Output: Yes
Max Scope Sigs:
Max Runs Automaton:
Max Scope Ordering:

2
4
6
✓ 8
10
12
14
16
18
20

(a) *S Parameter*

(b) *M, N and K Parameters*

Figure 6.6: *Tool Interface*

The user can set parameter M, K and S (Max Scope Sig, Max Runs Automaton and Max Scope Automaton, respectively) in the options menu. Then he presses the new button ”Automaton“ (at the right side of the ”Show“ button) and after entering the signature which is to be expanded, presses the generate button to create the DFA.

After the tool processing is complete complete, the DFA is shown to the user in the Alloy usually way of representing model. This user has all the usual features at his disposal (zoom in and out, moving the states, etc).

Model Extraction. Alloy has a powerful API which allow an external program to perform multiple actions in some model. Usually when one wants to visualize an instance, Alloyshows a visual representation with boxes for atoms and arrows for relations. Then each of these instances needed to be processed and save them somehow for further processing (application of DFA construction and minimization).

To easily process each instance, the tool developed uses a Extensible Markup Language (XML) representation of models. In addition to the visual representation of Alloy, we also have the XML representation of models. As an example, consider the following Alloy instance:

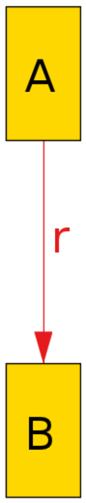


Figure 6.7: *Alloy Instance.*

This instance shows two atoms A and B and a relation r relating them. Alloy also represents this instance in the following XML:

```
< sig label="this/B" ID="4" parentID="2">
   < atom label="B$0"/>
< /sig>

< sig label="this/A" ID="5" parentID="2">
   < atom label="A$0"/>
< /sig>

< field label="r" ID="6" parentID="5">
   < tuple>  < atom label="A$0"/>  < atom label="B$0"/>  < /tuple>
   < types>  < type ID="5"/>  < type ID="4"/>  < /types>
< /field>

< sig label="univ" ID="2" builtin="yes">
< /sig>
```

Signatures are represent by *<sig>* tags and relations are represented by *<label>* tags. Inside a *<label>* table multiple tuples can appear as the relation relations the atoms. In this example, relation *r* relates atom A with atom B.

The tool then process these XML files and save each instance in a data structure to be further processed. Looking at the XML representation and knowing which signature is in the ordering of the *Trace Idiom* we can identify the order of the atoms solely looking at their lexicographic order ([43]). Each instance is saved as a graph and the atoms which comes from the ordering are represented are nodes in the graph. Their names is equal to the relations configuration. As an example consider the following instance where the signature A is the ordering:

```
< sig label="this/B" ID="4" parentID="2">
   < atom label="B$0"/>
   < atom label="B$1"/>
< /sig>

< sig label="this/A" ID="5" parentID="2">
   < atom label="A$0"/>
   < atom label="A$1"/>
< /sig>

< field label="r" ID="6" parentID="5">
   < tuple>  < atom label="A$0"/>  < atom label="B$0"/>  < /tuple>
   < tuple>  < atom label="A$0"/>  < atom label="B$1"/>  < /tuple>
   < tuple>  < atom label="A$1"/>  < atom label="B$1"/>  < /tuple>
   < types>  < type ID="5"/>  < type ID="4"/>  < /types>
< /field>
```

Just with this XML we know that atom $A0$ is previous to $A1$ and we can infer their names in the graph looking at all relations which relates A's to any other signature. In this case, only relation *r* relates A's to B's. When the tool read a instance like this, creates the following graph:

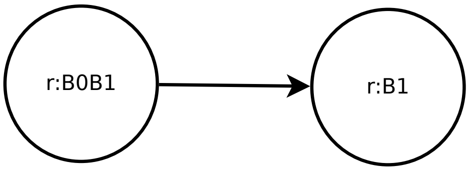


Figure 6.8: *Graph from an instance.*

Using this kind of instance extraction, the tool can easily retrieve a instance from a model in local state to a graph representation which can be further processed. The tool uses the following algorithm to get all possible instances within the parameters N, K and S (maximum length of the computation, maximum number of instances for each trace size and the signature in the ordering):

Algorithm 2 Get All Instances from a Alloy model

```
Require: $S \leftarrow$ signature in the ordering
Require: $W \leftarrow$ maximum number of instances for signatures not in the ordering.
Require: $N \leftarrow$ maximum length of the computation
Require: $K \leftarrow$ maximum number of instances for each trace
  for $i = 1 \rightarrow\ N$ do
    $j = 1$
    $instance \leftarrow$execute (run {} for $W$ but exactly $i$ $S$)
    $saveInstance(instance)$
    while $instantece.hasNext$ and $j <= K$ do
      $saveInstance(instance)$
      $j \leftarrow\ j + 1$
    $i \leftarrow\ i + 1$
```

The *saveInstance* function proceeds as explained before, firstly gets the XML of the instance and secondly transform that XML into a graph. The *.hasNext* predicate is provided by Alloy and indicates if there is more instances of the model withing the same scope. This algorithm allow us to find multiple instances of the model, which will result in richer DFA's.

Complete Cycle. When the user presses the *Generate* button in the interface, several steps are taken to create the minimum DFA of the system. Firstly all instances are read into XML files and saved into a temporary folder. When all instances are retrieved, the tool iterates throught the XML files and for each one, creates its DFA and minimize it with the existing DFA so far.

When all XML files are processed, the tool has a minimum DFA of system saved in an internal data structure. This data structure is then printed in three formats: A image created through Graphviz, a projection of the graph directly in Alloy and more important, a Uppaal model with all states and pre-defined transitions of the system.

6.4 Final Considerations

Until now we've seen method and subsequent tool support which transform a Alloy model into a correspondent DFAȦll the user has to do is to introduce the *Trace Idiom* and then he has a free DFA at a distance of a button press. This DFA isn't developed by intuition or any mean of human intervention, it is constructed based on a formal model (Alloy model), where several characteristics of the system have already been considered.

This approach is far distant from the usual one, where the user has two options when a model of a real-time system is needed: he may choose to model the entire real-time system into a tool which effectively supports real-time constraints but lacks of a good support for a behavior specification, or he choses to model the system in a tool with excellent support for behavior specification but lacks of a good real-time support and at some point need to re-model the whole system in a real-time tool, practically giving away all the previous work he made.

Although this conversion may prove valuable to a requirement analysis or a systems modeler, it should be used wisely. Below we will see a technique which help us to retrieve the correct DFA of the models. Without the following techniques, in some models the DFA we get isn't exactly the correct one.

Abstract Interpretation. When models have relations with multiplicity as *set*, sometimes the DFA constructed isn’t appropriated to real-time modeling. In such the cases, the should perform a simple re-factor in the model, making the DFA better suited to specify real-time constraints.

Suppose one wants to model the behavior of adding items to a *Bag*, with the restriction that an item can only be added after the receipt of a trigger:

Alloy Code Display 2 Simple Model of the Bag

```
open util/ordering[A]

sig Item{}
sig C{}
sig A{
  bag : set Item,
  trigger : lone C
}

pred addItem[a,a’:A, i:Item]{
  some a·trigger
  i not in a·bag
  a’·bag = a·bag + i
  no a’·trigger
}

pred addC[a,a’:A, c:C]{
  no a·trigger
  a’·trigger = c
  a’·bag = a·bag
}

fact{
  no first·bag no first·trigger
  all a:A, a’:a·next | some i:Item,c:C | addItem[a,a’,i] || addC[a,a’,c]
}
```

The instances which Alloy gives us, clearly allow us to see that the model evolves over time as expected:

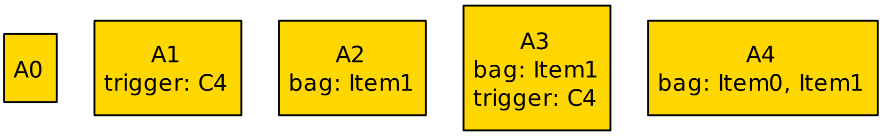


Figure 6.9: *Trace of execution.*

The $Item1$ it is only added to the bag after the appearance of a trigger and the same for $Item2$. The same kind of behavior it is reproduced for larger traces. If we apply the tool presented in Section 6.3 we will get the following DFA:

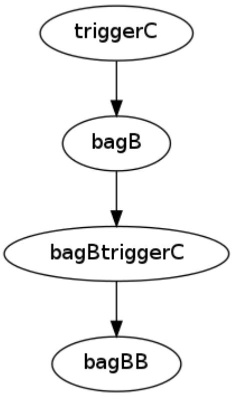


Figure 6.10: *DFA Generated without abstract interpretation.*

The reader might question him self: Why state *bagBtriggerC* transits to state *bagBB* instead of state *bagB*? If we want a DFA of this system, it makes sense that DFA created a cycle where the DFA add a item to the bag after the receipt of a trigger. Because Alloy doesn't know that for us adding one or adding 10 items to the bag is the same, it keeps adding different items to the bag making each new item in the bag a new state in the DFA.

In order create the correct DFA, the user should apply some kind of **abstract interpretation** relaxing the relation *bag* to represent that adding one or n items is the same. This abstract interpretation must ensure that predicates like the *addItem* still are used by the model but doesn't infer DFA like the one in Figure 6.10. Basically what ween need to reflect in the model is that it is insignificant if the relation *bag* has one or n items, what is important to distinguish is 0 items from one or more items:

Alloy Code Display 3 New predicate for adding items.

```
pred addItem[a,a’:A, i:Item]{
  some a·trigger
  i not in a·bag
  no a·bag ⇒ a’·bag = a·bag+ i else a’·bag=a·bag
  no a’·trigger
}
```

Using this predicate in our model, guarantees that and item can only be added after the receipt of a trigger and now we can generate a DFA suitable than the previous one:

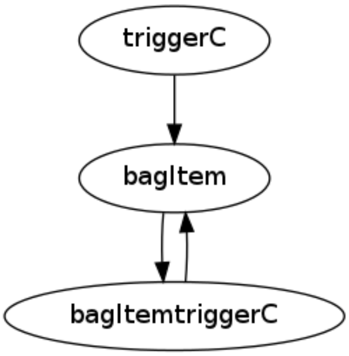


Figure 6.11: *Correct DFA Generated with the tool.*

In this DFA the requirement analyst can easily specify a real-time constraint like ”items shall be added at a rate of at least t time units“'.

This kind of *Abstract Interpretation* can help the requirement analyst to derive suitable DFAs to specify real-time constraints. The transformation we saw isn’t a ”must follow“, requirements analysts can and should abstract model relations as it suits him by looking at the evolution of the models and what kind of DFA he is getting.

River-Crossing with Time-Constraints. Suppose that adding to the river-crossing problem, we had five more constraints:

- The farmer takes 6 minutes to load and unload the goat or the wolf.
- The farmers takes 2 minutes to load and unload the cabbage.
- After unload a being, the farmer must stay in the margin at least 1 minute.
- If the farmer will cross alone, he doesn’t have to wait before departure.
- The farmer cannot stay more than 10 minutes if he is in the same side of the wolf.

If the river-crossing had these time-constraints and we want create a model of the problem we can start by creating a model of the problem in Alloy, forgetting the time constraints. When the model gave solutions to the non-temporal problem, we could use the tool to generate the DFA of Figure 6.5.

Now one just have to work on the Uppaal model obtained thought the Alloy to Uppaal conversion. The first thing one need to do is to declare a *clock* variable c to represent time in Uppaal : ***clock*** *c;*

With this clock variable we just need to go to every state in the *timed automata* previously generated and apply the time constrains. As described in Section 2.2 every transition between states has a *guard* condition, which must be true in order to the transition happen and *update* statement where we can set values to variables. Every state in Uppaal also has an *invariant* expression which must always be true in the automaton is in the state. Let p and q be two states of a *timed automata* which have a transition between them, then the guard and the update expression of the transition from p to q are γ_q^p and υ_q^p, respectively. The invariants of states p and q will be represented as λ_p and λ_q, respectively. For all states p and q in the *timed automata* previously generated for the river-crossing problem, if p and q are connected by a transition from p to q we proceed as follow:

- If the farmer will transport a goat or wolf from p to q: $\upsilon_q^p = c := 0$, $\gamma_q^p = c >= 7$.
- If the farmer will transport a cabbage from p to q: $\upsilon_q^p = c := 0$, $\gamma_q^p = c >= 3$.
- If the farmer will cross alone from p to q: $\upsilon_q^p = c := 0$, $\gamma_q^p = c >= 0$.
- If the farmer will stay in the same side of the wolf in state q: $\lambda_q = c <= 10$.

Applying the rules above will result in the Uppaal timed automata of Figure 6.12

In Uppaal the initial state is marked as a double circle (state *rightFarmerCabbageGoatWolf*), the state where the solution is achieved is the state colored with yellow (state *leftFarmerCabbageGoatWolf*). *Guards* are the green expressions in the transitions, *updates* are the blue expressions and the states invariants are the expressions in magenta.

This DFA transformation its a step further in the modeling of requirements. With this DFA transformation, a requirement analyst can start by specifying a requirements document with the approach described in Section 3.2 and then if at some point appears some real-time properties that he can't specify in Alloy can easily apply the *Trace Execution* idiom and then generate the DFAof the system.

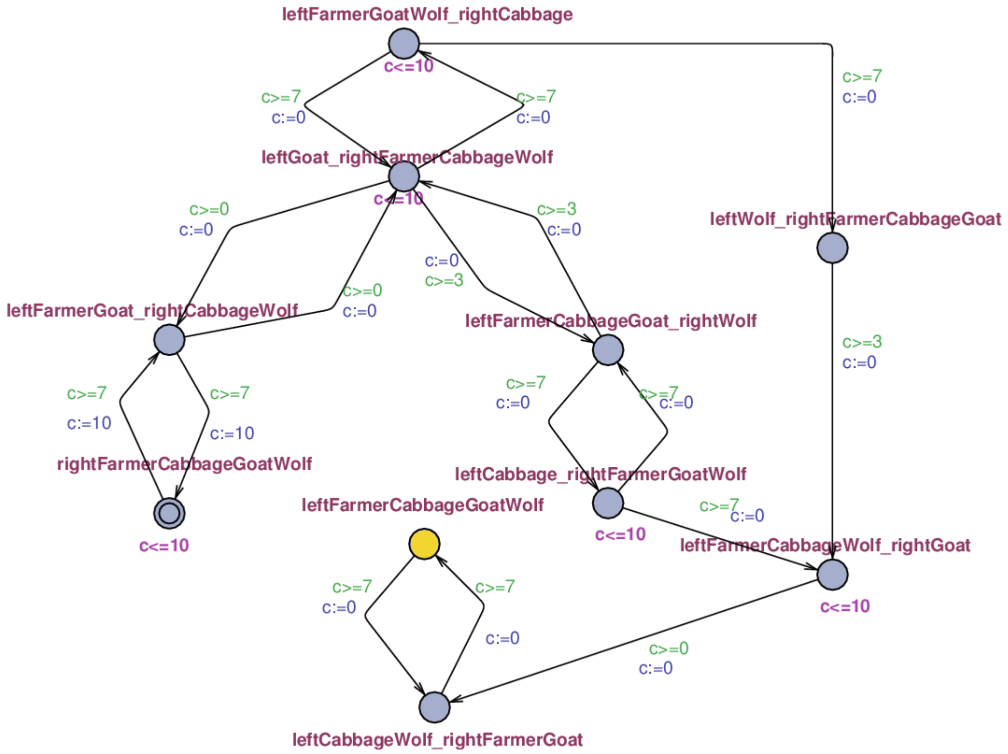

leftFarmerGoatWolf_rightCabbage
c<=10
c>=7
c:=0
c>=7
c:=0
leftGoat_rightFarmerCabbageWolf
c<=10
c>=0
c:=0
c>=3
c:=0
c>=7
c:=0
leftWolf_rightFarmerCabbageGoat
c:=0
c>=3
c>=0
c:=0
leftFarmerGoat_rightCabbageWolf
leftFarmerCabbageGoat_rightWolf
c>=3
c:=0
c>=7
c:=10
c>=7
c:=10
c>=7
c:=0
c>=7
c:=0
rightFarmerCabbageGoatWolf
leftCabbage_rightFarmerGoatWolf
leftFarmerCabbageGoatWolf
c>=7
c:=0
c<=10
c<=10
leftFarmerCabbageWolf_rightGoat
c>=7
c:=0
c>=7
c:=0
c<=10
c>=0
c:=0
leftCabbageWolf_rightFarmerGoat

Figure 6.12: *River Crossing with Real-Time Restrictions.*

6.5 Summary

This Chapter has shown how to convert a Alloy model to Uppaal and how this can be important when we need to model systems with explicit temporal properties. Converting Alloy to Uppaal is achieved by first introducing the Trace Idiom in the model and then converting it to a DFA. This transformation shows another way of seeing Alloy instances, where they represent paths in the DFA. After the initial DFA is obtained, a minimization algorithm is applied, collapsing similar states. Once the DFA is generated it suffices to convert it to the Uppaal format, and an Uppaal model is created.

The work presented in this chapter can be seen as further step to be used after chapters 3 and 5. The idea is to defer requirements stating explicit real time constraints to later stages in the modeling. In the initial phase everything can thus be handled in Alloy. When requirements engineers feel they can't go any further due to explicit real-time constraints they can take the model of all requirements thus far and transform it to a Uppaal model. With the model in this language it becomes easier to specify temporal constraints than if one has to do it with Alloy.

Chapter 7

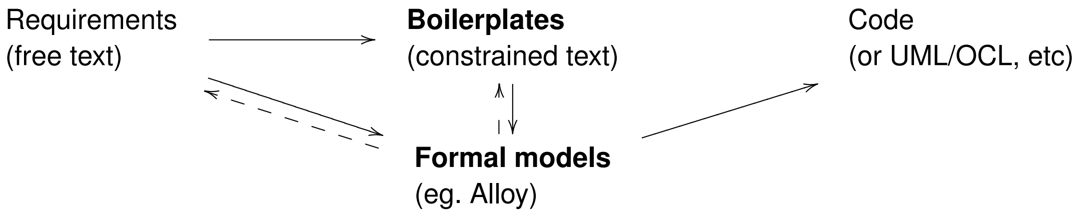
Conclusions and Future Work

This dissertation has tried to provide evidence of the advantages of using formal method techniques since the early phases of software development, namely in requirements analysis. An approach is put forward which emphasizes on traceability of requirements and mathematical models which provide the shift from informal to formal specifications upon which subsequent development steps should rely upon, notably code writing.

Such combined use of requirements documents and mathematical models allow requirements engineers to early spot design flaws and timely correct them, trimming risk and saving spurious development costs. Such a close link between text and model hopefully bends the learning curve which requirements engineers are faced with when adopting formal methods.

Particular emphasis is put on the Alloy formal specification language which provides an easy-to-use, lightweight approach to formal modeling. Model checking in Alloy is, in particular, an effective and practical way of checking requirement documents. The similarity between Alloy's navigation-styled notation and object-oriented programming makes it attractive to the average programmer. Its simple relational flavour makes it specially tractable in mathematical terms.

The timely adoption of formal modeling before code-writing is central to the approach put forward in this dissertation. The idea is to provide support for semi-automatically managing the process of distilling text into formal models, as seen in chapters 3 and 6, for instance. This is intended to relief the requirements engineers from manual tasks which usually are highly error prone and slow. This process linking text to formal models is captured by the following diagram:



The above diagram shows two options in Alloy-based requirement analysis: either one models requirements documents using one's own expertise in Alloy (Chapter 3) or boilerplated text can be used (Chapter 5) in-between as a means to generate formal models for free, using a repository of boilerplates and corresponding Alloy models.

Chapter 3 has shown how a requirements document can be manually translated to mathematical models. For improved traceability, this translation is not performed as a whole, but rather on a stepwise manner, showing explicitly which requirement maps into which part of the model. This chapter can be seen as a first introduction to chapter 5, where we go a step further in showing that some textual patterns can be related to corresponding patterns in formal modeling. Through such *boilerplated* text and Alloy, we can model an entire requirement without directly writing a single line of Alloy. This isn't fully achieved in Chapter 3, where we still need someone to build the models. The work presented in Chapter 5 is the subject of paper [10] and was presented to the software industry in the Dependability Workshop at Critical Software [1].

Alloy has proved a versatile language and tool in supporting the methods presented in Chapters 3 and 5 and in providing a bridge towards the world of real time constraints (Chapter 6). Such a need for a bridged process was identified by looking at how people create models in Alloy and how it becomes increasingly difficult to cope with real-time constraints, calling for a subsequent modeling phase in another (more elaborate) language where such constraints are native, as is the case of Uppaal. Such a transformation is hard and error prone and results in two models which represent different things. Chapter 6 presents a technique and a tool support which translate Alloy models in the *local-trace idiom* to Uppaal models, without any user intervention. This translation can be applied both in models from chapter 3 and 5.

Future Work

This dissertation raises several issues for future work. Chapter 3 presented a method and its tool support. This tool can evolve to an integrated tool with a GUI better supporting the requirement engineer to write requirements and their subsequent models, including graphics representation and assertions. One can force requirements engineers to write requirements in some standard layout (eg: XML) and then adapt the tool to process this layout. This would decouple the tool from LaTeX or any another specific document preparation system.

Concerning Chapter 5 which relates requirements boilerplates and Alloy, there is much space left for research and tool development. One could further investigate on the relationship between boilerplates and formal method models and create a wider repository of boilerplates and their models. One could develop further an algebra for boilerplates, adding new operations to the ones presented in Chapter 5 allowing to combine different kinds of models. This algebra of boilerplates would show how to combine different Alloy models, to represent the same problem. This combination would be made of well defined rules and not on user intuition or experience.

Some of these ideas have already been taken for development by the **Educed** [2] team, where the PROVA tool is currently under development to support the ideas presented in Chapter 5. PROVA offers the translation from boilerplated requirements to Alloy. Ongoing work includes further development of boilerplates for dynamic behavior, namely through boilerplates that constrain valid traces in Alloy and correspond directly to LTL formulæ in Temporal Alloy. This is addressed in [10].

One topic which can be improved is the relation between models and text. The intrinsic relation between text and formal models

$$\textbf{Requirements} \quad \rightleftarrows \quad \textbf{Formal Models}$$

[1] Please see: http://www.criticalsoftware.com/dependability-workshop.

[2] See: http://www.educed-emb.com.

can be further extended to identify what impact a change in text has in the model. Today this is achieved using expertise from modelers and requirement analysts. In the future such a task could become semi-automatic. This correlation can even be further expanded to add metrics on text quality based on formal models and through that, improve text quality.

Another point of interest for future research is to adapt the work of this dissertation to general purpose documents used by civil society (eg: company norms, regulations, legal text etc). Such work would be particularly interesting because it would dramatically increase the spectrum of documents which can be analyzed with *formal methods*. This is important if one wants to create a line of business using *formal methods* to analyze text. In fact, not only software industry has gained interest on linguistic improvements through formal methods. The core business of company PORTUGUÊSCLARO [3] is text-clearance, inspired in the ideas born with Plain English [16]. They share the same aim of rewriting text in a way which removes inconsistencies and ambiguities which can mislead readers. PORTUGUÊSCLARO has shown some interest on checking the use of formal models in assisting their text analysis, to help them clear text on a semantic basis, writing better English and Portuguese. This is an evidence that requirement engineering assisted with formal methods can evolve into a wider and broader topic: documents in general improved through formal methods, where no restriction is made on the type of documents one is addressing.

[3] Please See: http://portuguesclaro.pt

Bibliography

[1] I.F. Alexander and R. Stevens. *Writing Better Requirements*. Pearson Education, 2002. [cited at p. 4, 7]

[2] J. Backus. Acm turing award lectures. chapter Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs, pages 1977–. ACM, New York, NY, USA, 2007. [cited at p. 20]

[3] C. Baier and J. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. [cited at p. 6]

[4] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions*. Springer-Verlag, 2004. [cited at p. 6]

[5] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31:137–149, 2005. [cited at p. 19]

[6] R. Bird and O. Moor. *Algebra of programming*. Prentice-Hall international series in computer science. Prentice Hall, 1997. [cited at p. 19, 21, 59]

[7] J. Bowen and V. Stavridou. Safety-Critical Systems, Formal Methods and Standards. *Software Engineering Journal*, 8(4):189–209, July 1993. [cited at p. 2]

[8] H. Bowman, G. Faconti, J-P. Katoen, D. Latella, and M. Massink. Automatic verification of a lip synchronisation algorithm using uppaal. In *Proc. of the 3rd International Workshop on Formal Methods for Industrial Critical Systems*, pages 97–124, 1998. [cited at p. 23]

[9] F. Bruijn and H.L. Dekkers. Ambiguity in natural language software requirements: A case study. In *REFSQ*, pages 233–247, 2010. [cited at p. 6]

[10] D. Cadete, A. Cunha, J.M. Faria, J.N. Oliveira, and A. Passos. From boilerplated requirements to alloy: half-way between text and formal model. December 2011. (Submitted). [cited at p. 61, 84]

[11] T.D Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43:225–267, March 1996. [cited at p. 2]

[12] R.N. Charette. Why Software Fails. *IEEE Spectrum*, 2005. [cited at p. 2]

[13] P. Chauhan, E.M. Clarke, J.H. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, FMCAD '02, pages 33–51, London, UK, 2002. Springer-Verlag. [cited at p. 63]

[14] E.M. Clarke and J.M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28:626–643, December 1996. [cited at p. 2, 7]

[15] J. Cleland-Huang, R. Settimi, E. Romanova, B. Berenbach, and S. Clark. Best practices for automated traceability. *Computer*, 40:27–35, June 2007. [cited at p. 5]

[16] M. Cutts. *Oxford Guide to Plain English*. Oxford University Press, USA, 2007. [cited at p. 85]

[17] Information Assurance Directorate. U.S. government protection profile for separation kernels in environments requiring high robustness — version 1.03, June 2007. [cited at p. 41]

[18] J. S. Dong, P. Hao, S. Qin, and X. Zhang. The semantics and tool support of ozta. In K.-K. Lau and R. Banach, editors, *Formal methods and software engineering : 7th International Conference on Formal Engineering Methods, ICFEM 2005, 1-4 November, 2005, Manchester, UK ; proceedings.*, number 3785 in Lecture notes in computer science, pages 66–80. Springer, Berlin, November 2005. [cited at p. 64]

[19] J. S. Dong, P. Hao, X. Zhang, and S. Qin. Highspec : a tool for building and checking ozta models. In *28th International Conference on Software Engineering, 20-28 May 2006, Shanghai, China ; proceedings.*, pages 775–778. Association for Computing Machinery, May 2006. [cited at p. 64]

[20] J.S. Dong and R. Duke. Integrating object-z with timed automata. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 488–497, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 64]

[21] J.S. Dong, P. Hao, S. Qin, J. Sun, and Y. Wang. Timed automata patterns. *IEEE Trans. Softw. Eng.*, 34:844–859, November 2008. [cited at p. 64]

[22] M. Elizabeth, C. Hull, K. Jackson, and J. Dick. *Requirements engineering (2. ed.)*. Springer, 2005. [cited at p. 55, 56]

[23] F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. The linguistic approach to the natural language requirements quality: Benefit of the use of an automatic tool. In *Proceedings of the 26th Annual NASA Goddard Software Engineering Workshop*, SEW '01, pages 97–, Washington, DC, USA, 2001. IEEE Computer Society. [cited at p. 7]

[24] M.A. Ferreira and J.N. Oliveira. "relational thinking" for software engineering: a case study. *Journal paper (submitted).*, 2011. [cited at p. 21]

[25] M. Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. [cited at p. 6]

[26] M.F. Frias. Fork algebras in algebra, logic and computer science, 2002. Logic and Computer Science. World Scientific Publishing Co. [cited at p. 59]

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995. [cited at p. 17]

[28] A. Garis, A. Cunha, and D. Riesco. Translating alloy specifications to uml class diagrams annotated with ocl. In *Proceedings of the 9th international conference on Software engineering and formal methods*, SEFM'11, pages 221–236, Berlin, Heidelberg, 2011. Springer-Verlag. [cited at p. 19]

[29] B. Gerd, A. David, and K.G. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004. [cited at p. 25]

[30] R. Gheyi, T. Massoni, and P. Borba. Formally introducing alloy idioms. pages 22–37, Ouro Preto, Brazil, 2007. [cited at p. 19]

[31] T. Giannakopoulos, D.J. Dougherty, K. Fisler, and S. Krishamurthi. Towards an operational semantics for alloy. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 483–498, Berlin, Heidelberg, 2009. Springer-Verlag. [cited at p. 9]

[32] R.L. Glass. *Software runaways: monumental software disasters*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998. [cited at p. 2]

[33] J.A. Goguen and C. Linde. Techniques for Requirements Elicitation. In Stephen Fickas and Anthony Finkelstein, editors, *Requirements Engineering '93*, pages 152–164. IEEE, 1993. [cited at p. 3]

[34] O.C.Z. Gotel and C.W. Finkelstein. An analysis of the requirements traceability problem. In *International Conference on Requirements Engineering*, pages 94–101, 1994. [cited at p. 4]

[35] J. V. Guttag, J. J. Horning, Withs. J. Garl, K. D. Jones, A. Modet, and J. M. Wing. Larch: Languages and tools for formal specification. In *Texts and Monographs in Computer Science*. Springer-Verlag, 1993. [cited at p. 5]

[36] K. Havelund, K. Guldstr, and A. Skou. Formal verification of a power controller using the real-time model checker uppaal. In *In 5th International AMAST Workshop on Real-Time and Probabilistic Systems, volume Lecture Notes in Computer Science*, pages 277–298, 1999. [cited at p. 23]

[37] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using uppaal. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS '97, pages 2–, Washington, DC, USA, 1997. IEEE Computer Society. [cited at p. 23]

[38] C.L. Heitmeyer. Formal methods for specifying validating, and verifying requirements. *Journal of Universal Computer Science*, pages 607–618, 2007. [cited at p. 6]

[39] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. [cited at p. 68, 69, 70]

[40] ARINC INC. Arinc report 653p1-3 avionics application software interface, part 1, required services. Technical report, 2010. [cited at p. 41]

[41] T.K. Iversen, K.J. Kristoffersen, K.G. Larsen, M. Laursen, R.G. Madsen, S.K. Mortensen, P. Petterson, and C.B. Thomasen. Model-checking real-time control programs - verifying lego mindstorms systems using uppaal. In *In Proc. of 12th Euromicro Conference on Real-Time Systems*, pages 147–155. IEEE Computer Society Press, 2000. [cited at p. 23]

[42] D. Jackson. Dependable Software by Design. *Scientific American*, June 2006. [cited at p. 7]

[43] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006. [cited at p. 11, 16, 17, 75]

[44] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *In Proceedings of the International Conference on Software Engineering (ICSE)*, pages 730–733, 2000. [cited at p. 10]

[45] M. Jackson. Defining a discipline of description. *IEEE Software*, 15(5):14–17, 1998. [cited at p. 29]

[46] M. Jackson and P. Zave. Domain descriptions. In *International Symposium on Requirements Engineering*, pages 56–64, 1993. [cited at p. 29]

[47] J.L. Jacob. Trace Specifications in Alloy. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, chapter 9, pages 105–117–117. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010. [cited at p. 18, 59, 64]

[48] P. Jalote. *An integrated approach to software engineering*. Springer-Verlag New York, Inc., New York, NY, USA, 1991. [cited at p. 7]

[49] C.B. Jones. Systematic software development using vdm - teaching notes, 1995. [cited at p. 5]

[50] S. Jones, D. Till, and A.M. Wrightson. Formal methods and requirements engineering: Challenges and synergies. *Journal of Systems and Software*, 40(3):263–273, 1998. [cited at p. 6]

[51] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992. [cited at p. 32]

[52] I. Koren and C.M. Krishna. *Fault Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. [cited at p. 2]

[53] J. Kramer. Is abstraction the key to computing? *Commun. ACM*, 50(4):37–42, April 2007. [cited at p. 55]

[54] L. Lampor. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994. [cited at p. 47, 63]

[55] L. Lamport. *LATEX : a document preparation system : user's guide and reference manual*. Addison-Wesley Pub. Co., Reading, Mass., 1994. [cited at p. 35]

[56] L. Lamport. Real-time model checking is really simple. In *CHARME*, pages 162–175, 2005. [cited at p. 47]

[57] G.L. Lann. An analysis of the ariane 5 flight 501 failure - a system engineering perspective. In *Proceedings of the 1997 international conference on Engineering of computer-based systems*, ECBS'97, pages 339–346, Washington, DC, USA, 1997. IEEE Computer Society. [cited at p. 2]

[58] P.A. Laplante. *What Every Engineer Should Know about Software Engineering (What Every Engineer Should Know)*. CRC Press, Inc., Boca Raton, FL, USA, 2007. [cited at p. 3]

[59] K.G. Larsen, P. Petterson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997. [cited at p. 23]

[60] N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993. [cited at p. 2]

[61] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992. [cited at p. 5]

[62] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209. [cited at p. 6]

[63] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. [cited at p. 5]

[64] G.J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. [cited at p. 2]

[65] P. Naur and B. Randell. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968*. Scientific Affairs Division, NATO, Brüssel, 1969. [cited at p. 1]

[66] R. Neufville. The baggage system at denver: prospects and lessons. *Journal of Air Transport Management*, 1(4):229 – 236, 1994. [cited at p. 2]

[67] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, New York, NY, USA, 2000. ACM. [cited at p. 3]

[68] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20:760–773, 1994. [cited at p. 29]

[69] J.N. Oliveira. Generative and transformational techniques in software engineering ii. chapter Transforming Data by Calculation, pages 134–195. Springer-Verlag, Berlin, Heidelberg, 2008. [cited at p. 22]

[70] J.N. Oliveira. *Extended Static Checking by Calculation Using the Pointfree Transform*, pages 195–251. Springer-Verlag, Berlin, Heidelberg, 2009. [cited at p. 20]

[71] D.L. Parnas. Really rethinking 'formal methods'. *Computer*, 43:28–34, January 2010. [cited at p. 29]

[72] P. Poizat, C. Choppy, and J. Royer. From informal requirements to coop: a concurrent automata approach, 1999. [cited at p. 6]

[73] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, 6 edition, April 2004. [cited at p. 7]

[74] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27:58–93, January 2001. [cited at p. 5]

[75] A. Redouane. Experience using formal methods for capturing requirements of web-based applications. In *Proceedings of the 1st IEEE International Conference on Cognitive Informatics*, pages 213–221, Washington, DC, USA, 2002. IEEE Computer Society. [cited at p. 6]

[76] Daniel Sheridan. *Temporal Logic Encodings for SAT-based Bounded Model Checking*. PhD thesis, University of Edinburgh, November 2005. [cited at p. 6]

[77] D.J. Sheridan. *Temporal logic encodings for SAT-based bounded model checking*. University of Edinburgh, 2006. [cited at p. 63]

[78] I. Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9. edition, 2010. [cited at p. 2, 3]

[79] The Standish Group. Chaos report, 1995. http://www.cs.nmt.edu/~cs328/reading/Standish.pdf – last visited 15^{th} of June, 2008. [cited at p. 3]

[80] M. Vetterling, G. Wimmel, and A. Wisspeinter. Secure systems development based on the common criteria: the palme project. *SIGSOFT Softw. Eng. Notes*, 27:129–138, November 2002. [cited at p. 41]

[81] J. M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–26, September 1990. [cited at p. 5]

[82] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*, volume 1. Prentice Hall Upper Saddle River, NJ, 1996. [cited at p. 5]

[83] H. Yang, A. Willis, A. De Roeck, and B. Nuseibeh. Automatic detection of nocuous coordination ambiguities in natural language requirements. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 53–62, New York, NY, USA, 2010. ACM. [cited at p. 6]

Index

Appendix A

Partitioning Kernel Modeling

Secure Partitioning Kernel — A Case Study in Requirements Engineering

January 21, 2012

Contents

Chapter 1

Partition Requirements

1.1 Partition Attributes

Requirement Number	PRT#001
Title	Partition Attributes
Description	Each Partition shall have the following attributes: • Identifier: uniquely defined on a system-wide basis, and used to facilitate partition activation and message routing. • Memory Requirements: defines memory bounds (minimum and maximum quotas) of the partition, with appropriate code/data segregation. • Period: defines the activation period of the partition, and is used to determine the partition's runtime placement within the core module's overall time frame. • Duration: the amount of processor time (minimum and maximum quotas) given to the partition every period of the partition. • Criticality Level: denotes criticality level of partition. • Partition Health Monitor Table (health monitor reconfigurations): denotes instructions to the HM on the actions required. • Entry Point (i.e., partition initialization): denotes partition elaboration restart address. • System Partition: denotes if the partition is a system partition. • Lock level: denotes the current lock level of the partition. • Start condition: denotes the reason the partition is started
Rationale	N/A.

1.1.1 Alloy

```
sig Partition {
   minimum : one Int,
   maximum : one Int,
   period : one Int
}

fact Constraints{
   all p:Partition | gt[p·minimum,0]
   all p:Partition | gt[p·maximum,0]
   all p:Partition | gt[p·period,0]
   all p:Partition | gt[p·maximum,p·minimum]
   all p:Partition | gte[p·period,p·maximum]
}
```

1.1.2 Metamodel

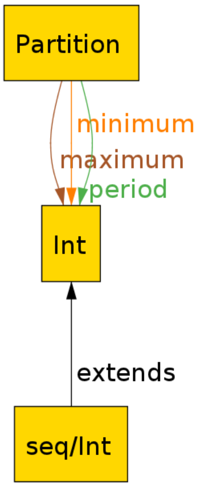


Figure 1.1: Requirement Metamodel.

1.2 Partition Modes

Requirement Number	PRT#004
Title	Partition Modes
Description	The Partition shall have the following modes: • IDLE: In this mode, the partition is not executing any processes within its allocated partition windows. The partition is not initialized (e.g., none of the ports associated to the partition are initialized), no processes are executing, but the time windows allocated to the partition are unchanged. • NORMAL: In this mode, the process scheduler is active. All processes have been created and those that are in the ready state are able to run. The system is in an operational mode. • COLD_START: In this mode, the initialization phase is in progress, preemption is disabled with LOCK_LEVEL=0 (process scheduling is inhibited) and the partition is executing its respective initialization code. • WARM_START: In this mode, the initialization phase is in progress, preemption is disabled with LOCK_LEVEL=0 (process scheduling is inhibited) and the partition is executing its respective initialization code. This mode is similar to the COLD_START but the initial environment (the hardware context in which the partition starts) may be different, e.g., no need for copying code from Non Volatile Memory to RAM.
Rationale	N/A.

1.2.1 Alloy

```
sig Time {}
sig Mode {}
sig Normal, Idle extends Mode {}

sig Partition {
  minimum : one Int,
  maximum : one Int,
  period : one Int,
  mode : Mode one → Time,
}

fact Modes{
  #(Normal)=1
  #(Idle)=1
  Mode = Normal + Idle
}
```

1.2.2 Metamodel

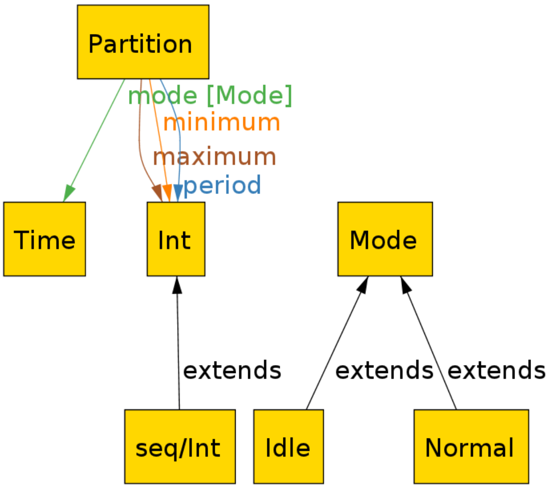


Figure 1.2: Requirement Metamodel.

1.3 Partition Secheduling Characteristics

Requirement Number	PRT#007
Title	Partition Scheduling Characteristics
Description	The main characteristics of the partition scheduling model shall be: • The scheduling unit is a partition. • Partitions have no priority. • The scheduling algorithm is predetermined, repetitive with a fixed periodicity, and is configurable by the system configuration only. At least one partition window is allocated to each partition during each cycle. • The core module level O/S exclusively controls the allocation of the resources to the partition.
Rationale	N/A.

1.3.1 Alloy

```
fun prox : Time → Time {
  ordering/next + last → first
}

fun diff [t,t' : Time] : Int {
  #(t' in (t·nexts + t) ⇒ (t·nexts & (t' + t'·prevs)) else (t·nexts + t'·prevs + t))
}

pred i2n [p : Partition, t : Time] {
  p·mode·t = Normal
  p·mode·(t·~prox) = Idle
}

pred n2i [p : Partition, t : Time] {
  p·mode·t = Idle
  p·mode·(t·~prox) = Normal
}

fact Scheduler{
  all t : Time | lone (mode·t)·Normal
  all p : Partition | some t : Time | p·mode·t = Normal
  all p : Partition, t : Time | i2n[p,t] ⇒
  {
    (some t' : Time | eq[diff[t,t'],p·period] and i2n[p,t'])
    &&
    (all t' : Time | lte[diff[t,t'],p·minimum] ⇒ p·mode·t'= Normal)
    &&
    (all t' : Time | gt[diff[t,t'],p·maximum] and lt[diff[t,t'], p·period]⇒ p·mode·t'=Idle)
    &&
    (all t' : Time | gt[diff[t,t'],0] and lt[diff[t,t'],p·period] ⇒ not i2n[p,t'])
  }
}
```

Chapter 2

Process Requirements

2.1 Processes Attributes

Requirement Number	PRC#004
Title	Processes Attributes
Description	A set of unique attributes shall be defined for each process within the system. These attributes differentiate between unique characteristics of each process as well as define resource allocation requirements. The following attributes a process shall have: • Base Priority - Denotes the capability of the process to manipulate other processes. • Period - Identifies the period of activation for a periodic process. A distinct and unique value should be specified to designate the process as aperiodic. • Time Capacity - Defines the elapsed time within which the process should complete its execution. • Current Priority - Defines the priority with which the process may access and receive resources. It is set to base priority at initialization time and is dynamic at runtime. • Process State - Identifies the current scheduling state of the process. The state of the process could be either dormant, ready, running or waiting.
Rationale	N/A.

2.1.1 Alloy

```
sig State {}
sig Ready, Running extends State {}

fact Modes{
  #(Normal)=1
  #(Idle)=1
  Mode = Normal + Idle

  #(Ready)=1
  #(Running)=1
  State = Ready + Running
}

sig Process{
  prt : one Partition,
  p_period : one Int,
  time_capacity : one Int,
  base_priority : one Int,
  curr_priority : Int one → Time,
  state : State one → Time,
}

fact ProcessConstraints{
  gt[Process·p_period,0]
  gt[Process·time_capacity,0]
  all p:Process | gt[p·base_priority, 0]
  all p:Process, t:Time | gt[p·curr_priority·t,0]
  all p:Process | eq[p·base_priority, p·curr_priority·first]
  all p:Process | gte[p·p_period,p·time_capacity]
  all p:Process | gt[p·p_period, p·prt·period]
  all p:Partition | some p':Process | p'·prt = p
}
```

2.1.2 Metamodel

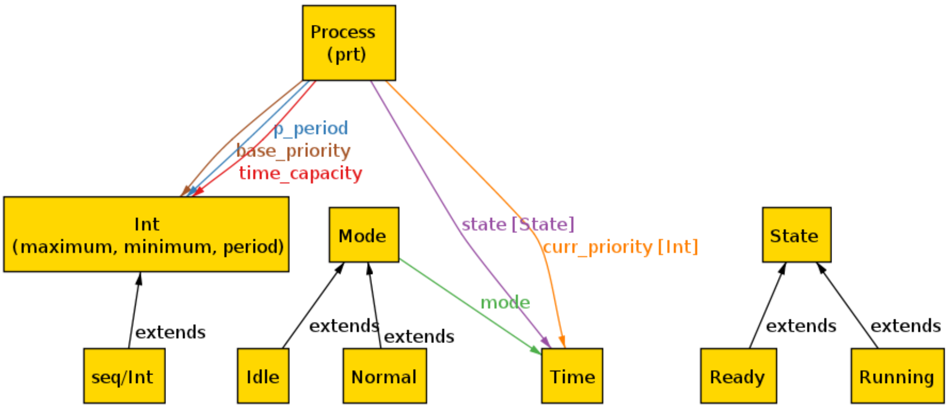


Figure 2.1: Requirement Metamodel.

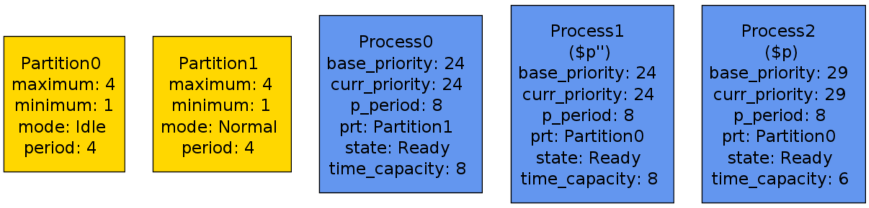
2.2 Processes Scheduling Model

Requirement Number	PRC#013
Title	Processes Scheduling Model
Description	The main characteristics of the scheduling model used at the partition level shall be: • One of the main activities of the O/S is to arbitrate the competition that results in a partition when several processes of the partition each want exclusive control over the processor. • Each process has a priority. • The scheduling algorithm is priority preemptive. If several processes have the same current priority, the O/S selects the oldest one. • Periodic and aperiodic scheduling of processes are both supported. • All the processes within a partition share the resources allocated to the partition.
Rationale	Scheduler has two levels, the system level in which the scheduler sees only the partitions and not the processes inside the partition, and the partition level, in which the scheduler sees the processes inside the partition.

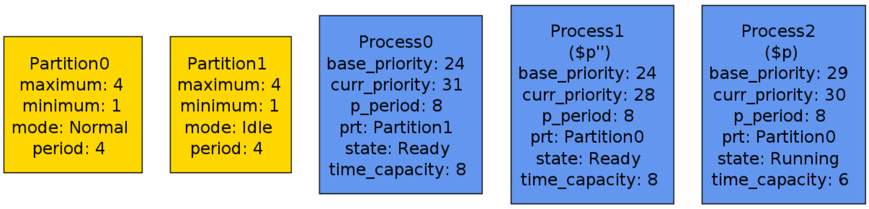
2.2.1 Alloy

```
pred re2ru[p : Process, t:Time]{
  p·state·t = Running
  p·state·(t·˜prox) = Ready
}

pred ru2re[p : Process, t:Time]{
  p·state·t = Ready
  p·state·(t·˜prox) = Running
}

pred HasMaxPriority[p:Process, t:Time]{
  all p':Process | p'·prt=p·prt ⇒ lte[p'·curr_priority·t,p·curr_priority·t]
}

fact Scheduler{
  //Partition
  all t : Time | lone (mode·t)·Normal
  all p : Partition | some t : Time | p·mode·t = Normal
  all p : Partition, t : Time | i2n[p,t] ⇒
  {
    (some t' : Time | eq[diff[t,t'],p·period] and i2n[p,t'])
    &&
    (all t' : Time | lte[diff[t,t'],p·minimum] ⇒ p·mode·t'=Normal)
    &&
    (all t' : Time | gt[diff[t,t'],p·maximum] and lt[diff[t,t'], p·period] ⇒ p·mode·t'=Idle)
     &&
    (all t' : Time | gt[diff[t,t’],0] and lt[diff[t,t’],p·period] ⇒ not i2n[p,t’])
  }

  //Process
  all t: Time | lone (state·t)·Running
  all p : Process | some t:Time | p·state·t = Running
  all p : Process, t:Time | p·state·t = Running ⇒ {
    p·prt·mode·t=Normal && HasMaxPriority[p,t]
  }
  all p : Process, t:Time | re2ru[p,t] ⇒ {
   (some t’:Time | eq[diff[t,t’],p·p_period] && re2ru[p,t’])
   &&
   (all t’:Time|gt[diff[t,t’],p·time_capacity] && lt[diff[t,t’],p·p_period]⇒ p·state·t’=Ready)
   &&
   (all t’:Time | gt[diff[t,t’],0] && lt[diff[t,t’],p·p_period]⇒  not re2ru[p,t’])
  }
}
```

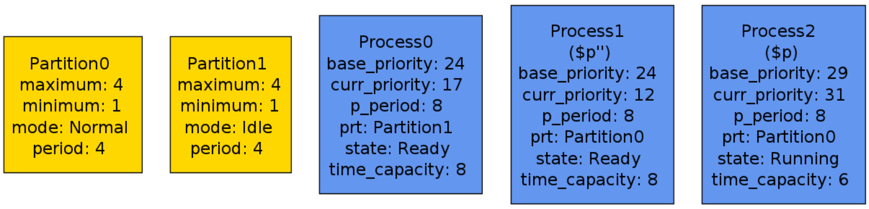
A.1 Scheduling Instance



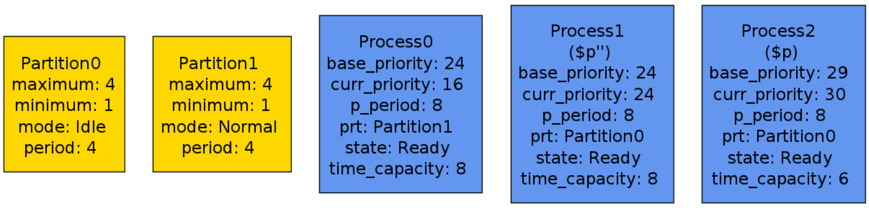
(a) *Time 0*



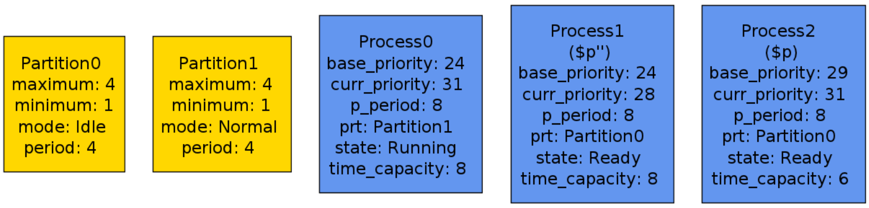
(b) *Time 1*



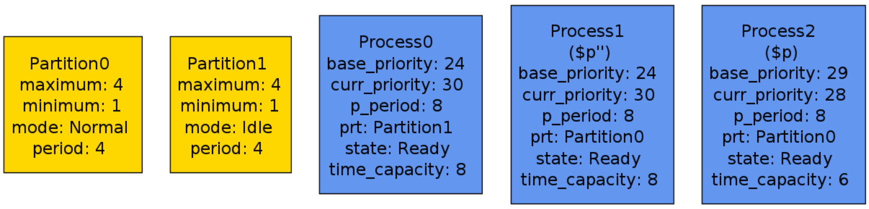
(c) *Time 2*



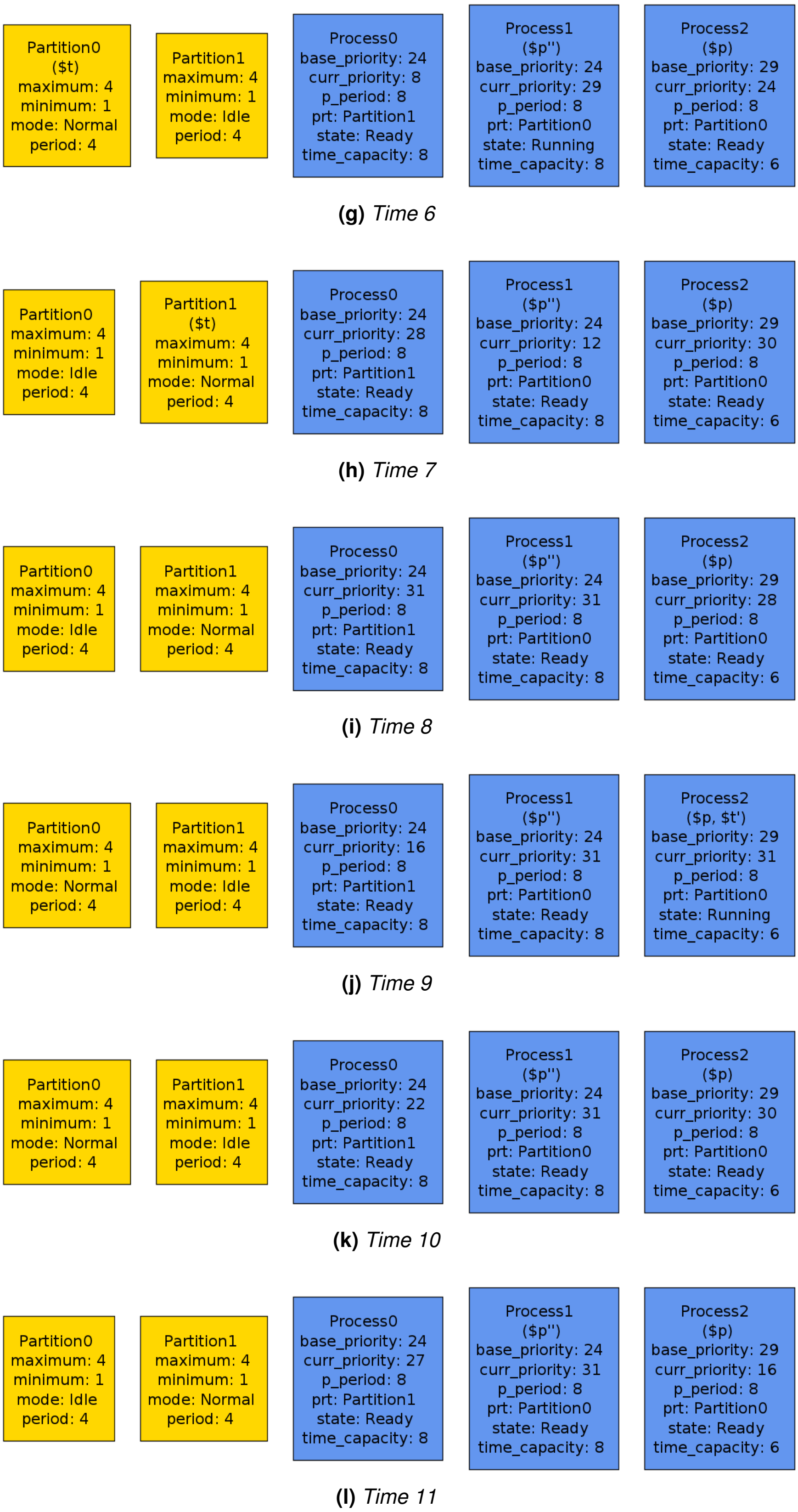
(d) *Time 3*

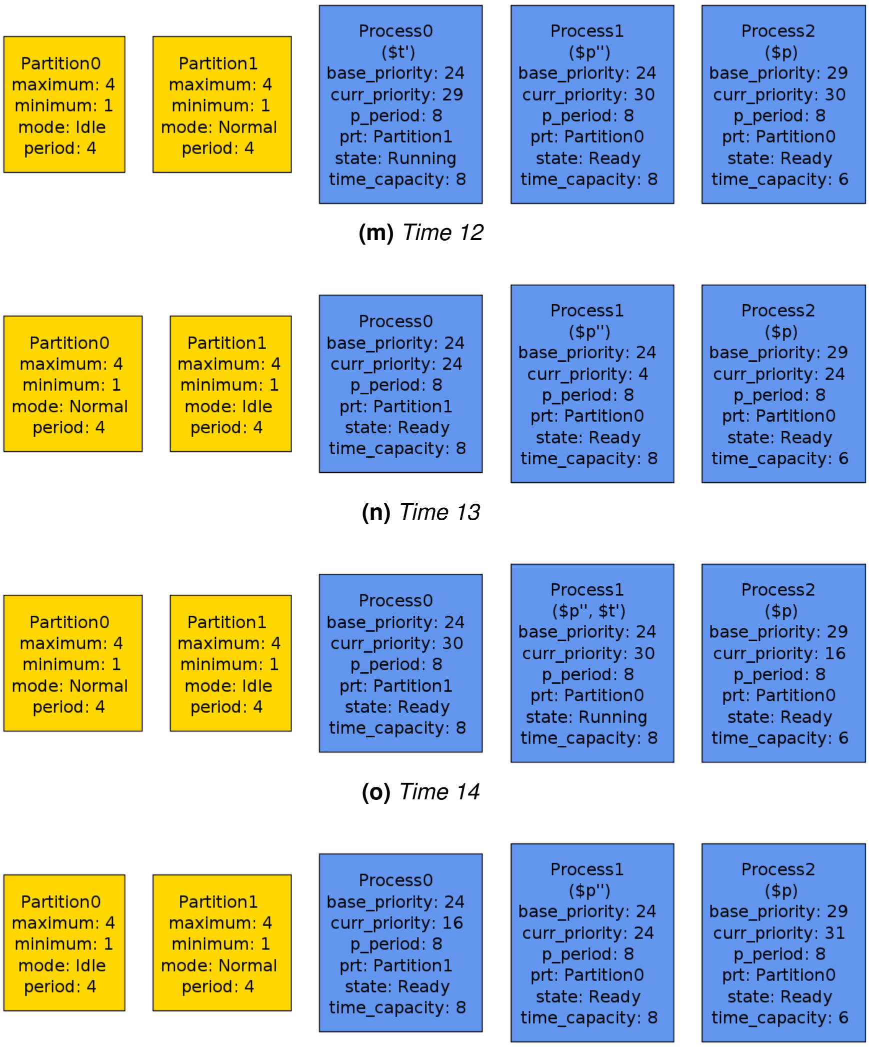


(e) *Time 4*



(f) *Time 5*





(m) *Time 12*

(n) *Time 13*

(o) *Time 14*

(p) *Time 15*

Figure A.-1: *Partitions Evolution over Time*