Modelling and Transformation of Non-Functional Annotations

DISSERTATION

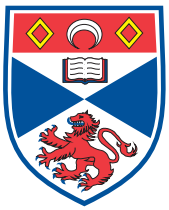
submitted to the

University of St. Andrews

for the degree of

Master of Science

by
Jorge Ibarra Delgado
jid@st-andrews.ac.uk



University
of
St Andrews

St. Andrews
Fife
Scotland
http://www.cs.st-andrews.ac.uk
2015-06-15

Abstract

A model is an abstract description of an artefact, we can obtain the key elements of a new or existing system and show these to the corresponding stakeholders in order to evaluate and validate our requirements, to verify our implementation and to have a rich documentation for further consultation. Model transformation is used to generate models of different kind, which can be an effective and useful resource when we are targeting an specific semantic model. One fundamental characteristic of model transformation is to use a formal target model which can be analysed and verified formally. Examples of such target models are different kinds of automata and Petri-nets. In this project we want to capture the specification of non-functional requirements, e.g. reliability, security, safety, availability, etc. into one model, like UML and transform this model to a target model that can be related to an existing verification tool. Concretely, this project consists of three main aspects. First, we created a language based on PlantUML extended with non-functional annotations called PlantNF. Second, we created an editor for PlantNF. Third, we developed a model-transformation tool from PlantNF to timed automata, the input language of UPPAAL. In this way our PlantNF language can be verified using the well known tool for timed automata, namely UPPAAL.

I declare that the material submitted for assessment is my own work, except where credit is explicitly given to others by citation or acknowledgement.

To all of the friends I have met in Nancy and St. Andrews.

Contents

Chapter 1

Introduction

1.1 Overview

In our everyday life, we can observe that we are surrounded by technological products which improve our quality of life. For instance, these products allow us to communicate with people at any time from any point in the globe using audio and video channels; travel all around the world using air and land vehicles with a vast range of integrated sensors that allow to read and process specific data, giving us the certainty that we can have a safe and reliable trip; the optimal, safe and secure operations of our power plants and even extend our lifespan by using pacemakers.

All of these products are composed of embedded software controls [36] [18], where each component communicates, shares, processes and stores data with other components, creating a complex network of systems of systems. The constant expansion of these products has produced an explosion in their diversity and a shorter and shorter time-to-market window, this indicates that a proper architecture is needed to construct these systems, this architecture must implement the principles of separation of concerns [33], also known as information hiding, abstraction or data hiding. Concurrency is likewise a vital piece in our scope and must be considered in both hardware and software. We as well require all stakeholders [35] to have a proper understanding of what we want and how we want it. Last but not least, we want our architecture to be reusable for upcoming projects that can share similar requirements.

As a part of the production of our architecture, it is necessary to rely on the creation of models that can represent our systems from different points of view. These models are part of the Views and Viewpoints [38] of the architecture which currently some sets are recognised and used in industry like 4 + 1 [26] and Views and Beyond [22]. Within these models we can determine how our hardware and software components will interact with each other, giving us a clear idea of what we expect the system to do. We can consider two ways to model our systems:

- **Informal models**: We can consider into this category elements like boxes connected by lines or the Unified Modelling Language [16]. Their characteristic is that they are reasonably applicable to software architectures and very easy to learn and apply, but because of their expressiveness, there is room for ambiguity and misunderstanding on stakeholders' interpretations. Actually, according to [41] it can force developers and organizations to operate in a way that fits the approach instead of making the approach fit the people.
- **Formal models**: These are constructed by applying semantical rules, giving us the possibility to execute validations on the construction of our models and also check the soundness of their different levels of abstraction. Their design requires more intellectual effort in comparison with informal models, that is why usually they are not used in the whole of a system but in critical sections or parts of the solution, where the specifications require the system to achieve one or many dependability attributes. However they have the advantage that they can be

computable and can give us the certainty that their soundness is founded on mathematical and logical principles. Some languages and tools available are: Event-B, which is a language that specifies models from the point of view of reactive components and the events raised through the stimulus provided, built from a very simple - abstract viewpoint to a granular complex system of systems [5]; and UPPAAL [11], an integrated tool that creates models of real-time systems as networks of timed automata.

1.2 The project and contributions

Related to the topic of system modelling, in this project we want to specify a choice of non-functional properties of a system i.e. performance and/or dependability attributes such as reliability, security, safety, etc) as model annotations and implement a model-driven transformation between the annotated model and the underlying model in an analysis tool, which in this case will be UPPAAL. The idea is that different kinds of annotations can be mapped onto different formal models for different analyses.

In model-driven engineering, models and model transformations play essential roles in software development.

There are many forms of model transformations including model-to-text transformation to generate code from models, text-to-model transformations to parse textual representations to model representations, model extraction to derive higher-level models from legacy code, and model-to-model transformations to normalise, weave, optimise, simulate and refactor models, as well as to translate between modelling languages. Model-to-model transformations can be particularly useful in formal analysis where the target model is a semantic model, and hence provide a link between informal notations more common in design with analysis tools for automatically verifying the semantic model.

The work done here had the objective to find out one of several ways to model non-functional requirements in a software engineering model like our target UML. And then decide for a semantic model (in this case UPPAAL) and perform a transformation that allows us to select elements of the source model and map them into the chosen target.

The next set of targets were proposed during the project's planning:

- Primary:
  - Explore the use of PlantUML [3] a simple and easy-to-learn modelling language that provides the necessary elements to construct use case, sequence, class, activity, component and state UML diagrams using textual notation. This language can be adapted to be able to create diagrams containing timing constraints as a minimum.
  - Once this language adaptation has been created, we want to perform a transformation directly to UPPAAL, which we are using as our target tool for model verification and validation.
- Secondary:
  - Find a match between the language to be created and the Modelling and Description Language for Stochastic and Timed Systems (MoDeST)[15], a high-level modelling language that enables elements well known from object oriented programming and procedural programming as exception handling, recursion and parallelism. This language has also implemented a connection to UPPAAL via mctau[14], that translates the models written in MoDeST to the XML-based language used for the UPPAAL integrated toolset.
  - In order to do this, we consider to do an extension of our previously defined grammar to include probabilistic statements.

  - Implement a solution that can be capable of performing these transformations.

- Tertiary:

  - Demonstrate the results on a case study of choice.

As a result from working on this project, we constructed, in effect, a language based on the grammar of PlantUML that can remain very simple if we only want to use it to model automata, however it integrates the sufficient elements to generate timed automatas. Then we created a solution that in it's first stage could make the transformation from our language to UPPAAL at its 4.0 version. Later, we added to our language the stochastic elements from MoDeST and we also updated our transformation tool so its output could be read by the version 4.1 of UPPAAL, which also integrates probability elements thanks to its Statistical Model-Checking extension.

As an extra effort, we worked on a tool to provide an easier way to write the script based on eclipse using the xtext [13] Eclipse plug-in. This a framework for the creation of programming languages and domain specific languages (DSL).

1.3 Outline of this work

Chapter 2 will provide more background information on the topic of embedded real-time systems, specially we will make a comparison and discussion between these systems and computer systems. We will also mention some characteristics and classifications of real-time systems. There will be also a section in this chapter dedicated to talk about system modelling. We will define concepts like formal system modelling, domain specific languages and model transformation.

In chapter 3 we will present a general overview of the solutions proposed by us and the influence taken from PlantUML MoDeST and UPPAAL. To illustrate how these integrated tools and languages helped us in the development of the language, we will use examples that at the same tame will provide us an understanding on the differentiation between formal and informal system modelling.

The next two chapters will discuss the architecture and implementation of our project, first we will introduce PlantNF, the language we created, and we will present its components as well as the implementation of our editor tool for the language. For our second solution, a model transformation tool, which will help us get our modelling language specification to a proper UPPAAL script, ready for verification, will be presented in chapter 5. We will present the architecture of the system, and implementation bits of the system as well as instructions on how can we use the solution for further work and demonstration.

The evaluation of our approach for transformation will be the subject of chapter 6. Finally, chapter 7 closes this project with conclusions and discussion on what could be scoped as additional work.

Chapter 2

Background and Related Work

2.1 Embedded Systems

Usually, as stated in [30], "a computer system works having in mind the notion that software is the realization of mathematical functions as procedures". This means that we give a certain input data and this will map into an output of data. The way in which we obtain the results is not as important, we could process the input with a high-level language like Java, Python or C# or even use assembly code and we could get the same result. The significance of the software is not affected by the mechanism.

However, Embedded software is not like that. Its main role is not the storage, writing, processing and modification of data, but the interaction of system's components with the physical world. It is executed in machines that are not exclusively computers operating in offices or racks of servers. They are cars, airplanes, telephones, audio equipment, robots, appliances, toys, security systems, pacemakers, heart monitors, weapons, televisions, printers, scanners, climate control systems, manufacturing systems, an the list can go on and on [30].

Then, as we may imply by the facts aforementioned, software that interacts with the physical world will require to take some properties from the physical world, for instance take account of time, power consumption and the fact that a process keeps running, without terminating, unless the system faces an error.

From the point of view of computer scientists, interaction with the physical world can appear to be a complicated problem. The consequence of this is that embedded software design is not having the rich variety of benefits a computer system is having. Computer systems come nowadays with the advantage of writing code applying principles such as polymorphism or generalization, used in object oriented programming; or use the well known automatic memory management used in the Java Virtual Machine or the Common Language Runtime; processors that calculate automatically the best management of resources for the average case scenario. However, many specialized engineers are still writing assembly code for individual digital signal processors that, as an advantage to the work they need to do, filter instructions in a deterministic way, in other words, they know how much cycles a set of assembly code are going to take.

We have to take account that "engineers that write embedded software are rarely computer scientists" [30]. They are experts in their domain and they understand the behaviour of the physical world and the interaction with the systems they are designing. These domain professionals can't afford for instance a Java program taking valuable time performing garbage recollection and user interface updating, for them this could mean a possible catastrophe that could incur in the loss human lives.

Yet, the complexity of emerging embedded applications is incrementing in a fast pace, some of these applications integrate elements from general computing, but still reliability is still having high standards, contrary to common, general-purpose software. There is a need to provide engineers with

tools that can give sufficient help to deliver robustness and at the same time embrace physicality as well.

The following concepts, obtained as well from [30] are vital to get into account in order to elaborate a proper software system, these properties originate on the reasoning that only having a result from an input and output is insufficient.

- **Timeliness**: In traditional computing, the measure of time is not reliable. As said before, modern processors operate with diverse mechanisms that make the life of embedded systems designers more difficult because these techniques are focused on the average case performance, which is not what real-time systems designers could consider as reliable. There is actually an active research on obtaining the worst case execution time [42]. We can also observe that even temporal logics consider time as a qualification instead of quantifications on time, expressed by words like "Eventually", "Generally" or "Always" [27] and we have to remember that embedded systems work on the physical world, where time is quantifiable, and the execution of systems must be quantifiable as well.

- **Concurrency**: Embedded systems are very likely to be concurrent systems because these systems are most of the time interacting with other systems cooperating, synchronizing data manipulation in a network of components. The chances are few where we could see a system of this nature interacting only with a single physical instance.

- **Liveness**: This is a critical issue for Embedded systems, Liveness is about achieving that the program have the capacity to keep operating continuously, without terminate or block waiting from some other input or event that will never happen to occur. As traditional computer scientists we can see that this is the contrary case from the point of view of a computer software system, where if the process keeps looping or non-terminating we might have a defective a defective class or program. However, in Embedded systems, terminating is considered as defective. The term deadlock is actually a well known coined term that describes occasions where there is premature termination of a system and it has to ve avoided at all costs.

  As an additional point, we cannot determine that our process is correct by only getting a right final answer. In an embedded system we have to take account of the temporal occurrences of diverse partial answers, which will lead us to the success of satisfying the non-functional properties of our system. These Non-functional properties include timing, power consumption, fault recovery, safety, security, availability, reliability, resilience and robustness.

- **Interfaces**: Thanks to the distribution of the object oriented design approach, software engineering has witnessed a great improvement. Using this paradigm we can see the different parts that make our system a whole, but we hide the "how" of the operations of our components by connecting them and establishing communication through the exposition of their interfaces. By selecting carefully the right choice of architectural styles [9], a combination of design patterns [32] and object modelling [19] type systems, adding the possibility to rely on the use of polymorphism and generalisation, we can have the necessary to create large, well established structures of code.

  However, a problem that has been pinned in object-oriented programming is that it doesn't specify how exactly concurrency and other dynamic structures will operate. To allow embedded software benefit from a component technology, this has to be aware of the dynamic properties once we define our interfaces [30].

- **Heterogeneity**: This concept plays a vital role in embedded systems because there exists a combination of computational styles and the technologies that implement these different components. This concept also refers about the combination of software and hardware systems, interacting with each other using specific technologies depending on the type of task the systems are performing

There is also heterogeneity in the form that embedded systems handle the events they react to. Depending if they are irregular events in time, like alarms or sensor triggers; or regular events, like the ones obtained from regular timed sampling devices, the events have different thresholds for reaction.

Most of engineers and computer scientist have a tendency to seek for a complete and extensive - unifying model, one such that could explain everything. We can find proposers claiming that their language or model is the right one, the silver bullet that will solve all the problems. Curiously, we find awkward that we have to combine multiple programming languages, despite the fact that it happens all the time. And, effectively, for embedded systems, this kind of vision is not fitting since different tools are bound to fit better or worse for distinct problems.

- **Reactivity**: Reactive systems are "those that react instantaneously and constantly to the environment they interact" [30]. They distinguish themselves as other two type of systems:
  - *Interactive systems*: [12] defines interactive systems as programs that interact at their own speed with users or with other programs; from a user point of view, a time-sharing system is interactive. The difference with reactive systems is that their interaction with the environment is made with a speed determined by the environment, not by the program itself. Interactive programs work at their own pace and mostly deal with communication, while reactive programs only work in response to external demands and mostly deal with accurate interrupt handling.
  - *Transformational systems*: According to [23], a transformation system is the one that accepts inputs, performs transformations on them and produces outputs, perhaps prompting a user from time to time to provide extra information.

  Reactive systems have real-time constraints, and are frequently safety-critical, to the point that failures in these systems could result in the loss of human life. Another characteristic of reactive systems is that, as discussed before, they have the property of liveness.

  A robust reactive system must be capable of adapting to changes. These changes can come from alterations in requirements, processing power or a change in the configuration or quantity of sensors or new standards coming from quality of service. The system then has to be available while changes in design are performed, and, of course, failure is not an option.

2.2 Real-Time Systems

Embedded systems, as reactive systems, are considered real-time systems, if we look at a couple of definitions that we can find in [29] we can recall some of the discussion from the properties mentioned before:

- "A real-time system is a computer system that must satisfy bounded response-time constraints or risk severe consequences, including failure". Where a failed system is a "system that cannot satisfy one or more of the requirements stipulated in the system requirements specification".
- "A real-time system is one whose logical correctness is based on both the correctness of the outputs and their timeliness". This definition exhibits the common theme among all definitions of what is "real-time", which is that the system must satisfy deadline constrains in order to be correct.

Events are commonly at the order of the day in real-time systems. An event [29] occurs when there is a change in the flow of control of the sequence of a program. E.g., case, if - then and while statements in any programming language are elements that are interpreted as a possible change in

flow of control. If we take it from an object oriented point of view, an event is triggered when the instantiation of an object or the calling of a method changes the sequence in the flow of control.

We can distinguish two types of events:

- Synchronous events: they occur in a time-predictable manner in the flow of control. A clear example of these can be found in conditional branches or by internal interrupts. We can anticipate when these events may occur.
- Asynchronous events: they happen in a time-unpredictable way in the flow of control and are caused by external stimulus. We can exemplify this by the situation where a user clicks the mouse button. The program listens to the action of mouse clicking getting the information brought by the mouse input and reacts accordingly to the specified time and requirements.

We can also recognise three different types of real-time systems according to their tolerance to failures:

- Soft Real-Time Systems: These are systems where performance can decline but they can't be destroyed by failure to meet response-time constraints.
- Firm Real-Time Systems: We can identify a firm real-time system if the system can withstand some failures meeting timing constraints but the obtained computations are futile [25]
- Hard Real-Time Systems: Also known as safety-critical real-time systems, these are applications that cannot tolerate failure, because the outcome of missing a single deadline may lead to a complete or catastrophic system failure.

The table 2.1 gives a couple of examples for each of the different types of real-time systems.

2.3 Dependability

Since this project is focused significantly into hard real-time systems, it is pertinent to define the concept of dependability and the properties that surround this term.

According to [7] Dependability is "the ability to deliver service that can justifiably be trusted". Another definition given by the same author and that aggregates an extra guideline is that dependability is "the ability to avoid the failures that are more frequent and more severe than is acceptable".

However definitions coined in standards differ from that definition, in these we can see a clear reference to availability:

- "The collective term used to describe the availability performance and its influencing factors: reliability performance, maintainability performance and maintenance support performance" [1] .
- "The extend to which the system can be relied upon to perform exclusively and correctly the system task(s) under defined operational and environmental conditions over a defined period of time, or at a given instant of time" [2].

Wrapping things up around the different definitions, we can perceive dependability as the capacity of a system to perform correctly and according to what the system stakeholders required. This dependability is composed by other terms [8] that we have to define:

- **Availability**: The ability of the system to deliver services when requested. Availability is measured as the limit of the probability that the system is functioning correctly at time t, as t approaches infinity. This is the steady-state availability of the system. It may be calculated [39] as:

System	Real-Time Classification	Explanation
The cooling system of a Nuclear Power Plant which must detect the sudden increment of temperature in the reactor.	Hard	Missing the deadline to activate the cooling system as soon as the temperature rising is detected may cause the meltdown of the containers, having as a consequence a devastating environmental damage and possible loss of human lives
Pacemaker		Missing the deadline to deliver the electrical pulses to the muscles according to the patient's needs may cause a heart stroke and death
Navigation controller of a floor cleaning robot	Firm	Missing a few navigation deadlines can cause the robot to get outside of its planned route, keeping some areas unclean and risking damage of the machine or surrounding objects
Prototype four legged robot		Missing some of the deadlines provokes the robot to move or stand in an inappropriate manner, probably causing some damages to the robot and the test environment
Console football game	Soft	Missing even several deadlines will only degrade performance
Online video streaming		Missing deadlines will only cause the video to stop

Table 2.1: Samples for Hard, Firm And Soft real-time systems, some of them taken from [29]

$$\alpha = \frac{MTTF}{MTTF + MTTR}$$

Where MTTF is the mean time to failure, and MTTR is the mean time to repair.

- **Reliability**: The capacity of the system to give services as specified. It measures the capability the system has to operate over time and it is usually measured as its mean time to failure or the expected life of the system. We can identify three types of reliability in a system from a socio-technical viewpoint :

  Hardware reliability: What is the probability of hardware components to fail and how long does it take to repair that failure?

  Software reliability: How likely is that a software component will bring an incorrect output? A clear difference we can notice between software and hardware reliability is that software does not wear out. The system can continue working after the error has been produced.

  Operator reliability: How likely is that the operator of the software will make

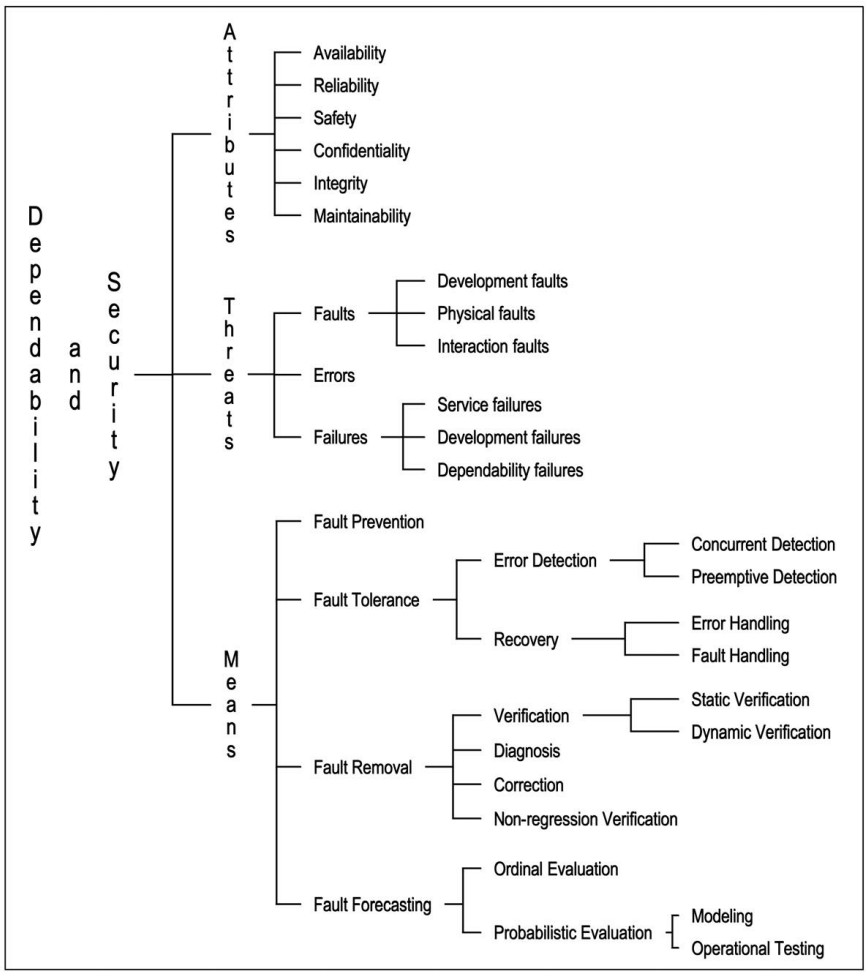


Figure 2.1: Dependability and security tree, taken from [7]

an error because of a mistaken input? How possible is that the software can ignore this error, having as a consequence the propagation of the mistake? [37]

System reliability and availability are closely related properties that can be expressed as numerical probabilities [37]. Availability is the total measure of time the system is available and the reliability is measured as the rate of occurrence of failure. As an example, if for every 100 we have 5 incorrect outputs, our rate of failure or reliability is of 0.05. If a system has an Availability of 0.99 this means that over some time period the system is available 99% of that time.

Depending on the required system availability can be more important than reliability or vice versa. For instance, a banking web application which has to be available 24 hours a day, but it can tolerate multiple failures and it can recover quickly. The users in general won't be affected by the low reliability.

- **Safety**: The absence of catastrophic consequences on the users and the environment. the Table 2.2 contains some of the most common terms used for safety.

Term	Definition
Accident (or mishap)	An unplanned event or sequence of events which results in human death or injury, damage to property, or to the environment
Hazard	A condition with the potential for causing or contributing to an accident
Damage	A measure of the loss resulting from a mishap. Damage can range from many people being killed as a result of an accident to minor injury or property damage
Hazard Severity	An assessment of the worst possible damage that could result from a particular hazard. Hazard severity can range from catastrophic, where many people are killed, to minor, where only minor damage results. When an individual death is a possibility, a reasonable assessment of hazard severity is 'very high'.
Hazard Probability	The probability of events occurring which create a hazard. Probability values tend to be arbitrary but range from 'probable' to 'implausible'.
Risk	This is a measure of the probability that the system will cause an accident. The risk is assessed by considering the hazard probability, the hazard severity, and the probability that the hazard will lead to an accident

Table 2.2: Terms related to safety, from [37]

- **Integrity**: The absence of improper system alterations. This is related to a security policy that defines which data should be modified by whom, which will bring a clear idea of unauthorized modification. Integrity is usually measured by the time and resources it would take an intruder to modify data or processes without authorization. However, the measures depend on the type of cryptographic protocols taken.

- **Maintainability**: The ability to undergo modifications and repairs. MTTR is the quantitative measure of maintainability. In spite of that, there are some aspects we have to take care of, like the politics of maintainability of the system, which can increase considerably the mean time to repair. As an example, if a system hardware component fails, the software of the system can send automatically a report to the manufacturer, which can then process the report and then send a replacement of the faulty piece to the user. Once the user has possession of the piece he can himself make the necessary adjustments to the system to get it back to operation.

- **Security**: The ability of the system to protect itself against accidental or deliberate intrusion. Traditionally, security issues are related to databases, where the concerns are confidentiality, authenticity and privacy of information. Security is essential as most systems are networked, meaning that systems are allowed to be accessed from the Internet.

2.4 System Modelling

System modelling is the process of developing an abstraction of a system. This abstraction can and is recommended to be done with multiple views or perspectives of that system. Usually, in industry system modelling is done by using graphical notations, the most known being the Unified Modelling Language (UML). However, there exist the possibility to develop formal, or mathematical, models.

During the requirement elicitation process, models are useful to obtain and explain the requirements of a system. For the design process, modelling is necessary to describe the system to the people that will be in charge of developing the system. For the stage that follows implementation it is necessary for documenting the architecture of the system [37].

One important thing about system modelling is its capacity of abstraction, i.e., to put out detail, simplifying the most important characteristics of the system. Nonetheless, this does not mean that we are making an alternative representation of our new solution or existing system under analysis,

Term	Definition
Asset	Something of value which has to be protected. The asset may be the software system or data used by that system.
Exposure	Possible loss or harm to a computing system. This can be loss or damage to data, or can be a loss of time and effort if recovery is necessary after a security breach.
Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm.
Attack	An exploitation of a system's vulnerability. Generally, this is from outside the system and is a deliberate attempt to cause some damage.
Threats	Circumstances that have potential to cause loss or harm. It can be considered as a system vulnerability that is subjected to an attack.
Control	A protective measure that reduces a system's vulnerability. Encryption is an example of a control that reduces a vulnerability of a weak access control system.

Table 2.3: Terms related to security, from [37]

the model still reflects the system but only by the key characteristics we want our stakeholders to visualise and understand.

If we try to imagine a model of a complex system that contains all of its main characteristics we can come to the conclusion that this can be a bit difficult, and what could happen if we gather different stakeholders to evaluate the model, would be there a mutual consensus and understanding on the relation and configuration between the system components? Most of the times the answer is: no. To solve this, we can create diverse views depending on the type of stakeholders we want to reach. Every company has its own standard of set of views that can satisfy their needs. However, there are coincidences between the choice of models that, together, can provide enough properties of the system to let all stakeholders know what it is the system going to do.

One example of this set of views is 4 + 1 [26], which foundation is the concept of software architecture of Perry & Wolf [34]:

$$SoftwareArchitecture = \{Elements, Form, Rationale\}$$

where elements are the "building blocks" of a system, i.e., the components, containers and connectors, or the part that answers the "what"question. The form is related to the configuration or the pattern that the elements will take, it answers the "how" question. Finally, the rationale gives an explanation on "why" these elements are configured in that manner, it gives the stakeholders the detail that is not easy to capture by only looking at the elements and form.

Having this on mind, then we can proceed to explain the structure of 4 + 1 which, as we can see by its name, is composed of 5 views, these can be represented by annotated or graphical models:

- The logical view, which is the object model of the design (when an object-oriented design method is used).
- The process view, which captures the concurrency and synchronization of the design.
- The physical view, which describes the mappings of the software onto the hardware and reflects its distributed aspect.
- The development view, which describes the static organisation of the software in its development environment.
- The illustration of a few selected use cases or scenarios, making it the fifth view

Another observation we can make based on the survey made by [21] is that most users of the UML thought that only five diagrams would be enough to represent the essentials of the system:

- Activity diagrams, which show the activities involved in a process or in data processing.
- Use case diagrams, which show the interactions between a system and its environment.
- Sequence diagrams, which show interactions between actors and the system and between system components.
- Class diagrams, which show the object classes in the system and the association between system components.
- State diagrams, which show how the system reacts to internal and external events.

2.5 Formal System Modelling

The use of informal models like UML may satisfy the needs of many software development teams, specially for those that use agile methodologies. However, many researchers have advocated the use of formal methods of software development. These are mathematically-based approaches to software development where a formal model of the software is defined. If we model a system by using a formal method and we implement our system by following its guidelines, we can eliminate software failures resulting from programming errors.

To create this formal model we have to translate the requirement from the users, which are usually manifested in natural language, diagrams made of boxes and lines and integration of use cases or stories, into a mathematical language that has defined semantics. This specification is a description that does not let in any ambiguity. We can check the consistence between the specification and the model's behaviour by using manual or automatic tools. If we think about it, using this method can be more reliable to get a consistent system. If we use natural languages or informal methods, we can have some inconsistencies hiding behind the expressiveness or interpretation of the language.

Formal methods are targeted usually for plan-based software processes, where the system is specified in its entire form before development. But since it is an expensive process, the use of this approach may be limited to only those components that are critical to the system's operation. Also, since this is a highly technical approach, the use of formal methods is generally targeted to the technical stakeholders. This can be quite problematic at the time to decide to adopt this approach into a system specification for the project managers because the stakeholders that know what they want from the system (product owners and domain experts) cannot understand a formal specification and also it can be difficult to determine if the cost of using this approach is less than using a traditional one, including the consequence of future changes, produced from system consistency repairing.

We can run tests on these models based on non-functional properties that we want to make certain our modelled system is capable of perform them. To do it there are specialised programs where we can introduce our model and perform the necessary queries and expect results. This is known as model checking.

One example of a formal model language and its corresponding model checking tool is Event B, a language that is used to construct the model of complex and discrete systems. Complex as the definition given in our Chapter 2 and discrete, defined as a system that can be abstracted as a succession of states altered by events that can cause the change of state. According to [5] these systems fall into the generic name of *transition systems*.

Event B works on an iterative - incrementally basis. Having enough requirements, we can set a very general abstraction of our system with the pre and post conditions to make the transitions of our system. We have also to put invariants and rules that constraint our model. Then when we have our first iteration ready, we can add more requirements to our specification. These new requirements will be integrated in a refinement of our original abstraction. The rules of this refinement have to

be consistent with the rules stated in our first iteration. We keep refining until we get the desired specification.

Once the model has been specified, we can use the model checking tool of Event-B, which can be found in the Rodin platform [24]. There, we specify properties we want our model to comply with and check if it can meet them.

2.6 Model Transformations

Model Transformations are mappings of one or more software engineering models into one or more target models. They are a key element in the model-driven software development. According to [28] many notations and tools have been created, but there is not an agreement on which model transformation specification to apply to which type of model transformation problem.

There exist three considerations on approaching a model transformations at the semantic level, these are:

- The languages the transformation operates upon.
- Horizontal versus vertical transformation (if the abstraction level is changed or not respectively).
- The level of automation and complexity of the transformation, and the semantic correctness.

[28] consider three categories of model transformations:

1. **Refinements**: Refine models towards implementations, like Platform Independent Model to Platform Specific Model transformations in the Model Driven Architecture. We can also include here the code generation from a model. These are usually exogenous, i.e. the source language is different from the target language and use a vertical approach.
2. **Quality improvements**: They improve the structure and organisation of the model. They normally operate in a single language, which make them endogenous, semantically preserving-horizontal transformations.
3. **Re-expressions**: These transform the model to a nearest equivalent, used in cases of migration, re-engineering or validation. They are exogenous, semantically preserving and horizontal transformations.

Transformation correctness depends on the following concepts [40]:

- Syntactic correctness: The target model of the source language must be syntactically well-formed. Also the semantic constraints of the target language are required to be satisfied.
- Definedness: The transformation is validly defined on each source model.
- Uniqueness: The transformation produces a unique result from a given starting model.
- Completeness: For individual rules means that every aspect of the target model that is automatically inferred must be explicitly specified. For entire transformations it means that each relationship can be established by a composition of transformation rules of the transformation.
- Semantic correctness: For each property of the source model, the target model satisfies the property.

Chapter 3

Project Description

3.1 Project Overview

In this chapter we are going to describe the general guidelines of our entire proposed solution. This solution is composed of:

- A modelling language, called PlantNF, that is helpful to describe systems that require to introduce non-functional requirements in the form of timed automata. As a part of this part of the solution, we started to build an editor based in our grammar and constructed with the help of a tool called xtext. Both the grammar and the tool used will be described in chapter 4.
- A model transformation tool that takes as input a file written in PlantNF language, takes the elements necessary from that script and gives a XML file that UPPAAL can read as an output. The architecture of that solution and the implementation process of the tool will be described in chapter 5

We are also going to explain the languages and integrated tools that we are using to base our project. These are PlantUML [3], UPPAAL [11] and MoDeST [15].

3.2 Plant UML

PlantUML is a language that provides the necessary elements to write different UML diagrams. It contains also a tool that creates a graphic model of sequence, use case, class, activity, component, state and object diagrams. We are not going to provide an explanation of every diagram because we consider it is out of scope, however, we are going to focus on the creation of state diagrams, since their elements are the closest to those created by the timed automata of UPPAAL, which will be explained in its respective section. To create a state diagram in PlantUML we use the instructions located in Listing 3.1.

```
1  @startuml
2
3    Closed : The door is closed & is not in use
4    Heating :
5    Opened :
6
7    [*] --> Closed
8    Closed --> Heating : set timer and press start
9    Heating --> Closed : finish heating or press stop
10   Closed --> Opened : press open
11   Opened --> Closed : close door
12 @enduml
```

Listing 3.1: Example with the basics to create a state diagram in PlantUML

The code contained in the listing contains the following elements:

- **@startuml & @enduml**: these components describe the beginning and the end of a PlantUML script.
- Transitions made with the '- ->' symbol. This transitions can contain a label that is written after a colon.
- States that can be declared in a transition or previously, like it is made in Listing 3.1 we can also put labels inside the states by using colons.

The resulting diagram from the code mentioned before can be seen in the Image 3.1. As we can see, we only need some lines of code to create a nicely designed diagram. When using the eclipse plug-in for PlantUML, the diagram is generated automatically as we write the diagram. In order to obtain the diagrams we have to install and have properly configured Graphviz [20], an open source graph visualization software that uses a scripting language to create diagrams with many features such as options for colors, fonts, line styles, hyperlinks, and custom shapes.

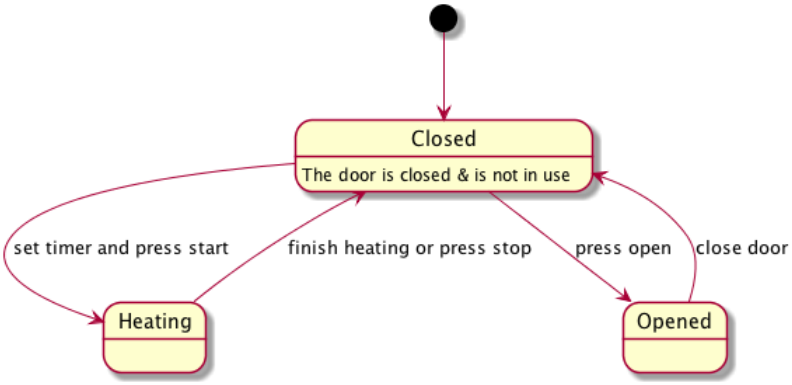


Figure 3.1: State diagram created using PlantUML

3.3 UPPAAL

UPPAAL is a tool suite to provide simulation and automatic verification of real-time systems. These systems are modelled as networks of timed automata [6].

A timed automata is an extension of a classical automata with a set of real-valued variables called clocks. These clocks increase their value as time goes by, this makes guards to satisfy the necessary conditions to have transitions and operations to be made. A timed automata is formed by six elements [17].

- A finite set of locations, also known as control states.
- An initial location.
- A finite set of clocks.
- A finite set of transitions.
- An invariant associated to each location.
- An alphabet of actions.

UPPAAL works using XML input files containing the necessary data to create the timed automata. In Image 3.2 we can see an example of the user interface and the diagram provided by the script provided in Listing 3.2

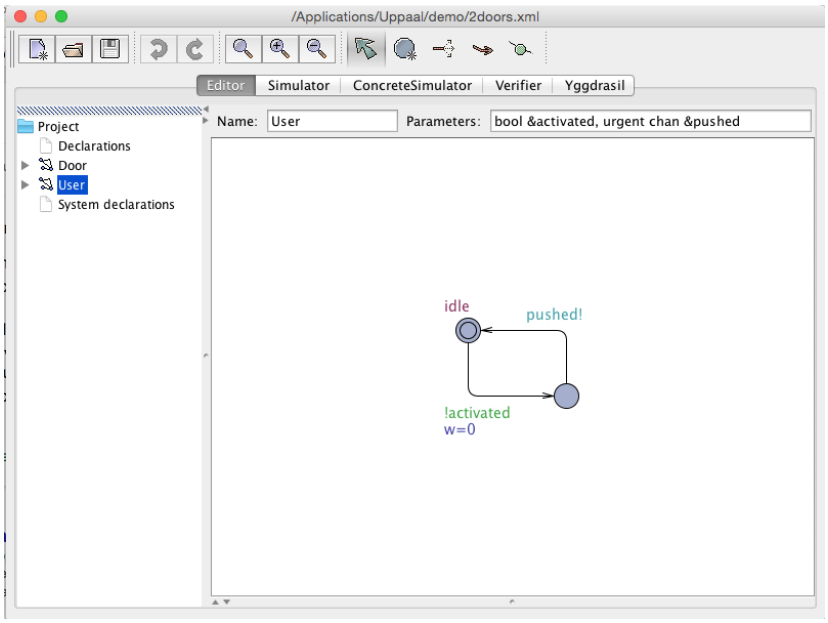


Figure 3.2: User interface and a little example of a timed automata

The meaning of each element and attribute in the XML code can be found in the subsection 5.2.3. There, we will link our meta-model with the modelling of UPPAAL diagrams.

```
<?xml version="1.0" encoding="utf-8"?>
  <!DOCTYPE nta PUBLIC '-//Uppaal Team//DTD Flat System 1.1//EN' 'http://www.it.uu.se/
     research/group/darts/uppaal/flat-1_2.dtd'>
  <nta>
    <declaration>
    </declaration>
    <template>
      <name x="32" y="16">User</name>
      <parameter>bool &activated, urgent chan &pushed</parameter>
      <declaration>clock w;</declaration>
      <location id="id6" x="192" y="160"></location>
      <location id="id7" x="96" y="96">
        <name x="72" y="64">idle</name>
      </location>
      <init ref="id7"/>
      <transition>
        <source ref="id7"/>
        <target ref="id6"/>
        <label kind="guard" x="72" y="168">!activated</label>
        <label kind="assignment" x="72" y="184">w=0</label>
        <nail x="96" y="160"/>
      </transition>
      <transition>
        <source ref="id6"/>
        <target ref="id7"/>
        <label kind="synchronisation" x="152" y="72">pushed!</label>
        <nail x="192" y="96"/>
      </transition>
    </template>
```

```
29      <system>
30        bool activated1, activated2; urgent chan pushed1, pushed2; urgent chan closed1,
        closed2; Door1 = Door(activated1, pushed1, closed1, closed2); Door2 = Door(activated2,
        pushed2, closed2, closed1); User1 = User(activated1, pushed1); User2 = User(activated2,
         pushed2); system Door1, Door2, User1, User2;
31      </system>
32    </nta>
```

Listing 3.2: XML code from an UPPAAL example

3.4 MoDeST

The Modelling and Description Language for Stochastic Timed Systems (MoDeST), is "a formalism that is aimed to support (i) the modular description of reactive system's behaviour while covering both (ii) functional and (iii) non-functional system aspects such as timing and quality-of-service constraints in a single specification" [15]. This language can provide us with the modelling of labeled transition systems, timed automata, Markov chains and decision processes. This language also presumes that the creation of such models can be done by programmers that are not experienced into formal methods. The reason of this is that the language is made with known constructs used in imperative languages, it also contains other elements from light-weight notations like modularization, exception handling , atomic assignments, simple data types and iteration. However, the semantics provide the basis for development of tool support and enables formal verification.

The following code, provided from the introduction of the language in [15], simulates a football match, this can help us understand how MoDeST works. The first code snippet declares the score and the players for each team.

```
int score[2];
int players[2];
players[0] = 11;
players[1] = 11;
```

Listing 3.3: MoDeST code example

One of the things that happens very often in a football game besides passing the ball is committing fouls to other players. In MoDeST this is done by using clocks that reset to 0 and the foul happens between an interval of 2 to 5 clock cycles. When the delay is achieved, the foul is allowed to happen in an urgent manner, meaning that it has to be executed at the latest.

```
process FoulPlay(){
  clock c;
  float delay;
  {= delay = UNIFORM(2,5),c=0 =};
  when (c==delay)
      urgent(c==delay)
          throw foul
}
```

Listing 3.4: MoDeST code example

Passing the ball is an activity were probability occurs, that's why the *palt* construct is used in the next snippet of code, describing a probabilistic choice between alternatives. The probability is given by weight expressions, which are mathematical expression embraced by colons in the code.

```
process Pass(int team){
do {
  :: kick palt {

    :0.9 . players[team]:
           self
    :0.9 . players[1-team]:
           other; break
```

```
    :0.1 . players[team]:
           {= score[team] +=1 =};
           goal; other; break
    :0.1 . players[1-team]:
           {= score[1-team] +=1 =};
           goal; break
    }
  }
}
```

Listing 3.5: MoDeST code example

In order to pass the ball, a team has to possess the ball, and then play again.

```
process Play(int Team){
  gotBall; Pass(team); Play(team)
  }
```

Listing 3.6: MoDeST code example

Next piece of code contains the behaviour of a team, which can be described as a parallel composition, this is done by calling the process Play and FoulPlay inside a *par* construct.

```
process Team(int team){
   par{
      ::Play(team)
      ::FoulPlay(team)
   }
}
```

Listing 3.7: MoDeST code example

The instantiation of the teams is done with the following set of code

```
process Team0(){
   hide{kick, self, goal}
       relabel {other, gotBall, foul}
       by {0to1,1to0,foul0}
         Team(0)
}

process Team1(){
   hide{kick, self, goal}
       relabel {other, gotBall, foul}
       by {1to0,0to1,foul1}
         Team(1)
}
```

Listing 3.8: MoDeST code example

Having defined the teams now it is possible to define a match, which contains the calling for both team processes to be executed in parallel. There is also the inclusion of a try catch block, very well known for object oriented programmers in Java or C#. In this case, our exceptional behaviour is a foul play, which in this case will cause a team to have a decrement on their players by 1 and resume the match.

```
process Match(){
    try{
       par{
          :: Team0()
          :: Team1()
       }
    }
    catch(foul0){
      players[1] -=1; Match()
    }
```

```
11      catch(foul1){
12        players[0] -=1; Match()
13      }
14  }
```

Listing 3.9: MoDeST code example

The game match is already defined, it contains what happens inside of the field, however, so far, the match never stops and there is the possibility of teams to achieve a negative number of players. To avoid these situations a Referee is instantiated which observes that, if the clock arrives to the 90 minutes of play it sends right away the order to stop de game. It also detects if one of the teams has totally depleted their players in the game.

```
process Referee(){
     clock  x = 0;
     par {
        :: when (x==90)
              urgent(x==90)
                 throw game over
        :: when(players[0]==0 v players[1]==0)
               urgent(players[0]==0 v players[1]==0)
                  throw noplayers
     }
}
```

Listing 3.10: MoDeST code example

Finally, all the specification of the match is defined in a parallel composition formed by the processes Match and Referee. They are inside a try catch block, so we can take care of the exceptions *game over* or *no players* brought by the referee. If there are no players left, the game stops. If the game is played for 90 minutes, the players then swap their team shirts.

```
try{
        par{
           :: Match()
           :: Referee()
        }
    }
catch(gameover){
        swap_shirts
}
catch(noplayers){
        tau
}
```

Listing 3.11: MoDeST code example

3.5 Description of the Project

As stated in the introduction of this chapter and because of the goals set in the planning of the project, there were two solutions created:

The first solution is a user interface that allows the edition of PlantNF script with code correction, created on a Eclipse platform using the xtext library. This user interface is a similar Eclipse platform, which has all the features of that kind of platform, however at this early stage it might be considered as maybe too much for a simple editor. This user interface objective is to allow future users of the project to create scripts in the language we designed in this project.

The other component is the tool that transform the language from PlantNF to an XML-ready to read file for UPPAAL in its version 4.0 or 4.1. In this iteration the software is working by using the command line.

As we can see in Figure 3.3, the intention of the current project development is to provide a PlantNF file that can be then transformed into a XML that UPPAAL will read. Once in UPPAAL we can verify the correctness of our model and validate that it works as we want.

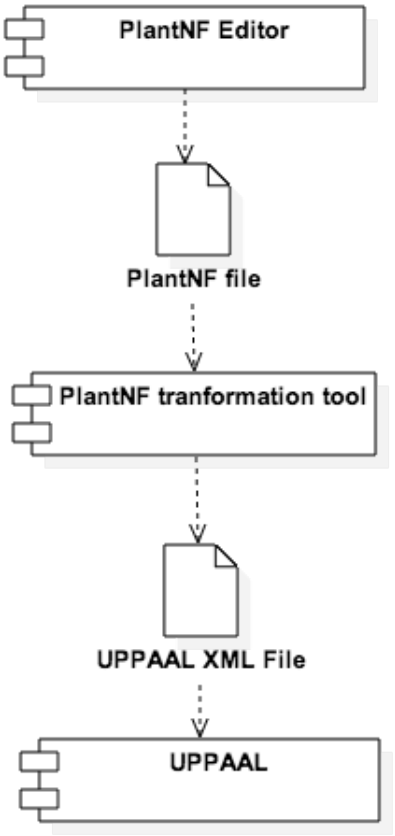


Figure 3.3: Component diagram displaying the artefacts we can obtain from each of the components made by us

Chapter 4

First Part: PlantNF

4.1 Introduction

The main subject of this chapter is to explain our contribution to the specification of non-functional requirements and model transformation area of research for model-driven development. Our project goal was to create a domain specific language that, as seen on prior chapters, could model some non-functional requirements, like availability or reliability. This semantic model could be then mapped into a target that could verify the correctness and that could be checked formally.

For this project we had the idea to take elements from PlantUML, which gives a very easy syntax to write UML diagrams, and then adjust the new grammar to make it coherent with our target language, which is UPPAAL. An extra feature is that we added some stochastic elements integrated in the latest beta version of UPPAAL and that also are integrated in MoDeST, as seen in chapter 3.

4.2 The Language

In this section we present the domain specific language created by us, called PlantNF. This is our third iteration of work into the language. In the first one, we created the rules that could satisfy a diagram that could be then transformed into UPPAAL v. 4.0. The second iteration integrated the definitions of branch transition, probability and exponential rate to the grammar. This was done by the influence of MoDeST and its work on mctau [14], a MoDeST-to-UPPAAL model transformation tool that is integrated in the MoDeST toolset. The third iteration added extra elements from UPPAAL that helped define in a coarse grained some sections of our grammar that were initially very abstract.

In the following set of code snippets we are going to explain the composition of our language.

Since our target language is the one used in UPPAAL, we will find in PlantNF a structure that is very similar to the resulting XML file. First of all, we start defining our script document as a composition of (i) a declaration, which in this first part contains *global* variables and clocks and it is mandatory to declare it. Then (ii) we have one or more templates, each one of these templates is a model of the actual timed automatas in the system. Finally (iii) a definition of the system model closes the file.

```
1 Document     ::= Declaration
2           Template+
3           SystemExt
```

In the next piece of code we will have our first interaction with the grammar of UPPAAL. Inside our 'Declarations' and 'EndDeclarations' terminals, we will use Declarations according to UPPAAL's grammar which can be a declaration of variables a type declaration, a function or a declaration of channel priority.

```
1 Declaration    ::= 'Declarations'
```

```
        Declarations
        'EndDeclarations'
```

As we stated before, a template is were we define the structure of our timed automatas. These automatas are formed by an ID terminal that defines its name. Then we have the option to incorporate or not parameters and declarations of *local* variables and clocks. As a part of our second iteration with the grammar, a set of branch points can be added into the automata, the branch is acting in a similar manner as a **palt** construct would act in the code of a MoDeST model specification. This will be explained further when we analyse the Label constructs of our grammar. Finally, a set of one or more transitions from one state to another finishes the formation of a template before closing our template with the 'EndTemplate' terminal.

```
Template    ::= 'Template'
        ID
        ParameterExt?
        Declaration?
        State+
        BranchPoint*
        Transition+
        'EndTemplate'
```

The next section of code is another UPPAAL exclusive targeted part. It consists of a system definition in which a system model is defined. A system model in UPPAAL consists of one or more concurrent processes, local and global variables, and channels.

As we have seen, global variables, functions and channels can be defined using the grammar for declarations. Even though the property of being global is part of these elements, they have a restricted access to templates. To be useful they have to be part of the arguments to the formal parameters of templates.

To define a system process in UPPAAL we have to use the definition given in the line 43 of Listing 4.1. This line contains a list of templates that will be instantiated into processes and that can be prioritised.

```
SystemExt     ::= 'System'
        System
        'EndSystem'
```

In the following construct, between the 'Parameters' and 'EndParameters' terminals we can have a set of parameters that can be declared in either call-by-value or call-by-reference manner. As used in C++, to make a call-by-reference the identifier must be prefixed with an ampersand symbol in the parameter declaration. A call-by-value parameter is excluded of having an ampersand as prefix.

```
ParameterExt  ::= 'Parameter'
        Parameter
        'EndParameter'
```

Now, for the states, also named locations, of our system, we first incorporate the 'ini', 'com' and 'urg' words. 'ini' is for Initial and it must be declared only once in our specification since it is our initial location in the automata. 'com' stands for Committed, which is a state were time is not allowed to pass and the next transition in the system must point to a location that is in-bounded by the committed state. 'urg' is for Urgent, an urgent location is the one where, as committed locations, time freezes, but only that. This is equivalent as well as to adding an extra clock c that is reset on every incoming edge, and adding an invariant x <= 0 to the location.

The 'loc' word stands for Location, it indicates that the following constructs will be part of a state. The only construct that is mandatory is the first identifier, which will be the id of our location. The second identifier is related to the displayed name of the location in the automata diagram. Then we can declare an invariant, which is an expression, however, the type checker of UPPAAL restricts the type of expressions that can be used here. Finally, we can also declare an exponential rate, this is also a part of UPPAAL 4.1, and the idea is that when there is no upper bound for delay, then the number given by the exponential rate is chosen as delay based on probability distribution on that rate.

```
State      ::= 'ini'?  ('com' | 'urg')? 'loc'
           ID (',' ID)?
           Invariant? (':' ExponentialRate)? ';'
```

In our grammar, a branch point will only need an identifier to know it's unique name in the model.

```
BranchPoint    ::= 'branch' ID ';'
```

In the following rule we can see that we incorporated the influence of PlantUML to describe the transitions in the form of an already declared identifier representing the outbound location pointing to the inbound location. Then, optionally, we can use the colon symbol (another construct from PlantUML) to add the necessary labels for our transition.

```
Transition  ::= ID -> ID (':' Label (',' Label)*)? ';'
```

A label in PlantNF is part of a transition, however, in the newest version of UPPAAL, labels can be seen as well inside the construction of a location, this will be seen in examples of our transformations in chapter 5.

Each type of label contains a semantically distinct type of expression:

- select: as we can see in the grammar, this construct contains a set of comma separated name : type expressions.
- guard: this is another expression that is finally checked by UPPAAL. Between the expressions that are admitted to a guard are boolean operations that are referenced to clocks, integer variables and constants.
- sync: the value contains an expression followed by the symbol '!' or '?' the synchronisation element is used between two different process to allow to fire the edges with the same name (but different symbol) at the same time.
- assign: also known as update, is another set of expressions separated by comma, these expressions are referred to clocks, integer variables and constants.
- comments: they don't have any semantic meaning in UPPAAL
- probability: this label can be used when using probabilistic branches, they give weight to possible choices similarly as MoDeST does it with the **palt** construct.

```
Label      ::= 'select :' SelectList
           | 'guard :' Expression
           | 'sync :' Expression ('!' | '?')
           | 'assign :' Expression (',' Expression)*
           | 'comments :' COMMENT
           | 'probability :' Expression
```

The rest of the grammar specifies the language used in UPPAAL [4]. Which because of timing constraints we could not arrive to integrate into our editor solution but is necessary as part of our grammar, which is why we include it here.

```
Declarations  ::= (VariableDecl | TypeDecl | Function | ChanPriority)*
VariableDecl  ::= Type VariableID (',' VariableID)* ';'
VariableID    ::= ID ArrayDecl*  ('=' Initialiser)?
Initialiser   ::= Expression
                  | '{' Initialiser (',' Initialiser)* '}'
TypeDecls     ::= 'typedef' Type ID ArrayDecl* (',' ID ArrayDecl*)* ';'

Function    ::= Type ID '(' Parameters ')' Block
Block     ::= '{' Declarations Statement* '}'
Statement     ::= Block
```

```
11          | ';'
12            | Expression ';'
13          | ForLoop
14          | Iteration
15          | WhileLoop
16          | DoWhileLoop
17          | IfStatement
18          | ReturnStatement
19 ForLoop      ::= 'for' '(' Expression ';' Expression ';' Expression ')' Statement
20 Iteration    ::= 'for' '(' ID ':' Type ')' Statement
21 WhileLoop      ::= 'while' '(' Expression ')' Statement
22 DoWhile      ::= 'do' Statement 'while' '(' Expression ')' ';'
23 IfStatment     ::= 'if' '(' Expression ')' Statement ( 'else' Statement )?
24 ReturnStatement ::= 'return' Expression? ';'
25
26 ChanPriority  ::= 'chan' 'priority' (ChanExpr | 'default') ((',' | '<') (ChanExpr | '
       default'))* ';'
27 ChanExpr     ::= ID
28                 | ChanExpr '[' Expression ']'
29
30 Type       ::= Prefix TypeId
31 Prefix       ::= 'urgent' | 'broadcast' | 'meta' | 'const'
32 TypeId       ::= ID | 'int' | 'clock' | 'chan' | 'bool'
33                   | 'int' '[' Expression ',' Expression ']'
34                   | 'scalar' '[' Expression ']'
35                   | 'struct' '{' FieldDecl (FieldDecl)* '}'
36 FieldDecl      ::= Type ID ArrayDecl* (',' ID ArrayDecl*)* ';'
37 ArrayDecl      ::= '[' Expression ']'
38                   | '[' Type ']'
39
40 Parameters     ::= (Parameter (',' Parameter)* )?
41 Parameter      ::= Type  '&'? ID ArrayDecl*
42
43 System       ::= 'system' ID ((',' | '<') ID)* ';'
44
45 Expression     ::= ID
46                   | INT
47                   | Expression '[' Expression ']'
48                   | '(' Expression ')'
49                   | Expression '++' | '++' Expression
50                   | Expression '--' | '--' Expression
51                   | Expression Assign Expression
52                   | Unary Expression
53                   | Expression Binary Expression
54                   | Expression '?' Expression ':' Expression
55                   | Expression '.' ID
56           | Expression '(' Arguments ')'
57           | 'forall' '(' ID ':' Type ')' Expression
58           | 'exists' '(' ID ':' Type ')' Expression
59                   | 'deadlock' | 'true' | 'false'
60 Arguments      ::= ( Expression ( ',' Expression )* )?
61 Assign       ::= '=' | ':=' | '+=' | '-=' | '*=' | '/=' | '%='
62           | '|=' | '&=' | '^=' | '<<=' | '>>='
63 Unary        ::= '+' | '-' | '!' | 'not'
64 Binary       ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
65                   | '+' | '-' | '*' | '/' | '%' | '&'
66                   | '|' | '^' | '<<' | '>>' | '&&' | '||'
67           | '<?' | '>?' | 'or' | 'and' | 'imply'
68
69 SelectList     ::= ID ':' Type
70                   | SelectList ',' ID ':' Type
```

Listing 4.1: Listing of the grammar that is used in UPPAAL and that we took into PlantNF

4.3 Examples

This section contains three examples that will illustrate the application of our rules in order to construct models of timed automata. The first and second examples are systems that we can find as demo in UPPAAL. The third one is a trivial diagram constructed with the stochastic features.

4.3.1 Vikings

```
Declarations
/*
 * Four vikings are about to cross a damaged bridge in the middle of the
 * night. The bridge can only carry two of the vikings at the time and to
 * find the way over the bridge the vikings need to bring a torch.  The
 * vikings need 5, 10, 20 and 25 minutes (one-way) respectively to cross
 * the bridge.

 * Does a schedule exist which gets all four vikings over the bridge
 * within 60 minutes?
 */

chan take, release;   // Take and release torch
int[0,1] L;   // The side the torch is on
clock time;   // Global time
EndDeclarations
```

Listing 4.2: Declarations of the vikings model label

The following snippet of code describes the general behaviour of the vikings. As we can see, the first location we use is the one named with id3, which is named as unsafe. We also put a parameter delay, which we will use to simulate the speed behaviour of our vikings. There also an internal clock that will allow the guard to measure when the viking can make the transition from one point to another.

```
Template Soldier
Parameters
const int delay
EndParameters
Declarations
clock y ;
EndDeclarations
loc id0 ;
loc id1 , safe ;
loc id2 ;
ini loc id3 , unsafe ;
id2 -> id3 :
  guard : y >= delay ,
  sync : release! ;
id1 -> id2 :
  guard : L == 1 ,
  sync : take! ,
  assign : y = 0 ;
id0 -> id1 :
  guard : y >= delay ,
  sync : release! ;
id3 -> id0 :
  guard : L == 0 ,
  sync : take! ,
  assign : y = 0 ;
EndTemplate
```

Listing 4.3: Template of the viking automata

The torch template describes how many vikings are using the torch, both the torch and the viking templates coordinate each other by using the take and release synchronisation statements.

```
Template Torch
loc id4 , one ;
urg loc id5 ;
ini loc id6 , free ;
loc id7 , two ;
id6 -> id5 :
  sync : take? ;
id5 -> id4 ;
id5 -> id7 :
  sync : take? ;
id4 -> id6 :
  sync : release? ,
  assign : L = 1 - L ;
id7 -> id4 :
  sync : release? ;
EndTemplate
```

Listing 4.4: Template of the torch automata

Finally, the system is defined as four processes, or vikings, that are using only one torch process. As we can see, the argument in each process describes the speed of the vikings.

```
System
const int fastest = 5; const int fast = 10; const int slow = 20; const int slowest = 25;
    Viking1 = Soldier(fastest); Viking2 = Soldier(fast); Viking3 = Soldier(slow); Viking4 =
     Soldier(slowest); system Viking1, Viking2, Viking3, Viking4, Torch;
EndSystem
```

Listing 4.5: System declaration for the vikings model

4.3.2 Train gate

This is a railway control system that manipulates the access to the gate for many trains. The system is defined with a number of trains (in this example, there are 6 trains), and a gate. The gate is a critical shared resource that has to be used only by one train at a time. The actions of each train, like moving or stopping take a delay in time.

```
Declarations
  const int N = 6;
  typedef int[0, N-1] id_t;

  chan appr[N], stop[N], leave[N];
  urgent chan go[N];
EndDeclarations

Template Train
  Parameters
    const id_t id
  EndParameters
  Declarations
    clock x
  EndDeclarations
  ini loc safe, Safe ;
  loc cross, Cross x <= 5 ;
  loc appr, Appr x <= 20 ;
  loc start, Start x <= 15 ;
  loc stop, Stop ;
  safe -> appr :
    sync: appr[id]! ,
    assign: x = 0 ;
  appr -> stop :
    guard: x <= 10 ,
    sync: stop[id]? ;
  stop -> start :
    sync: go[id]? ,
```

```
    assign: x = 0 ;
  start -> cross :
    guard: x >= 7 ,
    assign: x = 0 ;
  cross -> safe :
    guard: x >= 3 ,
    sync: leave[id]! ;
  appr -> cross :
    guard: x>= 10 ,
    assign: x=0 ;
EndTemplate

Template Gate
  Declarations
    id_t list[N+1];
    int[0,N] len;

    //Put an element at the end of the queue
    void enqueue(id_t element)
    {
      list[len++] = element;
    }

    //Remove the front element of the queue
    void dequeue()
    {
      int i = 0;
      len -= 1;
      while (i < len)
      {
        list[i] = list[i + 1];
        i++;
      }
      list[i] = 0;
    }

    //Returns the front element of the queue
    id_t front()
    {
      return list[0];
    }

    //Returns the last element of the queue
    id_t tail()
    {
      return list[len - 1];
    }
  EndDeclarations
  ini loc free, Free ;
  loc occ, Occ ;
  com loc id0 ;
  free -> occ :
    guard: len > 0 ,
    sync: go[front()]! ;
  free -> occ :
    select: e:id_t ,
    guard: len == 0 ,
    sync: appr[e]? ,
    assign: enqueue(e) ;
  occ -> free :
    select: e:id_t ,
    guard: e==front() ,
    sync: leave[e]? ,
    assign: dequeue() ;
  occ -> id0 :
```

```
94      select: e:id_t ,
95      sync: appr[e]? ,
96      assign: enqueue(e) ;
97    id0 -> occ :
98      sync: stop[tail()]! ;
99  EndTemplate
100
101 System
102   system Train, Gate;
103 EndSystem
```

Listing 4.6: Code of the train gate system modelled in PlantNF

4.3.3 Stochastic example

This example is a trivial one that displays the capacity of our language to display locations with exponential rate and transitions from branches to nodes with a probabilistic weight. This will be illustrated further in chapter 5

```
1  Declarations
2
3  EndDeclarations
4
5  Template Test
6      loc id0 ;
7      loc id1 ;
8      loc id2 , example : 5 ;
9      ini loc id3 ;
10     branch id4 ;
11     branch id5 ;
12     id5 -> id1 ;
13     id4 -> id0 ;
14     id5 -> id2 :
15         select: test ,
16         assign: test ,
17         probability: 0.5 ;
18     id4 -> id1 ;
19     id3 -> id4 ;
20     id3 -> id5 ;
21 EndTemplate
22
23 System
24     system Process;
25 EndSystem
```

4.4 The tool

The solution we are presenting right now is an effort to help into developing models in our language. The idea is to create an editor that will highlight the language constructions like in any contemporary IDE that will facilitate the scripting of PlantNF files. This solution has started using the help of a plug-in for Eclipse called xtext.

Xtext is an open source language development framework that can be used to create domain specific languages or general purpose programming languages. This tool gives the necessary to create an Eclipse-based development environments in a very few steps. The created languages in xtext can be run in the Java Virtual Machine and, according to their creators, it offers great configurability. After downloading the necessary to run the plug-in in Eclipse we created a project in this environment which has the following structure.

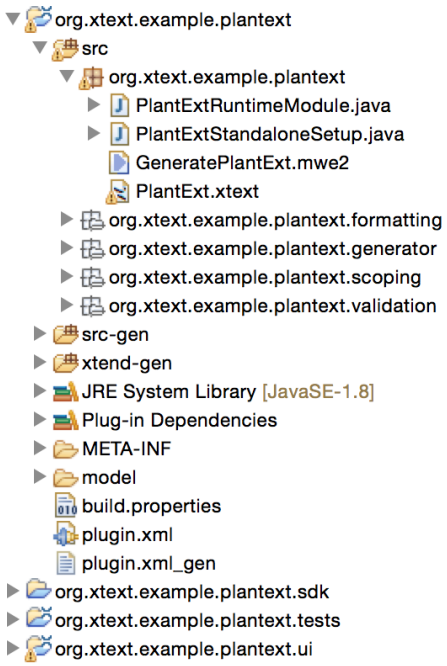


Figure 4.1: Structure of the xtext project for the PlantNF language

The user interface to create our scripts it's practically a fully functional copy of Eclipse that includes features like code formatting, code generator, a parser and linker. The following image shows how this environment is executed from within the xtext project. The result is an Eclipse interface that changes the colour of the structures and reserved words of our language. It also highlights sections of the code that don't comply with the established grammar.

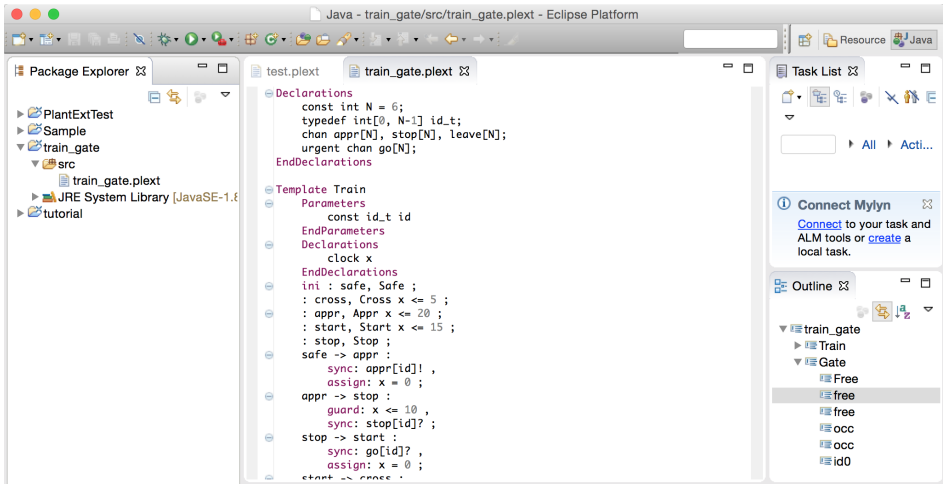


Figure 4.2: Screenshot of the user interface provided for our project

The specified grammar to make this project work is the following. Because of timing constrains we only arrived to get into the second stage of the implementation of the grammar for this project. This means that we can accept the constructs defined in our grammar in section 4.2 but the content that comes from UPPAAL is defined in this case a combination of any characters. In spite of this, we

can always check the correctness of our expressions' syntax and semantics by using UPPAAL once we transform our PlantNF scripts.

```
grammar org.xtext.example.plantext.PlantExt with org.eclipse.xtext.common.Terminals

generate plantext "http://www.xtext.org/example/plantext/PlantExt"

Document:
    declaration = Declaration
    (template += Template)+
    system = System
    ;

Declaration:
   'Declarations'
     (text += TEXT)*
   'EndDeclarations'
;

Template:
   'Template'
     name = ID (parameters = Parameter)? (declaration = Declaration)?
     (state += State)+
     (branchpoint += Branchpoint)*
     (transition += Transition)+
   'EndTemplate'
;

System:
   'System'
      (text += TEXT)*
   'EndSystem'
;

Parameter:
   'Parameters'
     (text += TEXT)*
   'EndParameters'
;

State:
   'ini'? ('com' | 'urg')? ':' id = ID (',' name = ID)?
   (text += TEXT)* (':' (exponentialRate += TEXT)*)? ';'
;

Branchpoint:
   name = ID ';'
;

Transition:
   outBoundState = ID '->' inboundState = ID (':'
     ('select' ':' (select += TEXT)+)? ','?
     ('guard' ':' (guard += TEXT)+)? ','?
     ('sync' ':' (sync += TEXT)+)? ','?
     ('assign' ':' (assign += TEXT)+)? ','?
     ('probability' ':' (probability += TEXT)+)? ','?
     ('comments' ':' (comments += TEXT)+)?
   ) ';'
;

TEXT: (ANY_OTHER | ID | INT | ';' | ',');
```

Listing 4.7: The grammar implemented into an xtext project

Chapter 5

Second Part: Transformation

5.1 Introduction

This chapter will explain the second solution proposed for the project. This solution consist in a PlantNf - UPPAAL model transformation tool that help us demonstrate that our language can provide the necessary to design models for non-functional requirements. As a matter of software design and implementation time, this project was the one that took the most of it.

5.2 Architecture of the solution

This section of the document presents the results of extraction of the code created during the process of implementation and its goal is to be the reference for future consultation on the solution.

5.2.1 Functional requirements

- The system must accept as input a file containing a script written in the language we created.
- The output has to be an XML file that can be read by the user interface of UPPAAL for further verification and validation.

5.2.2 Non-Functional requirements

- Reliability

  The language transformation tool should have an error ratio of 10% on its first month of testing. Decreasing gradually to a desirable 2% or less.

- Maintainability and Modifiability

  The system should be designed having loose coupling and high cohesion. This can be done by practising clean code principles [31]. This will assure that detection and repairing of bugs and further modifications will not produce cluttered code in the future.

5.2.3 Views

This section contains the models of the system expressed in the UML. This will help as a guideline to detect the implemented features of the solution.

Logical View

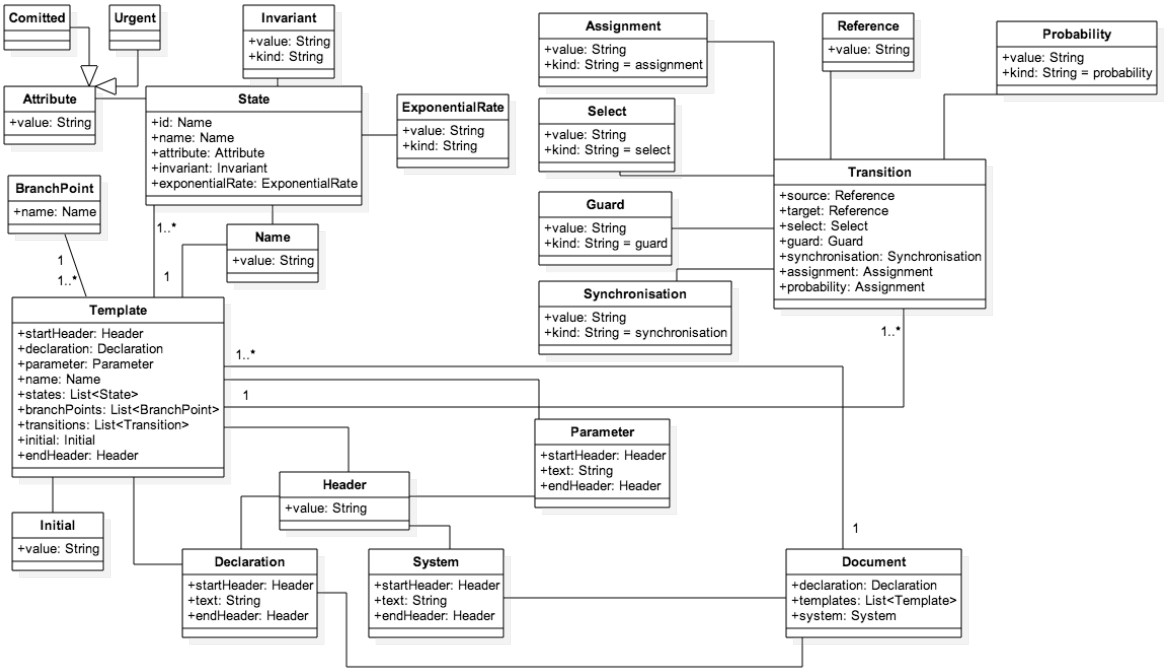


Figure 5.1: Class diagram representing the language elements of the solution

The reason behind the model seen in Figure 5.1 is to provide our transformation tool with a resource that can be equivalent to the language used in UPPAAL. Here we are going to identify the elements and attributes we can find in an UPPAAL XML file. We have to take special mention on the newest elements and attributes integrated into UPPAAL that add stochastic probability to the timed automatas, these are **branchpoint** and *probability* (note that bold names are used for elements and italic names are used for attributes):

- **nta**: this element is the root of the document.
- **declaration**: this element can appear once or multiple times in a document. The first appearance is obligatory and contains the *global* variables and clock declarations of the system. The rest of the occurrences are contained in each *template* element.
- **template**: this element represents each timed automata in the system. it is composed by the following elements: **name**, **parameter**, **declaration**, **locations**, **transitions**, **branchpoints** and **init**.
- **name**: this element contains the name of a location or a template, its attributes are x and y, which indicate the coordinates where the element will appear in UPPAAL's graphical user interface. Both attributes appear in other elements of the language and their function is the same.
- **parameter**: this element contains the parameters that we will use in each template.
- **location**: this element represent each one of the states of the diagram, it can contain the following elements: **name**, **committed** or **urgent**. It also contains the following attributes: *id*, x, y. *id* is an unique identifier of the **location**.

- **committed**: it is a self closing xml element that indicates that the state is committed. This means that "a committed state cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations" [10].
- **urgent**: it is another self closing xml element that indicates that the state is urgent. This means that the state cannot allow time to pass, it can be thought as having a guard with a value less than - equals 0. e.g. "clock c $\leq$ 0".
- **init**: it is a xml element that indicates the initial location, it is also self closing and it contains one attribute: *ref* which is the identifier of our initial location.
- **transition**: this xml element represents the transition from one location to another. This element contains the following elements: **source**, **target** a set of **labels** that depend on number and kind to the type of transition we use. If the transition is from location to location, these labels can have the following kinds: *select*, *guard*, *synchronisation*, *assignment*. Else, if the transition is from a branchpoint to location, the kinds can be: *select*, *assignment* and *probability*.
- **source**: is the element that indicates in its attribute *ref* the id of the location where the transition is outbound.
- **target**: is the element that indicates in its attribute *ref* the id of the location where the transition is inbound.
- **label**: this is a label that has many meanings, depending on the type of attribute it contains.
  - *guard*: this value is evaluated to a boolean. Only clocks, integer variables, and constants are referenced.
  - *select*: this value makes the label to contain a comma separated list of name : type expressions.
  - *synchronisation*: this value makes the value to contain an expression of the form "expression!" or "expression?".
  - *assignment*: it is a comma separated set of expressions that must only refer to clocks, integer variables, and constants and only assign integer values to clocks.
  - *probability*: when using a probabilistic branching, this attribute provides the weight into a choice.
- **system**: This element describes the definition of the system model. This model can be formed by one or more concurrent processes, local and global variables and channels.

Development View

Figure 5.2: Component diagram of the model transformation solution

The analyser will be an independent component that provides access to its service by an interface. The user can have the possibility to access this component by a graphical user interface or command-line. For this iteration we have opted for the second solution.

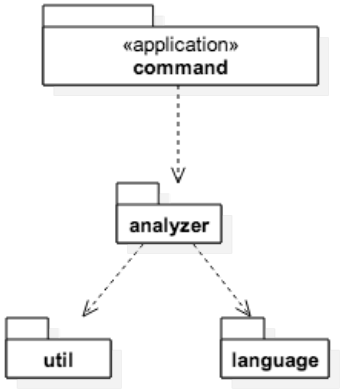


Figure 5.3: Package diagram of the model transformation solution

As stated before, the application will be operated by using the command line. In Figure 5.3 we also can see that the analyser uses two packages, one called util and other called language. Util contains methods that can be reused during the development of the system. Language is the package containing the classes exposed in Figure 5.1 and also some builder classes required by the analyser.

Process View

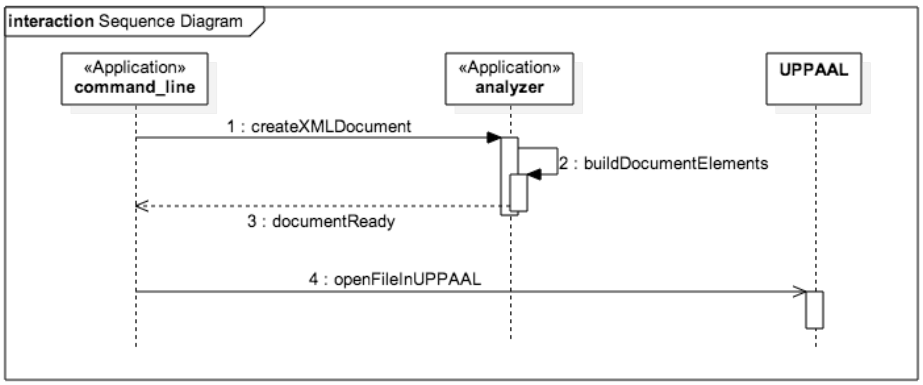


Figure 5.4: Sequence diagram representing the steps to transform the model and show the output in UPPAAL

The sequence diagram from Figure 5.4 shows the process that takes the system to transform the model done in our language to a model that can be opened and analysed by UPPAAL. However we would like to add that previous to that process we can use the tool that we implemented to create the model in our language with guidelines, like in the Eclipse IDE.

Physical View

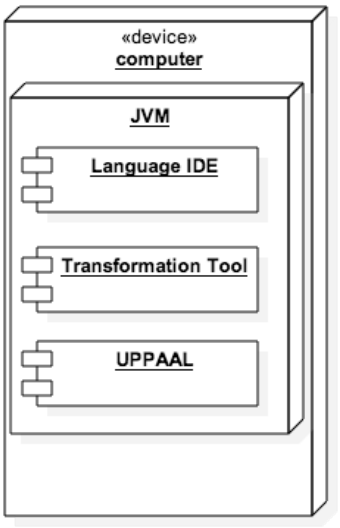


Figure 5.5: Deployment diagram representing the components that operate in our solution

The deployment diagram used in Figure 5.5 shows that the application can be used in any computer that has installed the Java Virtual Machine. Each of the components can be used separately in different time.

5.3 Implementation

The application is using a Object Oriented Traditional architectural style. To comply with this, the language to create the application is Java. The version of the Java Development Kit (JDK) is 8 and we are using the standard Java Virtual Machine.

The main project, which is the model transformation system, was made using the IntelliJ IDE. This decision was made after having bad experiences with NetBeans and finding that Eclipse was not user friendly because of its big capacity to be configurable. IntelliJ so far has proven to be a lightweight version of Eclipse with many of its benefits already out of the box.

The project also uses git as version control system (VCS). Having a version of every step we take in our implementation reduces the risk of wasting time if something goes wrong with our implementation and bugs found during testing are difficult to discover. By having each step saved in our version control system, we can roll-back to the step where our system was working properly without too much code and time to sacrifice. The advantage of git is that there is that by having our own copy of the project using that VCS, we have our own repository, which differs from other VCS like Subversion where the repository is centralised. However, having git as VCS carries the complexity of a distributed paradigm. In spite of that, once that the differences between the local and the remote operations have been understood, git is a nice tool to work with. There's also a public website (Github) were we can have our projects if we want to have them as open source. This project will be available in github to be downloaded in order to integrate more features in the future.

There were two main iterations in the development process of the solution:

- The first iteration of the project consisted in creating most of the system functionality and tests that relate to the required specifications previously mentioned in this chapter. Specifically, we

created the Analyser class and all the necessary data classes that could be used as a meta model that could help achieve the goal of creating a XML-input document for UPPAAL version 4.0.

- For the second iteration of the project we added the necessary classes to the meta model of the language and the consequent methods to the Analyser class. The goal in this iteration was to create an XML file that could be read by UPPAAL in its version 4.1

5.3.1 Structure and Contents of the System

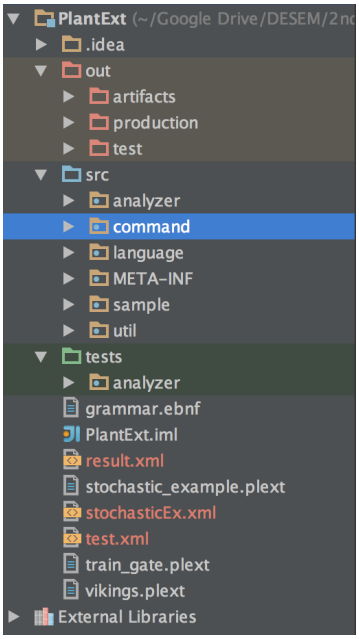


Figure 5.6: Image of the structure of the project from the used IDE

The project structure is constructed in accordance to the development view and the logical view. By looking at the Image 5.6, the folder src contains our desired structure. There we can find the analyser component, the command line component, the language meta model and the utils like the regex expression checker used to evaluate expressions used in PlantNF code.

Besided the classes that represent the meta model of the PlantNF, there are two classes implemented in this system that perform the needed operations to detect the language constructs and to construct the desired XML output. These classes are: AnalyzerImpl and XMLMarshaller. There is also an interface for the analyser implementation class that has only declared the method *createXMLDocument()*.

The following fragment of code contains the constructor of the AnalyzerImpl class, as well as the function that it's called through the interface that this class is implementing. We also included in this fragment one of the most complex methods in the system, in here we can see that we tried to use a builder design pattern to construct a PlantNF-UPPAAL's Template. Comments on the construction of this class will be made further in Chapter 7.

```
1    public AnalyzerImpl(String path) throws IOException {
2        this.documentBuilder = new DocumentBuilder();
3        this.tokens = new ArrayList<>();
4        this.templateTokensList = new ArrayList<>();
5        this.stream = Files.lines(Paths.get(path));
6    }
```

```
7
8       public Document createXMLDocument() {
9           stream.forEach(s -> getTokens(s));
10  //          tokens.forEach(s -> System.out.println(s));
11          if (verifyAnnotationsExist())
12          {
13              getTemplates();
14              buildAnnotationMainSection("Declarations");
15              buildAnnotationMainSection("System");
16              buildTemplate();
17              setDocument(documentBuilder.createDocument());
18              return getDocument();
19          }
20          else {
21
22          }
23          return null;
24      }
25
26      /**
27       * Construct a Template object from the statements inside Template and EndTemplate
28       */
29      private void buildTemplate() {
30
31          TemplateBuilder templateBuilder = new TemplateBuilder();
32          List<Template> templates = new ArrayList<>();
33
34          for (List<String> templateTokens : templateTokensList) {
35
36              Header templateAnnotation =
37                      Header.createHeader(templateTokens.get(templateTokens.indexOf(
    TEMPLATE_ANNOTATION)));
38              templateBuilder.setAnnotation(templateAnnotation);
39
40              Name name = Name.createName(templateTokens.get(templateTokens.indexOf(
    TEMPLATE_ANNOTATION) + 1));
41              templateBuilder.setName(name);
42
43              Header endTemplateAnnotation =
44                      Header.createHeader(templateTokens.get(templateTokens.indexOf("
    EndTemplate")));
45              templateBuilder.setEndAnnotation(endTemplateAnnotation);
46
47              if (templateTokens.indexOf(DECLARATIONS_ANNOTATION) > INDEX_NOT_FOUND) {
48                  buildAnnotationTemplateSection("Declarations", templateTokens,
    templateBuilder);
49              }
50
51              if (templateTokens.indexOf(PARAMETERS_ANNOTATION) > INDEX_NOT_FOUND) {
52                  buildAnnotationTemplateSection("Parameters", templateTokens,
    templateBuilder);
53              }
54
55              templateBuilder.setStates(getStates(templateTokens));
56              templateBuilder.setBranchPoints(getBranchPoints(templateTokens));
57              templateBuilder.setTransitions(getTransitions(templateTokens));
58              templateBuilder.setInitial(getInitial(templateTokens));
59
60              Template template = templateBuilder.createTemplate();
61              templates.add(template);
62              templateBuilder = new TemplateBuilder();
63          }
64
65          documentBuilder.setTemplates(templates);
```

```
66      }
```

Listing 5.1: Code fragment of the AnalyzerImpl class

The next code snippet contains the implementation of the XMLMarshaller class that transforms the Document object obtained from the Analyser into an XML document. This is done by using the native libraries included in the SDK of Java that allows serialisation of objects by only placing the correct annotations in our data classes.

```
1  import javax.xml.bind.JAXBContext;
2  import javax.xml.bind.JAXBElement;
3  import javax.xml.bind.Marshaller;
4  import javax.xml.namespace.QName;
5  import java.io.File;
6
7  /**
8   * Created by jid on 05/05/15.
9   */
10 public class XMLMarshaller {
11
12     public static void exportDocumentAsXML(Document document, File file) throws Exception{
13
14         JAXBContext jaxbContext = JAXBContext.newInstance(Document.class, Urgent.class,
       Committed.class);
15         Marshaller marshaller = jaxbContext.createMarshaller();
16
17         marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
18         JAXBElement<Document> jaxbElement = new JAXBElement<Document>(new QName("nta"),
       Document.class, document);
19         marshaller.marshal(jaxbElement, System.out);
20         marshaller.marshal(jaxbElement, file);
21     }
22 }
```

Listing 5.2: Code fragment of the XMLMarshaller class

5.3.2 Testing

The implementation process was made trying to use an test-driven development approach. We used unit testing with JUnit for the two classes that we used to implement the logic of the system. This helped us since the beginning of the coding of the class since we could use our input files and get results without waiting for the main command line component to be ready. Comments on the results and testing will be made in chapters 6 and 7.

5.3.3 User Instructions

The application at this stage can be executed in the command line or console. We need to have installed Java in our operative system, preferably the version 8 which was the one we used to implement or system. Now the only thing we have to do is to execute the provided jar file using a line similar to the one provided here in our command line.

java -jar PlantNF.jar vikings.plext result.xml

The first parameter (vikings.plext) is the source file that we want our program to analyse. The second parameter is optional and it is the target xml file where our resulting transformation will be stored. If we don't provide a name, the system will store the results in a file called default.xml.

5.4 Results From Examples

The following are the resulting XML code that can be read in UPPAAL, these come from the example scripts we introduced in section 4.3. Comments on these results will be made in the Evaluation chapter.

5.4.1 Vikings

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<nta>
    <declaration>
/*
 * Four vikings are about to cross a damaged bridge in the middle of the
 * night. The bridge can only carry two of the vikings at the time and to
 * find the way over the bridge the vikings need to bring a torch.  The
 * vikings need 5 , 10 , 20 and 25 minutes (one-way) respectively to cross
 * the bridge.

 * Does a schedule exist which gets all four vikings over the bridge
 * within 60 minutes?
 */

chan take , release;  // Take and release torch
int[0,1] L;  // The side the torch is on
clock time;  // Global time
</declaration>
    <template>
        <name>Soldier</name>
        <parameter>
const int delay
</parameter>
        <declaration>
clock y ;
</declaration>
        <location id="id0"/>
        <location id="id1">
            <name>safe</name>
        </location>
        <location id="id2"/>
        <location id="id3">
            <name>unsafe</name>
        </location>
        <init ref="id3"/>
        <transition>
            <source ref="id2"/>
            <target ref="id3"/>
            <label kind="guard">y&gt;=delay</label>
            <label kind="synchronisation">release!</label>
        </transition>
        <transition>
            <source ref="id1"/>
            <target ref="id2"/>
            <label kind="guard">L==1</label>
            <label kind="synchronisation">take!</label>
            <label kind="assignment">y=0</label>
        </transition>
        <transition>
            <source ref="id0"/>
            <target ref="id1"/>
            <label kind="guard">y&gt;=delay</label>
            <label kind="synchronisation">release!</label>
        </transition>
        <transition>
```

```
            <source ref="id3"/>
            <target ref="id0"/>
            <label kind="guard">L==0</label>
            <label kind="synchronisation">take!</label>
            <label kind="assignment">y=0</label>
        </transition>
    </template>
    <template>
        <name>Torch</name>
        <location id="id4">
            <name>one</name>
        </location>
        <location id="id5">
            <urgent/>
        </location>
        <location id="id6">
            <name>free</name>
        </location>
        <location id="id7">
            <name>two</name>
        </location>
        <init ref="id6"/>
        <transition>
            <source ref="id6"/>
            <target ref="id5"/>
            <label kind="synchronisation">take?</label>
        </transition>
        <transition>
            <source ref="id5"/>
            <target ref="id4"/>
        </transition>
        <transition>
            <source ref="id5"/>
            <target ref="id7"/>
            <label kind="synchronisation">take?</label>
        </transition>
        <transition>
            <source ref="id4"/>
            <target ref="id6"/>
            <label kind="synchronisation">release?</label>
            <label kind="assignment">L=1-L</label>
        </transition>
        <transition>
            <source ref="id7"/>
            <target ref="id4"/>
            <label kind="synchronisation">release?</label>
        </transition>
    </template>
    <system>
 const int fastest = 5 ; const int fast = 10 ; const int slow = 20 ; const int slowest = 25
     ; Viking1 = Soldier(fastest) ; Viking2 = Soldier(fast) ; Viking3 = Soldier(slow) ;
    Viking4 = Soldier(slowest) ; system Viking1 , Viking2 , Viking3 , Viking4 , Torch ;
 </system>
</nta>
```

5.4.2 Train Gate

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<nta>
    <declaration>
  const int N = 6;
  typedef int[0 , N-1] id_t;

  chan appr[N], stop[N], leave[N];
  urgent chan go[N];
```

```
</declaration>
   <template>
       <name>Train</name>
       <parameter>
  const id_t id
 </parameter>
       <declaration>
  clock x
 </declaration>
       <location id="safe">
           <name>Safe</name>
       </location>
       <location id="cross">
           <name>Cross</name>
           <label kind="invariant">x&lt;=5</label>
       </location>
       <location id="appr">
           <name>Appr</name>
           <label kind="invariant">x&lt;=20</label>
       </location>
       <location id="start">
           <name>Start</name>
           <label kind="invariant">x&lt;=15</label>
       </location>
       <location id="stop">
           <name>Stop</name>
       </location>
       <init ref="safe"/>
       <transition>
           <source ref="safe"/>
           <target ref="appr"/>
           <label kind="synchronisation">appr[id]!</label>
           <label kind="assignment">x=0</label>
       </transition>
       <transition>
           <source ref="appr"/>
           <target ref="stop"/>
           <label kind="guard">x&lt;=10</label>
           <label kind="synchronisation">stop[id]?</label>
       </transition>
       <transition>
           <source ref="stop"/>
           <target ref="start"/>
           <label kind="synchronisation">go[id]?</label>
           <label kind="assignment">x=0</label>
       </transition>
       <transition>
           <source ref="start"/>
           <target ref="cross"/>
           <label kind="guard">x&gt;=7</label>
           <label kind="assignment">x=0</label>
       </transition>
       <transition>
           <source ref="cross"/>
           <target ref="safe"/>
           <label kind="guard">x&gt;=3</label>
           <label kind="synchronisation">leave[id]!</label>
       </transition>
       <transition>
           <source ref="appr"/>
           <target ref="cross"/>
           <label kind="guard">x&gt;=10</label>
           <label kind="assignment">x=0</label>
       </transition>
   </template>
```

```
74     <template>
75         <name>Gate</name>
76         <declaration>
77   id_t list[N+1];
78   int[0,N] len;
79
80   //Put an element at the end of the queue
81   void enqueue(id_t element)
82   {
83    list[len++] = element;
84   }
85
86   //Remove the front element of the queue
87   void dequeue()
88   {
89    int i = 0;
90    len -= 1;
91    while (i &lt; len)
92    {
93     list[i] = list[i + 1];
94     i++;
95    }
96    list[i] = 0;
97   }
98
99   //Returns the front element of the queue
100  id_t front()
101  {
102   return list[0];
103  }
104
105  //Returns the last element of the queue
106  id_t tail()
107  {
108   return list[len - 1];
109  }
110 </declaration>
111         <location id="free">
112             <name>Free</name>
113         </location>
114         <location id="occ">
115             <name>Occ</name>
116         </location>
117         <location id="id0">
118             <committed/>
119         </location>
120         <init ref="free"/>
121         <transition>
122             <source ref="free"/>
123             <target ref="occ"/>
124             <label kind="guard">len&gt;0</label>
125             <label kind="synchronisation">go[front()]!</label>
126         </transition>
127         <transition>
128             <source ref="free"/>
129             <target ref="occ"/>
130             <label kind="select">e:id_t</label>
131             <label kind="guard">len==0</label>
132             <label kind="synchronisation">appr[e]?</label>
133             <label kind="assignment">enqueue(e)</label>
134         </transition>
135         <transition>
136             <source ref="occ"/>
137             <target ref="free"/>
138             <label kind="select">e:id_t</label>
```

```
            <label kind="guard">e==front()</label>
            <label kind="synchronisation">leave[e]?</label>
            <label kind="assignment">dequeue()</label>
        </transition>
        <transition>
            <source ref="occ"/>
            <target ref="id0"/>
            <label kind="select">e:id_t</label>
            <label kind="synchronisation">appr[e]?</label>
            <label kind="assignment">enqueue(e)</label>
        </transition>
        <transition>
            <source ref="id0"/>
            <target ref="occ"/>
            <label kind="synchronisation">stop[tail()]!</label>
        </transition>
    </template>
    <system>
  system Train , Gate;
 </system>
</nta>
```

Listing 5.3: Resulting code after transformation of code

5.4.3 Stochastic example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<nta>
    <declaration>

 </declaration>
    <template>
        <name>Test</name>
        <location id="id0"/>
        <location id="id1"/>
        <location id="id2">
            <name>example</name>
            <label kind="invariant"></label>
            <label kind="exponentialrate">5</label>
        </location>
        <location id="id3"/>
        <branchpoint id="id4"/>
        <branchpoint id="id5"/>
        <init ref="id3"/>
        <transition>
            <source ref="id5"/>
            <target ref="id1"/>
        </transition>
        <transition>
            <source ref="id4"/>
            <target ref="id0"/>
        </transition>
        <transition>
            <source ref="id5"/>
            <target ref="id2"/>
            <label kind="select">test</label>
            <label kind="assignment">test</label>
            <label kind="probability">0.5</label>
        </transition>
        <transition>
            <source ref="id4"/>
            <target ref="id1"/>
        </transition>
        <transition>
            <source ref="id3"/>
```

```
40              <target ref="id4"/>
41          </transition>
42          <transition>
43              <source ref="id3"/>
44              <target ref="id5"/>
45          </transition>
46      </template>
47      <system>
48       system Process;
49  </system>
50 </nta>
```

Chapter 6

Evaluation

Our model transformation project has been tested by comparing the results obtained with diagrams already packed on UPPAAL and one diagram created by us to make the comparison with the one we wrote in PlantNF to draw probabilistic branching. The resulting diagrams obtained demonstrate that we can get the same structures with the same locations connected with their respective edges. However, our transformation tool lacks the capacity to write a system of coordinates in each element that could improve the order of the elements in the diagram. This can be noticed once that we try to open the XML files in UPPAAL. The diagrams can be correct according to the integrated tool and we can even perform some model checking queries on them, but, as seen in the following series of images, the aspect of the models can be different and might present some confusion at a first impression for the user. The left diagram of the following series images are the results obtained by our transformation tool, the right side diagrams are the originals from UPPAAL.

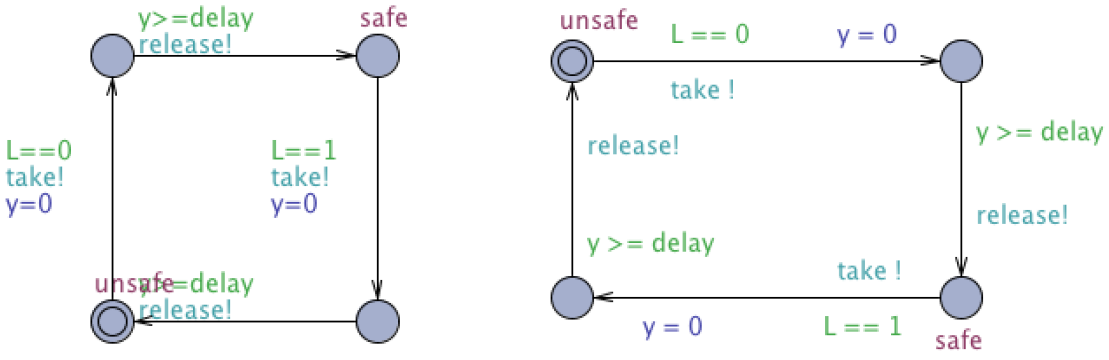


Figure 6.1: Comparison between our results and an UPPAAL made diagram of the vikings template

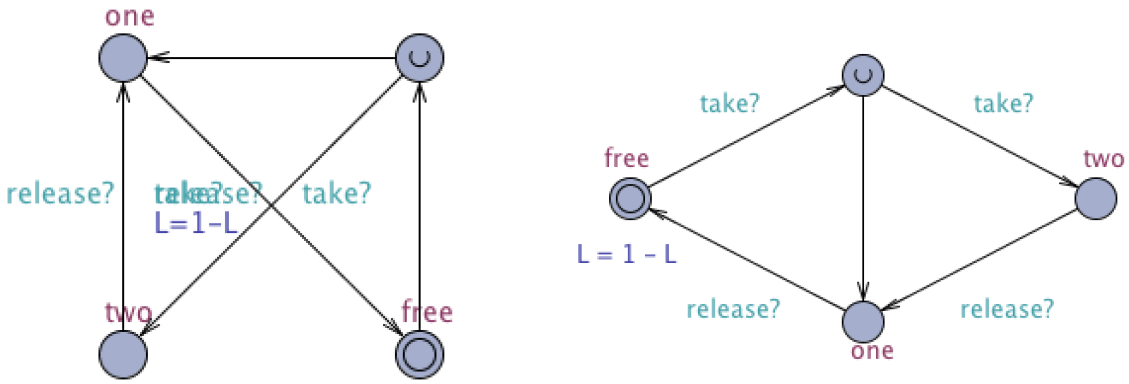


Figure 6.2: Comparison between our results and an UPPAAL made diagram of the torch template

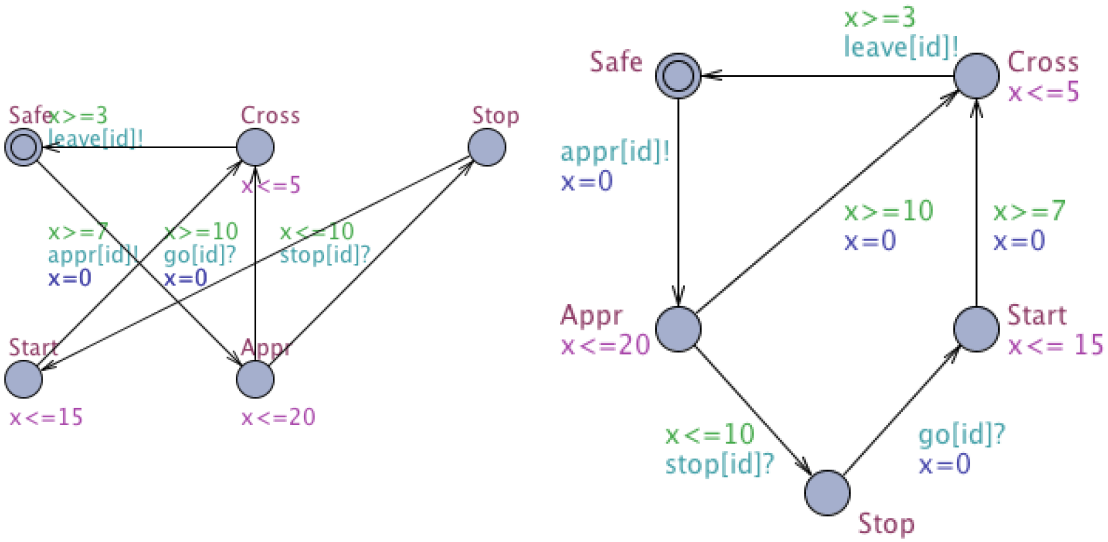


Figure 6.3: Comparison between our results and an UPPAAL made diagram of the train template

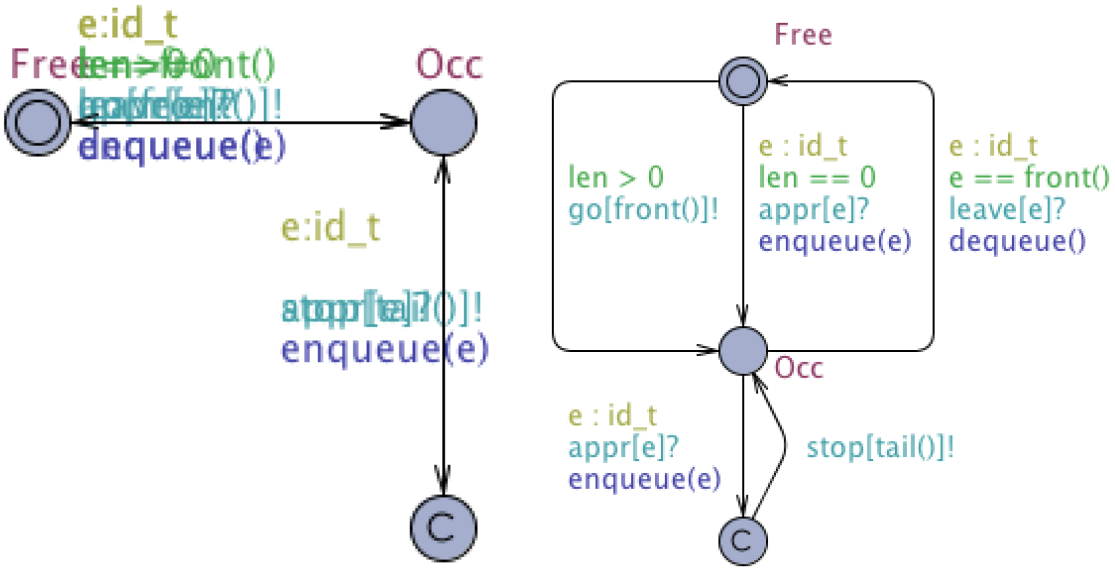


Figure 6.4: Comparison between our results and an UPPAAL made diagram of the gate template

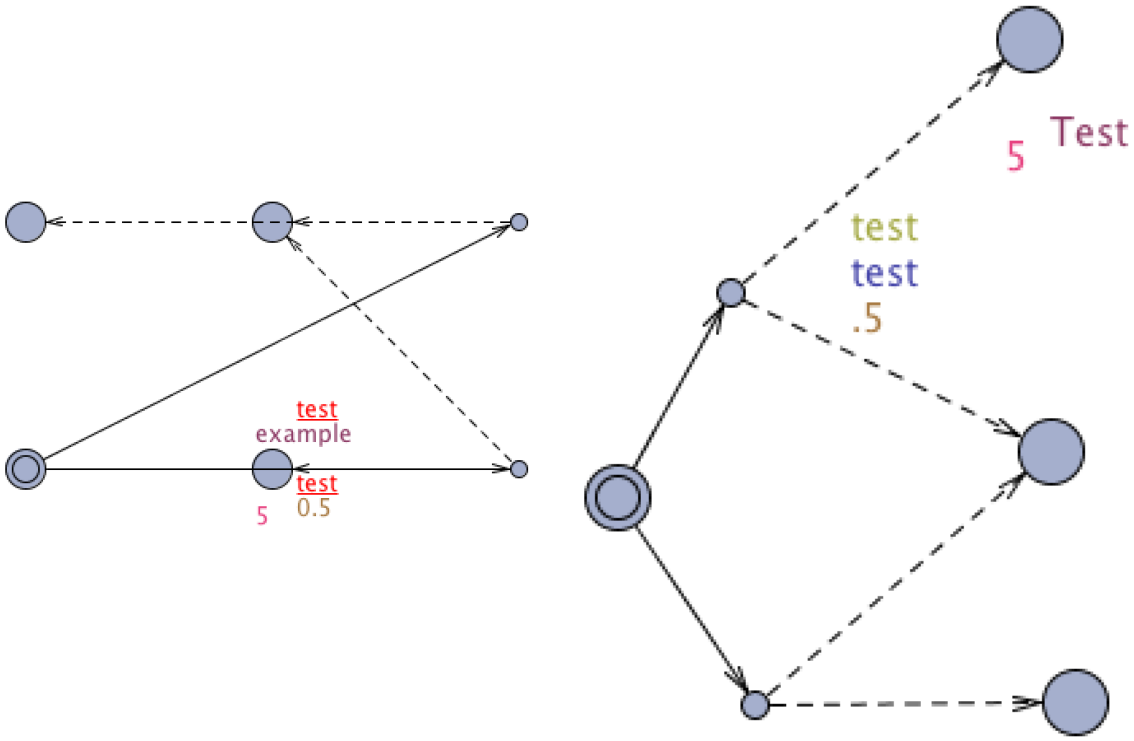


Figure 6.5: Comparison between our results and an UPPAAL made diagram of the stochastic example template

Chapter 7

Conclusion and future work

In this document we presented the problem of modelling non-functional requirements and make transformations between these models. We introduced our objectives and set of goals and proposed the idea of developing a new language that could bridge informal modelling with formal modelling.

We started the discussion of the subject by defining the concept of an embedded system and how they are part of our every day life as components of things we use very often like telephones, toys, cars or things that makes our life possible in case we have physical conditions, like pacemakers. The majority of these systems must react according to a bounded time constraint, because of having this timing commitment, these systems are known as real-time systems. We established the difference between soft, firm and hard real-time systems, in which the difference basically lies in the degree of consequences that a failure can carry into the entire system. In hard real time systems we need to take special care in the dependable attributes of our system. We explained some of these attributes, like availability, reliability, safety and maintainability and how could they be measured.

As a part of the background discussion we also talked about system modelling and the techniques used to abstract our systems and make them understandable for different types of stakeholders. We differentiated formal and informal system modelling by naming their respective characteristics and introduced the subject that concerns to us, which is model transformation and what are the considerations we have to take when we want to make such a transformation.

Having explained the background on our subject, we started the description of our project, in which we introduced the three languages and integrated tools that helped us pave our way into our solution. These were PlantUML, UPPAAL and MoDeST. Then we stated a brief foundation on the two projects we worked on.

Our first project was to propose a new language that would serve as an instrument to model non-functional requirements into networks of timed automata. This language name is PlantNF and we showed the constructs of this language and some examples that could help us as guidelines to create more models in this language. We also suggested the use of an editor tool we created using xtext.

The second project was the implementation of a PlantNF - UPPAAL model transformation tool. In the chapter dedicated to this tool we discussed the architecture of the system using different views and explaining the structure of the implemented system. We explained how to use the system by making the adequate execution of the jar file in the command line and the parameters it uses. Finally we showed the resulting output from our tool by using the examples provided in the previous chapter. We also made an evaluation of the system where we could see that our language works properly as long as we provide a correct input file. However there is still some work to do about the graphic presentation of the diagrams obtained from our transformations.

We consider that there is a lot of room for improvement and further work in our project, first of all, we could still manipulate our language and our transformation tool so we can target more languages and tools like PRISM, which is another probabilistic model checker that uses its own

language as well; or Tina, another model checking tool for Petri Nets. The editor project will also need an improvement in the grammar defined there.

Another improvement that can be done is the integration of queries constructs for UPPAAL. In their most recent version the .q files that were used for storing the queries for the model checker were integrated in their XML. Even though our language objective is to model systems, the fact that the model checking queries are now stored in the same XML as the model could make that our language - transformation tool can provide that service as well.

But before considering adding more features we recommend that performing some refactoring and code cleaning measures could help. Specially for the Analyser class, if we extract its methods in different classes and provide the necessary interfaces, we can have a nicely separated and distributed code that can be unit tested, which will give a better test coverage and prevent the creation of faulty code.

Bibliography

[1] *Quality Concepts and Terminology*. ISO, 1992.

[2] *Industrial-process measurement and control– evaluation of system properties for the purpose of system assessment – part 2: assessment methodology*. IEC, Geneva, 1993. The document can be consulted by contacting: ST: Y. Body.

[3] Plantuml. http://plantuml.sourceforge.net, 2015.

[4] Uppaal grammar. http://www.uppaal.com/index.php?sida=217&rubrik=101, 2015.

[5] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[6] Rajeev Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[7] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[8] Mario Barbacci, Mark Klein, Thomas Longstaff, and Charles Weinstock. Quality attributes. Technical Report CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1995.

[9] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[10] Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on uppaal. pages 200–236. Springer, 2004.

[11] Johan Bengtsson, Fredrik Larsson, Paul Pettersson, Wang Yi, Palle Christensen, Jesper Jensen, Per Jensen, Kim Larsen, and Thomas Sorensen. Uppaal: a tool suite for validation and verification of real-time systems, 1996.

[12] GÃl'rard Berry. Real time programming: Special purpose or general purpose languages. In *IFIP Congress*, pages 11–17, 1989.

[13] Lorenzo Bettini. A dsl for writing type systems for xtext languages. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 31–40, New York, NY, USA, 2011. ACM.

[14] Jonathan Bogdoll, Alexandre David, Arnd Hartmanns, and Holger Hermanns. Mctau: Bridging the gap between modest and uppaal. In *Proceedings of the 19th International Conference on Model Checking Software*, SPIN'12, pages 227–233, Berlin, Heidelberg, 2012. Springer-Verlag.

[15] Henrik Bohnenkamp, Pedro R. DâĂŹArgenio, Holger Hermanns, and Joost-Pieter Katoen. Modest: A compositional modeling formalism for hard and softly timed systems. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 32:812–830.

[16] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[17] C. Constant, T. JÃl'ron, H. Marchand, and V. Rusu. Validation of reactive systems. In S. Merz and N. Navet, editors, *Modeling and Verification of Real-TIME Systems - Formalisms and software Tools*, chapter 2, pages 51–76. HermÃĺs Science, January 2008.

[18] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. In *PROCEEDINGS OF THE IEEE*, pages 366–390, 1999.

[19] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Sci. Comput. Program.*, 48(1):21–42, 2003.

[20] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, Gordon Woodhull, Short Description, and Lucent Technologies. Graphviz âĂŤ open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag, 2001.

[21] John Erickson and Keng Siau. Can uml be simplified? practitioner use of uml in separate domains.

[22] David Garlan, Felix Bachmann, James Ivers, Judith Stafford, Len Bass, Paul Clements, and Paulo Merson. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition, 2010.

[23] D. Harel and A. Pnueli. Logics and models of concurrent systems. chapter On the Development of Reactive Systems, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[24] Michael Jastram and Prof Michael Butler. *Rodin User's Handbook: Covers Rodin V.2.8*. CreateSpace Independent Publishing Platform, USA, 2014.

[25] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[26] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, November 1995.

[27] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[28] Kevin Lano, Shekoufeh Kolahdouz Rahimi, and Iman Poernomo. Comparative evaluation of model transformation specification approaches. *Int. J. Software and Informatics*, 6(2):233–269, 2012.

[29] Phillip A. Laplante. *Real-Time Systems Design and Analysis: An Engineer's Handbook*. IEEE Press, Piscataway, NJ, USA, 1992.

[30] Edward A. Lee. Embedded software. In *Advances in Computers*, page 2002. Academic Press, 2002.

[31] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[32] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[33] David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[34] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, October 1992.

[35] Nick Rozanski and Eóin Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005.

[36] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal models for embedded system design. *Design Test of Computers, IEEE*, 17(2):14–27, Apr 2000.

[37] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[38] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[39] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons Ltd., Chichester, UK, 2nd edition edition, 2002.

[40] DÃąniel VarrÃş and Andras Pataricza. Automated formal verification of model tranformations.

[41] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *Software, IEEE*, 31(3):79–85, May 2014.

[42] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.